Владимирский государственный университет

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА i8086

Практикум

Владимир 2003

Министерство образования Российской Федерации

Владимирский государственный университет

И. А. ГОЛОВИНОВ

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА i8086

Практикум

Владимир 2003

УДК 681.3.06(076.5)
Г61

Рецензенты

Кандидат технических наук,
доцент Владимирского государственного педагогического университета

Ю. А. Медведев

Зав. кафедрой вычислительной техники, доктор технических наук,
профессор Костромского государственного технологического университета

М. Г. Левин

Печатается по решению редакционно-издательского совета
Владимирского государственного университета

Головинов И. А.

Г61 Программирование на языке Ассемблера i8086: Практикум / Владим. гос. ун-т. Владимир, 2003. 72 с.

ISBN 5-89368-406-0

Практикум содержит основные сведения о внутренней архитектуре процессора Intel 8086 и программировании на языке Ассемблера для этого процессора. Рассмотрены элементы структуры процессора, вопросы адресации памяти и организации ввода-вывода. Отдельное внимание уделено средствам модульного программирования на языке Ассемблера. По каждой теме приведены примеры, иллюстрирующие использование тех или иных средств.

Данный практикум предназначен для студентов специальностей 220100 "Вычислительные машины, комплексы, системы и сети" и 071900 "Информационные системы в бизнесе и менеджменте" первого курса обучения.

Табл. 7. Ил. 7. Библиогр.: 4 назв.

УДК 681.3.06(076.5)

ISBN 5-89368-406-0

1. ВВЕДЕНИЕ В АРХИТЕКТУРУ INTEL 8086 И ЯЗЫК АССЕМБЛЕРА

1.1. Общие сведения об архитектуре Intel 8086

1.1.1. Элементы внутренней структуры процессора Intel 8086

Процессор логически можно разделить на три части: арифметико-логический блок (АЛБ), управляющее устройство, блок сопряжения с шиной. АЛБ используется для выполнения элементарных операций обработки данных (арифметические и логические операции), управляющее устройство обеспечивает порядок выполнения набора операций АЛБ, в совокупности реализующих ту или иную команду процессора, а блок сопряжения с шиной управляет обменом данными с внешними устройствами, в том числе с памятью.

Процессор Intel 8086 имеет 14 16-разрядных регистров, которые используются для управления исполнением команд, адресации и выполнения арифметических и логических операций. Каждый регистр имеет собственное обозначение.

Сегментные регистры **CS** (Code Segment), **DS** (Data Segment), **SS** (Stack Segment), **ES** (Extra Segment) предназначены для адресации сегментов. Регистр **CS** содержит номер сегмента кода, **DS** - данных, **SS** - стека. Регистр **ES** используется для адресации памяти при выполнении операций со строками.

Регистры общего назначения **AX** (Accumulator), **BX** (Base), **CX** (Counter), **DX** (Data) являются основными рабочими регистрами и могут быть использованы по усмотрению программиста. Регистр **AX** - это первичный аккумулятор, он используется в качестве первого операнда и результата при выполнении арифметических и логических операций, регистр **BX** используется для хранения второго операнда, а также как базовый регистр при косвенной, относительной и индексной адресации, регистр **CX** используется при организации циклов, а регистр **DX** - при выполнении арифметических операций с 32-разрядными числами.

Регистры общего назначения допускают обращение не только к целому регистру, но также к младшему (обозначается буквой **L**) и старшему (обозначается буквой **H**) байтам по отдельности. Так, старший байт регистра **AX** обозначается через **AH**, а младший байт регистра **DX** - через **DL**.

Регистры-указатели **SP** (Stack Pointer) и **BP** (Base Pointer) используются для доступа к содержимому стека. Регистр **SP** всегда указывает на вершину

стека, а регистр **BP** используется для косвенной адресации стека при организации подпрограмм.

Индексные регистры **SI** (Source Index) и **DI** (Destination Index) используются при индексной адресации и в операциях обработки строк. Эти регистры могут также использоваться в операциях сложения и вычитания.

Регистр указателя команд **IP** (Instruction Pointer) всегда содержит адрес (смещение в сегменте кода) следующей выполняемой команды и используется для выборки очередной команды из памяти. Явное изменение значения регистра **IP** невозможно, однако он изменяется неявно при выполнении команды условного или безусловного перехода.

Регистр флагов содержит 9 активных битов (из 16) - флагов. Флаги делятся на условные (отражают результат последней выполненной операции) и управляющие (определяют выполнение некоторых специальных функций). Каждый флаг имеет собственное обозначение, как показано ниже. Их описание см. в табл. 1.1.

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

Таблица 1.1

Флаги процессора Intel 8086

Флаг	Название	Назначение
CF	Carry	Устанавливается в 1, если при выполнении операции возникает перенос или заем из старшего бита, в противном случае он сбрасывается в 0
PF	Parity	Устанавливается в 1, если младшие 8 бит результата операции содержат четное число единиц, в противном случае он сбрасывается в 0
AF	Auxiliary carry	Устанавливается в 1, если при выполнении операции сложения (вычитания) возникает перенос (заем) из бита 3; этот флаг используется в операциях двоично-десятичной арифметики
ZF	Zero	Устанавливается в 1, если результат операции равен 0, в противном случае он сбрасывается в 0
SF	Sign	Равен старшему биту результата; таким образом, этот флаг показывает знак результата операции
TF	Trap	Если этот флаг установлен в 1, то после выполнения каждой команды процессор генерирует внутреннее прерывание; этот режим используется при отладке программ

Окончание табл. 1.1

Флаг	Название	Назначение
IF	Interrupt	Если этот флаг установлен в 1, процессор распознает маскируемые прерывания, в противном случае маскируемые прерывания игнорируются
DF	Direction	Этот флаг используется командами обработки строк; если он установлен в 1, то строка обрабатывается начиная от последнего символа, в противном случае - от первого символа
OF	Overflow	Устанавливается в 1, если при выполнении операции возникло переполнение, в противном случае он сбрасывается в 0

В общем виде действия процессора включают в себя следующие шаги:

1) выборка следующей команды по адресу, определяемому номером сегмента (содержимым сегментного регистра **CS**) и смещением (содержимым регистра указателя команд **IP**);

2) загрузка команды в регистр команды и ее дешифрация с одновременным инкрементом регистра указателя команд **IP** для адресации следующей по порядку команды;

3) выполнение команды; в случае команды перехода - загрузка в сегментный регистр **CS** и в регистр указателя команд **IP** адреса перехода;

4) повторение шагов 1 - 3.

В процессоре Intel 8086 адрес следующей команды равен сумме **IP** и **CS×16**, а регистр команды представлен 6-байтной очередью FIFO (First In - First Out, первым пришел - первым ушел), которая непрерывно заполняется, когда системная шина не требуется для других операций. Такое опережение значительно увеличивает производительность процессора, т. к. к моменту завершения исполнения текущей команды следующая команда чаще всего уже находится в процессоре. В случае перехода очередь сбрасывается и не дает экономии времени исполнения, однако в среднем это происходит нечасто.

Длина команд составляет от одного до шести байт, но очередь позволяет считывать команды только словами по четным адресам. В случае же перехода по нечетному адресу процессор считывает сначала один байт, а затем продолжает считывать словами по четным адресам. Причем если в очереди остался свободным один байт, процессор не обратится к памяти до тех пор, пока в очереди не освободится целое слово.

1.1.2. Система команд процессора Intel 8086

Для программиста ключевым ресурсом вычислительной машины является система реализуемых ей команд. Каждая машинная команда разделяется на группы бит (поля): поле кода операции и одно или несколько полей операндов. Код операции показывает, что нужно делать, а операнды определяют необходимую команде информацию и могут содержать данное, адрес данного, косвенный указатель на данное или другую информацию, относящуюся к обрабатываемым командой данным. Каждой команде сопоставлена мнемоника. Процессор Intel 8086 реализует систему команд, показанную в табл. 1.2. Все команды в соответствии с выполняемыми ими функциями разделены по группам.

Таблица 1.2

Система команд процессора Intel 8086

Команда	Описание	Команда	Описание
Команды пересылки данных			
IN	ввод из порта	**POP**	извлечение из стека
LAHF	загрузка флагов в **AH**	**POPF**	восстановление флагов
LDS	загрузка через **DS**	**USH**	включение в стек
LEA	загрузка адреса	**PUSHF**	сохранение флагов
LES	загрузка через **ES**	**SAHF**	загрузка флагов из **AH**
MOV	пересылка	**XCHG**	обмен значениями
OUT	вывод в порт	**XLAT**	кодирование по таблице
Арифметические операции			
AAA	корр. после сложения	**DAS**	корр. после вычитания
AAD	корр. перед делением	**DEC**	декремент
AAM	корр. после умножения	**DIV**	беззнаковое деление
AAS	корр. после вычитания	**IDIV**	деление со знаком
ADC	сложение с переносом	**IMUL**	умножение со знаком
ADD	сложение	**INC**	инкремент
CBW	расширение байта	**MUL**	беззнаковое умножение
CMP	сравнение	**NEG**	дополнительный код
CWD	расширение слова	**SBB**	вычитание с заемом
DAA	корр. после сложения	**SUB**	вычитание
Логические операции			
AND	логическое умножение	**SAL**	арифм. сдвиг влево
NOT	логическое отрицание	**SAR**	арифм. сдвиг вправо
OR	логическое сложение	**SHL**	логич. сдвиг влево

Продолжение табл. 1.2

Команда	Описание	Команда	Описание
Логические операции			
RCL	сдвиг влево через **CF**	**SHR**	логич. сдвиг вправо
RCR	сдвиг вправо через **CF**	**TEST**	тест
ROL	цикл. сдвиг влево	**XOR**	исключающее ИЛИ
ROR	цикл. сдвиг вправо		
Команды условного перехода			
JA	выше	**JNGE**	не больше и не равно
JAE	выше или равно	**JNL**	не меньше
JB	ниже	**JNLE**	не меньше и не равно
JBE	ниже или равно	**JNO**	нет переполнения
JC	перенос	**JNP**	нечетно
JCXZ	**CX = 0**	**JNS**	положительный результат
JE	равно	**JNZ**	не ноль
JG	больше	**JO**	есть переполнение
JGE	больше или равно	**JP**	четно
JL	меньше	**JPE**	четно
JLE	меньше или равно	**JPO**	нечетно
JNA	не выше	**JS**	отрицательный результат
JNAE	не выше и не равно	**JZ**	ноль
JNB	не ниже	**LOOP**	переход по счетчику
JNBE	не ниже и не равно	**LOOPE**	переход пока равно
JNC	нет переноса	**LOOPNE**	переход пока не равно
JNE	не равно	**LOOPNZ**	переход пока не ноль
JNG	не больше	**LOOPZ**	переход пока ноль
Команды передачи управления			
CALL	вызов подпрограммы	**RET**	возврат из подпрограммы
JMP	безусловный переход		
Команды обработки строк			
CMPS	сравнение строк	**REPE**	повтор пока равно
CMPSB	из байтов	**REPNE**	повтор пока не равно
CMPSW	из слов	**REPNZ**	повтор пока не ноль
LODS	загрузка строки	**REPZ**	повтор пока ноль
LODSB	из байтов	**SCAS**	просмотр строки
LODSW	из слов	**SCASB**	из байтов
MOVS	пересылка строки	**SCASW**	из слов
MOVSB	из байтов	**STOS**	запись в строку

Окончание табл. 1.2

Команда	Описание	Команда	Описание
Команды обработки строк			
MOVSW	из слов	**STOSB**	из байтов
REP	повтор	**STOSW**	из слов
Команды прерывания			
INT	прерывание	**IRET**	возврат из прерывания
INTO	переполнение		
Команды управления состоянием процессора			
CLC	сброс флага **CF**	**LOCK**	блокировка шины
CLD	сброс флага **DF**	**NOP**	нет операции
CLI	сброс флага **IF**	**STC**	установка флага **CF**
CMC	инверсия флага **CF**	**STD**	установка флага **DF**
ESC	команда сопроцессору	**STI**	установка флага **IF**
HLT	останов процессора	**WAIT**	ожидание сопроцессора

1.1.3. Организация памяти

Наименьшей единицей данных, с которой работает компьютер, является бит (bit). Значением бита может быть либо ноль, либо единица. Группа из восьми битов называется байтом (byte) и представляет собой наименьшую адресуемую единицу - ячейку. Биты в байте нумеруют справа налево цифрами 0…7:

7	6	5	4	3	2	1	0

Каждому из байтов присвоен уникальный адрес памяти начиная с нулевого (самый младший адрес).

Двухбайтовое поле образует шестнадцатиразрядное машинное слово (word), биты в котором нумеруются справа налево цифрами 0…15:

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

Байт с меньшим адресом считается младшим. В архитектуре Intel 8086 принято, что менее значащий байт слова - младший (т. е. его адрес всегда меньше), более значащий байт слова старший (т. е. его адрес всегда больше).

В компьютере принята двоичная система представления данных. Символьная информация кодируется в соответствии с одним из определенных

стандартных кодов (например ASCII, ANSI, ISO, KOI). Числовые данные кодируются двоичным кодом. Отрицательные числа представляются в дополнительном коде. Для удобства представления данных человеку часто используется шестнадцатеричная система счисления.

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 обязательной является постановка нуля впереди, например **0BA8H**.

Слово адреса процессора Intel 8086 - 20 бит, поэтому этот процессор непосредственно может адресовать не более 2^{20} **= 1M** оперативной памяти. Адрес памяти в общем случае определяется парой 16-разрядных чисел: номером сегмента и величиной смещения, каждое из которых может изменяться в пределах от **0** до 2^{16} **= 64K**. Сегмент - это непрерывная область памяти объемом **64K**, адрес нижней границы которой кратен 16, а смещение - это номер байта внутри этой области.

Используя эту пару, процессор вычисляет целевой адрес так: номер сегмента сдвигается на 4 бита влево (что соответствует умножению на 16) и к результату прибавляется смещение, образуя 20-разрядный линейный адрес. Это позволяет 16-разрядному процессору Intel 8086 адресовать пространство оперативной памяти в 2^{20} **= 1M**, которое логически разделено на перекрывающиеся сегменты, каждый из которых имеет размер **64K** и начинается на 16-байтной границе.

Использование механизма сегментации обеспечивает несколько преимуществ:

1) емкость оперативной памяти может достигать **1M**, хотя команды процессора оперируют 16-битными адресами;

2) упрощается использование отдельных областей памяти для кода программы, ее данных и стека;

3) при каждом выполнении программы ее код или данные могут размещаться в различных областях памяти (сегментах); работоспособность программы обеспечивается за счет того, что в командах используются не полные адреса, а только смещения относительно базового адреса сегмента.

Однако этот механизм имеет и свои недостатки:

1) в любой момент времени через сегментный регистр адресуется не более **64K** памяти, поскольку **64K** - это максимальный объем памяти, адресуемый 16-битным смещением; поэтому при работе с большими объектами в памяти приходится вручную обеспечивать правильную установку сегментных регистров;

2) в отличие от регистров общего назначения, сегментные регистры не могут использоваться в качестве источников или приемников значений

операндов в арифметических и логических командах; фактически, единственная операция, которую можно выполнять с сегментными регистрами, - это копирование значений между сегментными регистрами и регистрами общего назначения или памятью;

3) поскольку сегменты могут перекрываться, то одна и та же физическая ячейка памяти может адресоваться множеством возможных сочетаний значений сегмент/смещение.

Вообще, сегментированная модель памяти достаточно сложна, а принципы ее работы не отвечают интуитивным представлениям человека о работе памяти. Из всего этого следует, что процессор Intel 8086 весьма ограничен в использовании памяти и лучше подходит для работы с памятью в блоках, объем которых не превышает **64K**.

Хотя процессор Intel 8086 может обращаться к слову по любому адресу, при нечетном адресе требуется два обращения к памяти вместо одного: для младшего и для старшего байтов.

1.2. Программирование на Ассемблере для процессора Intel 8086

1.2.1. Понятие языка Ассемблера

Язык Ассемблера является ориентированной на человека формой команд процессора (машинного языка). Машинный язык и язык Ассемблера функционально эквивалентны, но на Ассемблере намного проще программировать, поскольку он позволяет использовать мнемонические имена команд машинного языка. При этом компилятор Ассемблера последовательно транслирует команды из мнемонического вида в их машинный эквивалент. Таким образом, язык Ассемблера в основном представляет собой прямой аналог машинного языка, но реализованный в том виде, с которым человек может работать более эффективно.

Полезным качеством Ассемблера является то, что он позволяет управлять действиями процессора по операциям и с максимальной эффективностью. К числу его недостатков можно отнести тот факт, что при каждой операции выполняется совсем немного функций, что отражает ограниченные возможности того, на что в действительности способен процессор. Например, процесс сложения двух длинных целых чисел и сохранения результата в третьем целом значении занимает на языке С только одну строку:

```
i = j + k;
```

а на Ассемблере процессора 8086 это потребует шести строк:

```
mov    ax, j
mov    dx, j+2
```

```
add     ax, k
addc    dx, k+2
mov     i, ax
mov     i+2, dx
```

Конечно, объем скомпилированного кода на языке С будет не меньше (вероятнее всего, больше), чем на языке Ассемблера, но легче написать одну строку на С, чем шесть на Ассемблере.

Причина использования Ассемблера состоит в том, что он напрямую позволяет делать все то, на что способен процессор. С другой стороны, хорошо написанная на Ассемблере программа позволит получить код наименьшего объема с наименьшим временем выполнения. Качество выполняемого кода, получаемого в других языках, страдает от того, что приходится выполнять трансляцию с этого языка на машинный язык, а код на языке Ассемблера отображается в машинный язык непосредственно, без малейшей потери эффективности. На языке Ассемблера вы указываете компьютеру, что нужно делать, и он делает именно это не больше и не меньше.

1.2.2. Структура программы

Исходный текст программы (физически содержащийся в файле в расширением **ASM**) состоит из операторов Ассемблера, каждый из которых занимает отдельную строку этого текста. Различают два типа операторов: команды и директивы. Первые при компиляции преобразуются в команды процессора, которые исполняются после загрузки в память загрузочного модуля программы (физически содержащегося в файле с расширением **COM** или **EXE**). Операторы второго типа управляют процессом компиляции преобразования текста исходной программы в код объектного модуля (физически содержащегося в файле с расширением **OBJ**). Ассемблер транслирует операторы один за другим, генерируя последовательность из команд процессора и байтов данных.

Формат операторов Ассемблера

Общий формат оператора Ассемблера имеет следующий вид:

```
[Метка[:]]
    Мнемоника [Операнд1 [{,Операнд2}]] [;Комментарий]
```

Здесь элементы, указанные в квадратных скобках, могут отсутствовать, а элементы в фигурных скобках могут повторяться один или более раз. Пробелы вводятся произвольно, но как минимум один пробел должен следовать после мнемоники.

Метка - это идентификатор, связанный с адресом первого байта того оператора, в котором она появляется. Мнемоника - это мнемоническое обозначение соответствующей команды процессора или директивы Ассемблера. Комментарий - это любая последовательность символов, начиная с символа ";" до конца строки, которая поясняет соответствующий оператор.

Метки используются как операнды в операторах программы для ссылки на адреса команд (например, при условных и безусловных переходах) и данных (например, переменных, массивов, структур). Имена меток могут состоять из следующих символов: "A" - "Z", "a" - "z", "_", "@", "$", "?", "0" - "9". Символы "0" - "9" не могут использоваться в качестве первых символов имени метки. Символы "$" и "?" имеют специальное значение, поэтому их не следует использовать в именах пользовательских меток. Имена меток не должны совпадать с именами регистров, мнемониками команд процессора, а также с ключевыми словами Ассемблера (встроенными переменными, операциями, директивами).

Каждая метка должна определяться только один раз, то есть имена меток должны быть уникальными (исключением являются локальные метки, см. далее). Как операнды метки могут использоваться любое число раз.

Метка может занимать всю строку. В этом случае значением метки является адрес команды или директивы, которая должна следовать в следующей строке программы. При определении метки следует завершать ее двоеточием в случае, если после метки следует команда процессора. Если же метка определяется для директивы (при описании данных, сегментов, подпрограмм), то двоеточие не ставится.

Основным полем в строке программы на Ассемблере является поле мнемоники. Мнемоники команды компилируются непосредственно в те команды процессора Intel 8086, которым они соответствуют. В отличие от мнемоник команд, директивы не компилируются в исполняемый код, они лишь управляют различными аспектами работы компилятора - от типа генерируемого кода (для процессоров 8086, 80286, 80386 и т. д.) до определения сегментов и формата создаваемых файлов листингов, и, таким образом, обеспечивают высокоуровневые средства программирования на Ассемблере.

Операнды оператора Ассемблера описываются выражениями. Выражения конструируются на основе операций над целочисленными и символьными константами, метками переменных и именами регистров с использованием знаков и имен операций.

Операции Ассемблера

При конструировании выражений в Турбо Ассемблере могут использоваться операции, указанные в табл. 1.3 (перечислены в порядке убывания приоритета).

Таблица 1.3

Операции Турбо Ассемблера

Приоритет	Обозначение	Описание
1	`()`	группировка
	`[]`	индексация
	`LENGTH`	возврат числа элементов переменной
	`MASK`	получение битовой маски
	`SIZE`	возврат числа байт переменной
	`WIDTH`	возврат длины в битах
2	`.`	обращение к полю структуры
3	`:`	переопределение сегмента
4	`OFFSET`	возврат смещения
	`PTR`	приведение типов
	`SEG`	возврат номера сегмента
	`THIS`	создание операнда
	`TYPE`	возврат размера типа в байтах
5	`HIGH`	возврат старших 8 бит
	`LOW`	возврат младших 8 бит
6	`+`	унарный плюс
	`-`	унарный минус
7	`*`	умножение целых чисел
	`/`	деление целых чисел
	`MOD`	остаток от целочисленного деления
	`SHL`	сдвиг влево
	`SHR`	сдвиг вправо
8	`+`	сложение целых чисел
	`-`	вычитание целых чисел
9	`EQ`	отношение "равно"
	`GE`	отношение "больше или равно"
	`GT`	отношение "больше"
	`LE`	отношение "меньше или равно"
	`LT`	отношение "меньше"
	`NE`	отношение "не равно"
10	`NOT`	побитовое логическое отрицание

Окончание табл. 1.3

Приоритет	Обозначение	Описание
11	**AND**	побитовое логическое умножение
12	**OR**	побитовое логическое сложение
	XOR	побитовое исключающее ИЛИ
13	**SHORT**	установка типа метки **SHORT**
	.TYPE	возврат контекста выражения

Назначение имен выражениям

Если выражение появляется в программе несколько раз, удобно присвоить ему имя, которым затем пользоваться при обращении. Это позволяет не только заменить коротким именем длинное выражение, но и воспользоваться содержательным именем, которое проще запомнить. Назначение имен выражениям выполняется с помощью директивы **EQU**, которая имеет следующий формат:

```
Имя_выражения EQU Выражение
```

Здесь именем выражения может быть любой допустимый идентификатор, а выражение может иметь формат любого допустимого операнда, быть любым выражением, вычисление которого дает целочисленную или символьную константу (тогда имя выражения будет именем константы), или быть любой допустимой мнемоникой. Имя, связанное с выражением, не должно быть определено ранее и не может быть позднее переопределено. Ниже приведен ряд примеров использования директивы **EQU**.

```
ALPHA   EQU 256              ; Связывает имя ALPHA с
                             ;    константой 256
DATA    EQU Height+12        ; Связывает имя DATA с
                             ;    непосредственным
                             ;    операндом Height+12
FIELD   EQU [bx].Name[si]    ; Связывает имя FIELD с
                             ;    операндом [bx].Name[si]
                             ;    (обращение к структуре)
BETA    EQU APHA-30          ; Связывает имя BETA с
                             ;    константой 256-30=226
ADDR    EQU Var+BETA         ; Связывает имя ADDR с
                             ;    непосредственным
                             ;    операндом Var+226
```

Если выражение в директиве **EQU** содержит идентификатор (имя переменной, метки или выражения), то либо он должен представлять собой все выражение, либо он должен определяться в программе до оператора **EQU**. Так, директива

```
AB      EQU Data1
```

допустима независимо от того, где в программе определяется идентификатор **Data1**, а директива

```
AB      EQU Data1+2
```

вызовет ошибку времени компиляции, если идентификатор **Data1** не определен ранее.

Еще одна важная причина использования имен вместо выражений заключается в упрощении сопровождения программы. Предположим, что в программе используется некоторая константа, которая может при модификации программы измениться. Тогда применение директивы **EQU** позволяет, связав с этой константой имя, изменить в этом случае соответствующее определение константы один раз вместо поиска и модификации каждой строки программы, в которой используется данная константа.

Полностью аналогичной директиве **EQU** является директива **=**, однако ее преимущество заключается в том, что определенные с ее помощью имена можно позднее переопределять, что особенно полезно в макрокомандах.

Директивы определения данных

Директивы резервирования и инициализации памяти под переменные имеют следующий формат:

```
Имя_переменной Директива Операнд [{, Операнд}]
```

или

```
Имя_массива Директива Целое DUP (Операнд [{, Операнд}])
```

Здесь имена переменной или массива - любые допустимые идентификаторы, операнды - выражения, вычисление которых дает целочисленную или символьную константу или оператор **DUP**, а директива определяет размер в байтах соответствующей переменной или каждого элемента соответствующего массива и может быть одним из следующих ключевых слов:

DB - для выделения байтов,

DW - для выделения слов (2 байта),

DD - для выделения двойных слов (4 байта),

DQ - для выделения четверных слов (8 байт),

DT - для выделения 10 байтовых полей.

Для выделения неинициализированных полей памяти в качестве операндов при определении переменных и в качестве операндов оператора **DUP** при определении массивов следует использовать символ "?". Ниже приведен ряд примеров выделения памяти.

```
i       DW 0                    ; Определяет слово i и
                                ;    инициализирует его 0
String DB 'Hallo'               ; Определяет байтовое поле
                                ;    String, содержащее коды
```

```
                                    ;    символов 'Hallo'
Array  DD 10 DUP (?)                ; Определяет массив Array из
                                    ;    10 неинициализированных
                                    ;    двойных слов
```

А эта директива

```
Table  DB 100 DUP (0, 2 DUP (1, 2), 0, 3)
```

резервирует память для 100 элементов, каждый из которых содержит 7 байт и инициализирован так (в шестнадцатеричном коде):

```
00 01 02 01 02 00 03
```

Все элементы, распределяемые одной директивой определения данных, должны быть одного и того же типа. Однако во многих задачах (например экономических) желательно иметь возможность определять переменные структурного типа, которые имеют несколько полей различного типа. Для определения структурного типа используется директива **STRUC**. Она имеет следующий формат:

```
Имя_типа STRUC
   {Директивы определения переменных-полей структуры}
Имя_типа ENDS
```

Определения структурных типов не могут быть вложенными. Следующий пример иллюстрирует применение директивы **STRUC** для определения типа записи о работнике предприятия:

```
TPerson STRUC
   ID         DD ?            ; табельный номер
   FirstName  DB 30 DUP (?)   ; имя
   SecondName DB 30 DUP (?)   ; отчество
   LastName   DB 30 DUP (?)   ; фамилия
TPerson ENDS
```

Определение структурированного типа не выделяет память и не инициализирует ее значениями - оно просто определяет тип. Для выделения памяти под переменную структурного типа и ее инициализации применяются директивы следующего формата:

```
Имя_переменной Имя_типа <[Операнд [{, Операнд}]]>
```

Ниже приводится пример определения переменных структурированного типа, определенного в предыдущем примере:

```
John      TPerson <, 'John'>
Annie     TPerson <>
Personnel TPerson 8 DUP (<>) ; массив из 8 структур
```

Имена полей структуры используются для доступа к соответствующим полям с помощью операции . доступа к полю структуры. Так, чтобы полю **ID** переменной **John** присвоить значение 1, нужно выполнить такие команды:

```
mov WORD PTR John.ID, 1
mov WORD PTR John.ID+2, 0
```

Операция . может использоваться в сочетании с различными режимами адресации, например следующий код делает то же, что и приведенный выше:

```
mov bx, OFFSET John
mov WORD PTR [bx].ID, 1
mov WORD PTR [bx].ID+2, 0
```

Директивы определения сегментов

Одна из задач компилятора заключается в назначении смещений используемым символическим именам (переменных, меток). Чтобы назначать смещения компилятор должен знать точную структуру каждого сегмента. В больших программах для данных, кода и стека используют различные сегменты, которые в общем случае в исходном тексте программы оформляются с помощью директивы **SEGMENT**, имеющей следующий формат:

```
Имя_сегмента SEGMENT [Тип_объединения]
   [{Операторы Ассемблера}]
Имя_сегмента ENDS
```

Типы объединения будут рассмотрены позднее.

Кроме этого компилятору необходимо знать точное соответствие между сегментами и сегментными регистрами, что позволяет проконтролировать определенные виды ошибок, например, определена ли используемая переменная в сегменте данных или нет. Назначение сегментов сегментным регистрам выполняется в помощью директивы **ASSUME**, которая имеет следующий формат:

```
ASSUME Назначение [{, Назначение}]
```

где каждое назначение имеет такой формат:

```
Имя_сегментного_регистра:Имя_сегмента
```

Так, следующая директива назначает регистру **CS** сегмент кода с именем **CodeSeg1**, а регистру **DS** - сегмент данных с именем **DataSeg1**:

```
ASSUME CS:CodeSeg1, DS:DataSeg1
```

Один и тот же сегмент можно назначать разным сегментным регистрам. Директивы **ASSUME** можно также позднее использовать для переназначения сегментных регистров.

Здесь важно понять, что директива **ASSUME** не загружает номера сегментов в соответствующие сегментные регистры. Для всех сегментных регистров, кроме **CS**, который загружается командой межсегментного перехода при передаче управления программе, загрузка должна выполняться явно командами пересылки:

```
DataSeg1 SEGMENT
   i     DB 0
   ...
```

```
DataSeg1 ENDS
CodeSeg1 SEGMENT
         ASSUME CS:CodeSeg1, DS:DataSeg1
Begin:
   mov    ax, DataSeg1
   mov    ds, ax
   ...
CodeSeg1 ENDS
```

Имя **DataSeg1** является именем сегмента, а не переменной, поэтому при трансляции первой команды **MOV** операнд-источник должен быть непосредственным значением - номером сегмента данных. Но поскольку размещение сегментов в памяти и их адреса на этапе компиляции не известны, то соответствующий операнд команды будет сформирован компоновщиком (см. далее).

Когда компилятор встречает в операнде команды имя, он предполагает для нее сегментный регистр по умолчанию в соответствии с режимом адресации. Если регистром по умолчанию принимается, например, **DS**, а имя не находится в сегменте, назначенном регистру **DS**, то компилятор просматривает сегменты, назначенные регистрам **CS**, **ES** и **SS**. Если имя найдено, то компилятор вводит перед командой, содержащей его, префикс замены сегмента, который заставляет команду использовать вместо **DS** соответствующий сегментный регистр, в противном случае фиксируется ошибка.

Сегментные регистры по умолчанию разрешается заменять явно, помещая их перед именем с последующим двоеточием, например

```
add    ax, ES:Delta
```

Вместо имени сегментного регистра допускается указывать имя сегмента, но при условии, что эти имена связаны директивой **ASSUME**, например предыдущий пример эквивалентен следующему:

```
         ASSUME ES:ExtraSeg1
...
add    ax, ExtraSeg1:Delta
```

Замена сегментного регистра по умолчанию должна выполняться явно всегда, когда используемое имя не может быть найдено ни в одном из сегментов, назначенных сегментным регистрам, или соответствующие назначения не сделаны.

*Окончание программы. Директива **END***

Конец программы на Ассемблере указывает директива **END**, имеющая следующий формат:

```
END Метка
```

Здесь метка ссылается на адрес команды, с которой начинается выполнение программы. Этот адрес называется точкой входа программы, именно по этому адресу загрузчик операционной системы передаст управление после того, как исполняемый модуль будет загружен и размещен в памяти.

1.2.3. Процедура компиляции, компоновки и отладки программ

В общем случае для преобразования исходного модуля, содержащего текст программы на языке программирования, в исполняемый модуль, который может быть запущен на выполнение, необходимо два этапа: компиляция и компоновка. В процессе компиляции выполняется лексический и синтаксический анализ исходного текста модуля, по результатам которого генерируется машинный код. Компоновка выполняется для того, чтобы скомпоновать модули, из которых состоит программная система, в единый загрузочный модуль, разрешив при этом часть неразрешенных на этапе компиляции межмодульных ссылок (обращения к переменным, передача управления).

Компилятор ***TASM***

Компилятор преобразует исходный модуль на языке программирования в объектный модуль. Компилятор Турбо Ассемблера запускается командой **TASM**, она имеет следующий формат:

```
TASM [параметры]
   ASM-файл [, OBJ-файл] [, LST-файл] [, XRF-файл]
```

Здесь **ASM-файл** указывает имя файла с расширением **ASM**, содержащего исходный модуль; **OBJ-файл** указывает имя файла с расширением **OBJ**, содержащего объектный модуль; **LST-файл** указывает имя файла с расширением **LST**, содержащего листинг; и **XRF-файл** указывает имя файла с расширением **XRF**, содержащего перекрестные ссылки. **ASM-файл** является входным, остальные генерируются компилятором в процессе обработки исходного текста программы.

Из параметров командной строки наиболее важным является параметр **/zi**, который указывает компилятору включить в объектный модуль отладочную информацию, которую можно использовать при отладке программы.

Компоновщик ***TLINK***

Компоновщик позволяет связать объектные модули в исполняемый модуль. Компоновщик запускается командой **TLINK**, она имеет следующий формат:

```
TLINK [параметры]
   OBJ-файл [{, OBJ-файл}] [, EXE-файл] [, MAP-файл]
```

Здесь **OBJ-файл** указывает имя файла с расширением **OBJ**, содержащего объектный модуль, полученный в результате компиляции; **EXE-файл** указывает имя файла с расширением **EXE**, содержащего исполняемый модуль; **MAP-файл** указывает имя файла с расширением **MAP**, содержащего карту распределения сегментов в памяти. **OBJ-файлы** являются входными, остальные генерируются компоновщиком в процессе связывания объектных модулей в исполняемый.

Из параметров командной строки наиболее важным является параметр **/v**, который указывает компоновщику включить в исполняемый модуль отладочную информацию, которую можно использовать при отладке программы.

*Отладчик **TD***

Отладчик позволяет исполнять программу в контролируемой программистом среде, анализировать изменения переменных программы, регистров и флагов процессора, выполнять трассировку и прерывание выполнения программы с целью тестирования и отладки. Отладчик запускается командой **TD**, она имеет следующий формат:

```
TD [параметры] [EXE-файл [параметры EXE-файла]]
```

Здесь **EXE-файл** указывает файл с расширением **EXE**, содержащий исполняемый модуль. Отладчик имеет полноэкранный интерфейс пользователя, так что исполняемый модуль можно не указывать в командной строке, а загрузить позднее с помощью соответствующей команды меню.

1.3. Пример разработки программы на языке Ассемблера

Задача. Разработать программу вычисления числа сочетаний из **m** по **n**, используя формулу:

$$C(m, n) = m \cdot \ldots \cdot [m - (n - 1)]/n!,$$

где **m**, **n** - натуральные числа либо нули, $n \le m$.

Определим объем памяти для хранения исходных данных (**m** и **n**) и результата вычисления (**C(m, n)**). Тем самым мы заодно выясним тот набор значений исходных данных, при которых программа будет работать корректно.

Чтобы иметь возможность провести деление числителя ($m \cdot \ldots \cdot [m - (n - 1)]$) на знаменатель (**n!**) в формуле для **C(m, n)** непосредственно командой процессора **DIV**, числитель, рассматриваемый как беззнаковое целое, должен помещаться по крайней мере в двойное слово, т. е.

$$m \cdot \ldots \cdot [m - (n - 1)] \le 2^{32} - 1 = 4\ 294\ 967\ 295,$$

а знаменатель, также рассматриваемый как беззнаковое целое, - по крайней мере в слово, т. е.

$$n! \le 2^{16} - 1 = 65\,535,$$

откуда получаем, что $n \le 8$ ($8! = 40\,320$, $9! = 362\,880$).

Определив безусловное (не зависящее от **m**) ограничение для **n**, определим безусловное (не зависящее от **n**) ограничение для **m**. Выражение для числителя формулы **C(m, n)** равно произведению последних **n** чисел натурального ряда, не превышающих **m**. Пусть **m** фиксировано, тогда числитель будет тем больше, чем, очевидно, больше **n**. А поскольку $n \le 8$, то для определения максимального допустимого **m**, при котором числитель формулы **C(m, n)** не превысит $2^{32} - 1$, положим $n = 8$. При фиксированном же **n** числитель будет тем больше, чем, очевидно, больше **m**.

Основываясь на этих рассуждениях, получим, что $m \le 19$. Действительно, при $m = 19$ имеем

$$19 \cdot 18 \cdot 17 \cdot 16 \cdot 15 \cdot 14 \cdot 13 \cdot 12 = 3\,047\,466\,240 \le 4\,294\,967\,295,$$

а при $m = 20$ уже

$$20 \cdot 19 \cdot 18 \cdot 17 \cdot 16 \cdot 15 \cdot 14 \cdot 13 = 5\,079\,110\,400 > 4\,294\,967\,295.$$

Ясно, что при $m \le 19$, $n \le 8$ числитель формулы **C(m, n)** никогда не превысит числа $2^{32} - 1$ ни при каком сочетании значений **m** и **n**.

Если делимое и делитель - целые положительные числа, и делимое не превышает $2^{32} - 1$, то частное, очевидно, также не превысит $2^{32} - 1$. Но если частное превысит $2^{16} - 1$, то при выполнении команды **DIV** процессор сгенерирует прерывание **int0** ("деление на нуль"). Чтобы этого не произошло, необходимо дополнительно проверить ограничение для **m**.

Итак,

$$m \cdot \ldots \cdot [m - (n - 1)]/n! \le 2^{16} - 1 = 65\,535.$$

Очевидно, что при фиксированном **n** правая часть неравенства тем больше, чем больше **m**. Определим ограничение на **m**, для которого выполняется это неравенство, при $n = 8$. Получим, что $m \le 18$. Действительно, при $m = 18$ имеем

$$18 \cdot 17 \cdot 16 \cdot 15 \cdot 14 \cdot 13 \cdot 12 \cdot 11/8! = 43\,758 \le 65\,535,$$

а при $m = 19$ уже

$$19 \cdot 18 \cdot 17 \cdot 16 \cdot 15 \cdot 14 \cdot 13 \cdot 12/8! = 75\,582 > 65\,535.$$

А теперь покажем, что при $n < 8$ неравенство $m \le 18$ не ужесточится. Действительно, пусть $m = 18$ (правая часть неравенства максимальна), а $n < 8$. Тогда

$$18 \cdot \ldots \cdot [18 - (n - 1)]/n! \le 18 \cdot 17 \cdot 16 \cdot 15 \cdot 14 \cdot 13 \cdot 12 \cdot 11/8!,$$

а уж тем более

$$18 \cdot \ldots \cdot [18 - (n - 1)]/n! \le 65\,535.$$

При **n > 8** будем использовать свойство сочетаний **C(m, n) = C(m, (m-n))** для замены значения **n > 8** значением, не превышающим 8. Здесь следует обеспечить, чтобы **m - n ≤ 8** и **n ≤ m**. Из первого неравенства следует, что **m - 8 ≤ n** и, следовательно, **m - 8 ≤ n ≤ m**. Если **m = 18**, то **10 ≤ n ≤ 18**, а при **n = 9** получим **m - n = 9 > 8**, что нарушает первое неравенство. Для его соблюдения необходимо, чтобы **m ≤ 17**, тогда при **m = 17** имеем **9 ≤ n ≤ 17**.

Таким образом, окончательно получим, что **m ≤ 17**, **n ≤ m**. Если **m** и **n** удовлетворяют полученным ограничениям, то ни при каком сочетании значений **m** и **n** не произойдет ни переполнения при вычислении числителя формулы **C(m, n)**, ни прерывания **int0** при вычислении самого **C(m, n)**. Для представления **m** и **n** в памяти достаточно выделить по байту, а для **C(m, n)** - слово.

При итеративном вычислении числителя формулы **C(m, n)** возникает необходимость умножить 32-разрядное число (текущее значение **C(m, n)**) на 16-разрядное (текущий множитель **m - i**, где **i = (0,..., n - 1)**), причем результат (в итоге - числитель формулы **C(m, n)**) не превысит $2^{32} - 1$, т. е. будет также 32-разрядным числом **X**, которое можно представить так

$$X = hi(X)\cdot 2^{16} + lo(X),$$

где **hi(X)** - старшее, а **lo(X)** - младшее слово 32-разрядного числа **X**. При этом если **X = A·B**, где **A** 32-разрядное, а **B** - 16-разрядное число, то

$$X = (hi(A)\cdot 2^{16} + lo(A))\cdot B = hi(A)\cdot B\cdot 2^{16} + lo(A)\cdot B,$$

причем

$$lo(X) = lo(lo(A)\cdot B)$$
$$hi(X) = lo(lo(hi(A)\cdot B) + hi(lo(A)\cdot B)).$$

Ниже приведен исходный код программы, решающей поставленную задачу с учетом всех тонкостей, которые обсуждались выше.

```
const
   m = 9;
   n = 5;

var
   Cmn: dword;
   Den: word;
   i:   byte;

begin
   if n > 8 then n := m - n;
   Den := 1;
   for i := n downto 2 do Den := Den*i;
```

```
   Cmn := 1;
   for i := m - n + 1 to m do Cmn := Cmn*i;
   Cmn := Cmn/Den
end.

;-------------------------------------------------------------
_Data_  SEGMENT                 ;
   m    EQU 9                   ; мощность множества
   n    EQU 5                   ; размер сочетания
   Cmn  DW  ?                   ; результат Cmn
   Den  DW  ?                   ; знаменатель
_Data_  ENDS                    ;
                                ;
_Code_  SEGMENT                 ;
        ASSUME DS:_Data_        ;
        ASSUME CS:_Code_        ;
Begin:                          ;
   mov    ax, _Data_            ; загрузка регистра DS
   mov    ds, ax                ;
   mov    bx, n                 ; приведение n > 8 к n <= 8
   mov    cx, bx                ;
   cmp    bx, 8                 ;
   jbe    nOK                   ;
   neg    bx                    ;
   add    bx, m                 ;
nOK:                            ; вычисление n!
   mov    ax, 1                 ;
   xor    si, si                ;
   cmp    cx, si                ;
   jz     DenOK                 ;
DenC:                           ;
   mul    cx                    ;
   loop   DenC                  ;
DenOK:                          ;
   mov    Den, ax               ;
   mov    ax, 1                 ; вычисление числителя
   mov    cx, n                 ;
   cmp    cx, si                ;
   jz     NumOK                 ;
   neg    bx                    ;
   add    bx, m                 ;
   inc    bx                    ;
NumC:                           ;
   mul    bx                    ; умножение 32-разрядного
   xchg   ax, si                ;    числа на 16-разрядное
   mov    di, dx                ;
   mul    bx                    ;
   add    ax, di                ;
```

```
   xchg   ax, si                ;
   inc    bx                    ;
   loop   NumC                  ;
NumOK:                          ;
   mov    dx, si                ;
   div    Den                   ; вычисление Cmn
   mov    Cmn, ax               ;
   mov    ax, 4C00H             ; завершение программы
   int    21H                   ;
_Code_  ENDS                    ;
                                ;
        END Begin               ;
```

1.4. Варианты индивидуальных заданий

1) Вычислите **n**-й член **F(n)** последовательности Фибоначчи. В этой последовательности первые два члена равны 1, а каждый последующий равен сумме двух предыдущих.

2) Задано число. Определить, является ли оно членом последовательности Фибоначчи. Если да, то вернуть его порядковый номер, в противном случае - 0.

3) Вычислить наибольший общий делитель двух чисел без использования команды деления.

4) Задано слово и границы битового поля. Выделить из слова битовое поле и представить его как слово.

5) Даны положительное число и переменная-флаг. Дополнить число контрольным битом по четности, если флаг сброшен, и по нечетности, если флаг установлен. Контрольный бит должен быть самым младшим.

6) Вычислить произведение двух 32-разрядных чисел без знака в предположении, что результат получится также 32-разрядным числом. В случае переполнения установить переменную-флаг, в противном случае сбросить его.

7) То же, что в п. 6, но для чисел со знаком.

8) Вычислить максимальное число, факториал которого по длине не превысит слова.

9) Задано число. Вычислить максимальный показатель, с которым степень этого числа по длине не превысит слова.

10) Задано восемь положительных чисел длиной в байт каждое. Распределить значащие биты последнего числа по старшим битам первых семи.

1.5. Контрольные вопросы

1) Почему операции умножения и деления реализованы отдельно для чисел со знаком и без знака, а операции сложения и вычитания - нет?

2) Что произойдет, если длина результата деления 32-разрядного числа превысит 16 бит? Почему?

3) Что произойдет при выполнении команды **`add ax, 1`**, если **`ax`** содержит **`0FFFFH`**? Возникнет ли при этом прерывание? Почему?

4) Почему рекомендуют использовать команду **`inc ax`** вместо **`add ax, 1`** и команду **`dec ax`** вместо **`sub ax, 1`**, а также команду **`xor ax, ax`** вместо **`mov ax, 0`**?

5) Почему рекомендуют данные, с которыми работают программы, выравнивать по границе слова?

6) Какое ограничение и почему накладывается на значение целевого адреса в командах условного перехода? Как его преодолеть?

7) Какими командами и каким образом следует пользоваться при организации сложения и вычитания многоразрядных чисел?

8) Какая команда предназначена для безусловной передачи управления по целевому адресу? Опишите алгоритм ее работы. В чем ее отличие от команд условного перехода?

9) Запишите на языке Ассемблера шаблоны для реализации условных операторов **`if`** и **`case`** языка Pascal.

10) Запишите на языке Ассемблера шаблоны для реализации операторов цикла **`while`** и **`repeat`** языка Pascal.

11) В чем принципиальное отличие команды от директивы? Поясните назначение директивы **`ASSUME`**.

12) Куда помещается очередная выполняемая команда после считывания ее из памяти? Опишите механизм функционирования соответствующей структуры процессора.

13) В чем назначение флага **`AF`**? На выполнение каких команд и как он оказывает воздействие?

14) Укажите принципиальную особенность размещения слов в памяти в архитектуре Intel 8086. Какое влияние на процедуру считывания слова из памяти оказывает базовый адрес, по которому размещено слово в памяти?

15) Опишите механизм формирования полного физического адреса памяти в архитектуре Intel 8086. Какой объем памяти физически адресуется в этой архитектуре? Почему?

2. РЕЖИМЫ АДРЕСАЦИИ. ОБРАБОТКА МАССИВОВ. ОБРАБОТКА СТРОК

2.1. Режимы адресации процессора Intel 8086

Режимом адресации называется способ определения адреса операнда. Режимы адресации разделяются на два класса - режимы адресации данных и режимы адресации переходов. Рассмотрим режимы адресации данных.

1. Непосредственный. Значение операнда длиной 8 или 16 бит является непосредственной частью команды.

```
Пример:   mov    ax, 10
```

2. Прямой. Смещение адреса операнда указано прямо в команде.

```
Пример:   mov    i, ax
```

3. Регистровый. Операнд содержится в указанном командой регистре; 16-битный операнд может находиться в регистрах **AX**, **BX**, **CX**, **DX**, **SI**, **DI**, **SP**, **BP**, а 8-битный - в регистрах **AL**, **AH**, **BL**, **BH**, **CL**, **CH**, **DL**, **DH**.

```
Пример:   inc    cx
```

4. Регистровый косвенный. Смещение адреса операнда находится в базовом регистре **BX** или **BP** или в индексном регистре **SI** или **DI**.

```
Пример:   sub    ax, [bx]
```

5. Регистровый относительный. Смещение адреса операнда определяется как сумма 8- или 16-битного относительного смещения и содержимого базового (**BX** или **BP**) или индексного (**SI** или **DI**) регистров.

```
Пример:   add    ax, [bp+2]
```

6. Базовый индексный. Смещение адреса операнда определяется как сумма содержимого базового (**BX** или **BP**) и индексного регистров (**SI** или **DI**).

```
Пример:   mov    ax, [bx+si]
```

7. Относительный базовый индексный. Смещение адреса операнда определяется как сумма 3-х слагаемых: 8- или 16-битного относительного смещения, содержимого индексного регистра (**SI** или **DI**) и содержимого базового регистра (**BX** или **BP**).

```
Пример:   mov    ax, [bx+si+2]
```

Теперь рассмотрим режимы адресации переходов.

1. Внутрисегментный прямой. Смещение адреса перехода определяется как сумма 8- или 16-битного относительного смещения и текущего содержимого регистра указателя команд **IP**. Когда относительное смещение имеет длину 8 бит, этот режим называют "коротким переходом". Данный режим допустим и в условных, и в безусловных переходах, но условные переходы - всегда короткие.

```
Пример:
          jb      IsBelow
          ...
IsBelow:
          ...
```

2. Внутрисегментный косвенный. Смещение адреса перехода есть содержимое регистра или ячейки памяти, указанные в любом режиме адресации данных, кроме непосредственного. Содержимое регистра указателя команд **IP** заменяется соответствующим содержимым регистра или ячейки памяти. Данный режим допустим только в командах безусловного перехода.

```
Пример:   jmp     [bx]
```

3. Межсегментный прямой. В команде указана пара сегмент:смещение; сегмент загружается в сегментный регистр **CS**, а смещение - в регистр указатель команд **IP**. Данный режим допустим только в командах безусловного перехода.

```
Пример:   call    FAR PTR QuickSort
```

4. Межсегментный косвенный. Содержимое регистров **CS** и **IP** заменяется содержимым двух смежных слов памяти, адрес которых указан в любом режиме адресации данных, кроме непосредственного и регистрового. Младшее слово загружается в регистр указатель команд **IP**, а старшее - в сегментный регистр **CS**. Данный режим допустим только в командах безусловного перехода.

```
Пример:   call    FAR PTR [bp+4]
```

Во всех режимах адресации с участием регистра **BP** в качестве сегментного регистра по умолчанию используется **SS**, в остальных режимах - регистр **DS**. Замену сегментного регистра нельзя производить в следующих специальных случаях:

– при участии в адресации регистра **SP** сегментным регистром всегда служит **SS**;

– в командах обработки строк в качестве сегментного регистра для операнда-приемника всегда используется **ES**;

– при формировании адреса следующей выполняемой команды всегда используется регистр **CS**.

2.2. Пример разработки программы обработки массива

Задача. Дан массив **X** чисел длины **n ≤ 255**, представляющий собой результат наблюдения за некоторой дискретной случайной величиной, принимающей целые значения из промежутка **[0; 255]**. Разработать про-

грамму вычисления оценки математического ожидания этой случайной величины по формуле: **M = (ΣX[i])/n**.

Как видно из формулы, **M** - не что иное, как среднее значение элементов исходного массива. Поскольку среднее значение не может превысить максимального (а максимальное равно 255), то **M ≤ 255**. Однако **M** - в общем случае дробное число, а процессор Intel 8086 не имеет встроенных средств представления и обработки дробных чисел, эта проблема отдана на откуп программисту.

Поэтому представим результат в виде 16-разрядного целого числа, у которого старшие 8 бит представляют целую часть (и этого достаточно, ведь **M ≤ 255**, а его целая часть - тем более), а младшие 8 бит - дробную (это позволит вычислять **M** с точностью до $1/(2^8 - 1)$, что даст по крайней мере 2 верных цифры).

```
const
   n = 10;
   x: array [1..10] of byte =
      (5, 7, 25, 150, 96, 54, 19, 24, 1, 83);
   M: word = 0,

var
   i: byte;

begin
   for i := n downto 1 do M := M + x[i];
   M := M/n
end
```

```
;-------------------------------------------------------
_Data_  SEGMENT                 ;
   n    EQU 10                  ; мощность множества
   x    DB  5, 7, 25, 150, 96,\ ; множество
            54, 19, 24, 1, 83  ;
   M    DW  0                   ; результат M
_Data_  ENDS                    ;
                                ;
_Code_  SEGMENT                 ;
        ASSUME DS:_Data_        ;
        ASSUME CS:_Code_        ;
Begin:                          ;
   mov    ax, _Data_            ; загрузка регистра DS
   mov    ds, ax                ;
   xor    ax, ax                ; обнуляем рабочие регистры
   xor    dx, dx                ;
   xor    cx, cx                ;
```

```
   mov    bl, n                ; в bl - мощность множества
   mov    cl, bl               ; в cx - счетчик циклов
Continue:                      ;
   mov    si, cx               ;
   mov    dl, x[si - 1]        ; в dx - очередной элемент
   add    ax, dx               ; накапливаем сумму в ax
   loop   Continue             ;
   div    bl                   ; вычисляем целую часть M
   xchg   ch, al               ; вычисляем дробную часть M
   div    bl                   ;
   xchg   ah, ch               ;
   mov    M, ax                ;
   mov    ax, 4C00H            ; завершение программы
   int    21H                  ;
_Code_  ENDS                   ;
                               ;
        END Begin              ;
```

2.3. Команды обработки строк

Одним из важных применений компьютеров является обработка текстов, которая связана с выполнением определенных операций над последовательностями байт, содержащих коды символов, т. е. строками. Такими операциями являются пересылка и сравнение строк, поиск, вставка, удаление или замена подстроки в строке. Хотя эти операции можно реализовать командами общего назначения, в процессоре Intel 8086 для поддержки операций обработки строк предусмотрены специальные команды, выполняющие эти операции более эффективно.

Любая команда обработки строк (КОС) непосредственно оперирует одним байтом или словом. Тип операнда может определяться самой мнемоникой (например, **CMPSB** - сравнение цепочек байт, **LODSW** - загрузка цепочки слов) либо неявно (например, **MOVS Str** - переслать строку **Str**). Независимо от используемого формата КОС ее фактические операнды всегда адресуются регистрами **DS:SI** (операнд-источник) и **ES:DI** (операнд-приемник), которые должны быть загружены до выполнения КОС.

При этом допускается явное переопределение сегмента. При использовании формата КОС, явно определяющего тип операнда, указывать операнды не обязательно, поскольку он(и) адресуются неявно регистрами **DS:SI** и **ES:DI**, однако их наличие делает программу более прозрачной и позволяет компилятору проконтролировать адресуемость операндов.

Кроме выполнения соответствующей операции, любая КОС выполняет автоматический инкремент или декремент регистров **SI** и (или) **DI**. Это повышает эффективность КОС при обработке последовательности смеж-

ных байт или слов. Выполнение автоинкремента или автодекремента управляется флагом **DF**: если **DF** сброшен, то выполняется автоинкремент, при установленном **DF** выполняется автодекремент.

Процессор Intel 8086 поддерживает КОС, указанные в табл. 2.1.

Таблица 2.1

Команды обработки строк процессора Intel 8086

Мнемоника	Описание	Логика
CMPS	сравнение строк	**[DS:SI] - [ES:DI]**
CMPSB	из байтов	**SI := SI +|- 1; DI := DI +|- 1**
CMPSW	из слов	**SI := SI +|- 2; DI := DI +|- 2**
LODS	загрузка строки	
LODSB	из байтов	**AL := [DS:SI]; SI := SI +|- 1**
LODSW	из слов	**AX := [DS:SI]; SI := SI +|- 2**
MOVS	пересылка строки	**[ES:DI] := [DS:SI]**
MOVSB	из байтов	**SI := SI +|- 1; DI := DI +|- 1**
MOVSW	из слов	**SI := SI +|- 2; DI := DI +|- 2**
SCAS	просмотр строки	
SCASB	из байтов	**AL - [ES:DI]; DI := DI +|- 1**
SCASW	из слов	**AX - [ES:DI]; DI := DI +|- 2**
STOS	запись в строку	
STOSB	из байтов	**[ES:DI] := AL; DI := DI +|- 1**
STOSW	из слов	**[ES:DI] := AX; DI := DI +|- 2**

Поскольку любая КОС обрабатывает только один байт или слово, то для обработки строк необходимо организовать их циклическое выполнение. С этой целью в системе команд процессора Intel 8086 предусмотрены специальные команды-префиксы, упрощающие построение циклов с КОС (табл. 2.2).

Таблица 2.2

Префиксы повторения процессора Intel 8086

Мнемоника	Описание	Условие останова
REP	повтор пока **CX ≠ 0**	**CX = 0**
REPE	повтор пока **CX ≠ 0** и **ZF = 1**	**CX = 0** или **ZF = 0**
REPNE	повтор пока **CX ≠ 0** и **ZF = 0**	**CX = 0** или **ZF = 1**
REPNZ	повтор пока **CX ≠ 0** и **ZF = 0**	**CX = 0** или **ZF = 1**
REPZ	повтор пока **CX ≠ 0** и **ZF = 1**	**CX = 0** или **ZF = 0**

Префикс повторения указывается перед КОС. В любом случае при использовании префиксов повторения необходимо перед выполнением цикла загрузить в регистр **CX** длину обрабатываемой цепочки в байтах или словах. При каждом повторении значение регистра **CX** уменьшается на 1.

Использование КОС с префиксами повторения вместо команд общего назначения позволяет значительно сократить размер кода и время его выполнения. В результате следующий фрагмент, выполняющий копирование строки

```
      mov    si, OFFSET SrcStr    ; установка индекса источника
      mov    di, OFFSET DstStr    ; установка индекса приемника
      mov    cx, LENGTH SrcStr    ; загрузка длины в счетчик
                                  ;
   Copy:                          ;
      mov    al, [si]             ; пересылка байта из источника
      mov    [di], al             ; в приемник
      inc    si                   ; инкремент индекса источника
      inc    di                   ; инкремент индекса приемника
      loop   Copy                 ;
```

может быть записан так:

```
      mov    si, OFFSET SrcStr    ; установка индекса источника
      mov    di, OFFSET DstStr    ; установка индекса приемника
      mov    cx, LENGTH SrcStr    ; загрузка длины в счетчик
      cld                         ; сброс DF - сканируем вперед
                                  ;
      rep movs SrcStr, DstStr     ; копирование строки
```

2.4. Пример разработки программы обработки строк

Задача. Даны две строки. Определить, содержится ли первая строка во второй. Если да, то вычислить начальную позицию первого вхождения, в противном случае вернуть нуль. Длина строки не может превышать **64K**, т. е. результат - 16-разрядное число.

```
const
   SrcStr: string = "AbcDefGhi";
   Substr: string = "Def";
   m = 9;
   n = 3;
   j: word = 0;
   i: word = 0;

begin
   while i < n and j <= m - n do
      if Substr[i] = SrcStr[j + i] then
         i := i + 1
```

```
      else
      begin
         i := 0;
         j := j + 1
      end;

   if j > m - n then j := 0
end

;-------------------------------------------------------------
_Data_  SEGMENT                  ;
   SrcStr DB  "AbcDefGhi"        ; исходная строка
   Substr DB  "Def"              ; подстрока
   m      EQU 9                  ; длина исходной строки
   n      EQU 3                  ; длина подстроки
   Index  DW  0                  ; результат
_Data_  ENDS                     ;
                                 ;
_Code_  SEGMENT                  ;
        ASSUME DS:_Data_         ;
        ASSUME ES:_Data_         ;
        ASSUME CS:_Code_         ;
Begin:                           ;
   mov    ax, _Data_             ; загрузка регистров DS и ES
   mov    ds, ax                 ;
   mov    es, ax                 ;
   mov    dx, n                  ; в dx - длина подстроки
   mov    ax, OFFSET SrcStr      ; в ax - адрес подстроки
   mov    bx, ax                 ; в bx - предел цикла
   add    bx, m                  ;
   sub    bx, dx                 ;
   cld                           ; сканируем вперед
ScanStr:                         ;
   cmp    bx, ax                 ;
   jb     End                    ; подстрока не найдена
   mov    si, ax                 ;
   mov    di, OFFSET SubStr      ;
   mov    cx, dx                 ;
   repe   cmpsb SrcStr, SubStr   ; сравнение подстроки
   je     Found                  ; подстрока найдена
   inc    ax                     ; подстрока пока не найдена
   jmp    ScanStr                ;
Found:                           ;
   sub    ax, (OFFSET SrcStr)-1  ; преобразуем адрес в индекс
   mov    Index, ax              ;
End:                             ;
   mov    ax, 4C00H              ; завершение программы
   int    21H                    ;
```

```
_Code_  ENDS                    ;
                                ;
        END Begin               ;
```

2.5. Варианты индивидуальных заданий

1) Вернуть последнее вхождение элемента массива с заданным значением или 0, если элемент не найден.

2) Выполнить поиск максимального и минимального элементов массива, подсчитать количество положительных, отрицательных и нулевых элементов массива.

3) Выполнить скалярное произведение двух векторов. Обеспечить проверку на равенство длин векторов.

4) Выполнить реверс заданного массива.

5) Выполнить сортировку заданного массива.

6) Выполнить транспонирование заданной матрицы.

7) Выполнить умножение двух заданных матриц. Обеспечить проверку допустимости выполнения операции.

8) Выполнить преобразование строки в число. Если строка не содержит символьное представление числа, установить переменную-флаг, иначе сбросить его.

9) Выполнить преобразование числа в строку.

10) Преобразовать кодировку строки из **CP-866** в **CP-1251**.

11) Вернуть начальную позицию последнего вхождения подстроки в строке или 0, если подстрока не найдена.

12) Задана строка, начальная и конечная позиции подстроки. Выделить подстроку.

13) Задана строка и массив символов. Заменить каждое вхождение символа из массива в строке на заданный символ.

14) Реализовать операцию конкатенации строк.

15) Выполнить преобразование символов строки в верхний и нижний регистр.

2.6. Контрольные вопросы

1) Какой режим адресации использован в команде **inc Lst[bx]**, где **Lst** - имя глобальной переменной?

2) В чем состоит принципиальное отличие межсегментных переходов от внутрисегментных?

3) Какие ресурсы (команды, регистры, флаги) процессора Intel 8086 предназначены для работы с последовательностями байт или слов?

4) Какова роль флага **DF** при работе со строками? Какие команды предназначены для управления им?

5) В каких режимах адресации используются регистры **SI** и **DI**?

6) В каких режимах адресации используются регистры **BX** и **BP**?

7) Какой сегментный регистр используется по умолчанию в командах обработки строк для операндов, адресуемых через регистры **SI** и **DI**?

8) Какой сегментный регистр используется по умолчанию при адресации через регистр **BP**?

9) Опишите алгоритм вычисления адреса операнда при использовании относительного базового индексного режима адресации. На каком этапе производится это вычисление, в ходе компиляции или в ходе выполнения программы?

10) Какой сегментный регистр используется по умолчанию при адресации переходов? Можно ли использовать для этих целей какой-либо другой сегментный регистр? Какой?

3. ВВЕДЕНИЕ В МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

3.1. Введение

Большинство практических программ оказываются достаточно сложными и включают в себя решение нескольких различных задач. Для эффективного решения этих задач в рамках одной программы используют средства модульного программирования, в основе которого лежит идея о представлении программы в виде совокупности сравнительно небольших фрагментов - модулей, каждый из которых ориентирован на решение одной конкретной задачи. При этом возникает необходимость определения межмодульных интерфейсов (связей по управлению и по данным), которые должны быть как можно более простыми (минимальными).

Модульное программирование предоставляет следующие преимущества:

– процессы проектирования и документирования программы могут быть упорядочены, а их результаты могут быть более прозрачными;

– разные модули можно разрабатывать, компилировать, тестировать и сопровождать независимо друг от друга при условии неизменности интерфейсов;

– одни и те же модули можно использовать многократно при разработке различных программ.

В языке Ассемблера предусмотрено три средства, предназначенных для реализации модульных программ. Первое заключается в структурировании фрагментов кода таким образом, чтобы он мог быть использован многократно (организация подпрограмм). Второе средство позволяет вводить секции кода с помощью одного оператора (макрокоманды). Третье предназначено для структурирования данных и фрагментов кода таким образом, чтобы к ним можно было легко получить доступ из нескольких различных модулей (экспорт и импорт идентификаторов).

3.2. Организация подпрограмм

3.2.1. Стек

Во многих ситуациях программе требуется временно запомнить информацию, а затем считывать ее в обратном порядке. Одним из примеров может служить запоминание и восстановление счетчиков при организации вложенных циклов. В процессоре Intel 8086 для реализации необходимых в циклах счета, проверки и перехода удобно применять регистр **CX** и команды организации циклов. Но так как последние предназначены для использования только регистра **CX**, при вложении циклов возникает проблема нехватки регистров.

Проблема временного хранения данных решается путем реализации в компьютере структуры, называемой стеком. Стек - это область памяти, взаимодействие программ с которой построено по принципу LIFO (Last In - First Out, последним пришел первым ушел), т. е. помещаемая в стек информация считывается в обратном порядке. Наиболее важная область применения стека - организация подпрограмм.

Стек рассчитан на косвенную адресацию через специальный регистр - указатель стека (**SP** - Stack Pointer). При включении элементов в стек производится автоматический декремент указателя стека, а при извлечении - инкремент, таким образом стек растет в сторону младших адресов. Помещение элемента в стек называется включением (push), а обратное действие - извлечением (pop). Адрес последнего включенного в стек элемента (он содержится в регистре указателя стека) называется вершиной стека (TOS - Top Of Stack).

В процессоре Intel 8086 реализовано 4 команды, предназначенных для непосредственного включения и извлечения информации из стека, - **POP**, **POPF**, **PUSH**, **PUSHF**. Эти команды не допускают непосредственной адреса-

ции, хотя все остальные режимы адресации разрешены. Стек оперирует только словами, так что попытка включения байта все равно приведет к включению целого слова. На регистр флагов влияет только команда **POPF**, которая извлекает содержимое вершины стека в регистр флагов.

Физический адрес стека формируется из содержимого регистров **SP** и **SS** или **BP** и **SS**, причем **SP** служит неявным указателем стека для всех операций включения и извлечения, а **SS** - сегментным регистром стека. Значение регистра **SS** является самым младшим адресом (т. е. границей) области стека и называется базой стека. Начальное значение регистра **SP** есть наибольшее смещение, которого может достигать стек. Регистр **BP** предназначен для произвольных обращений к стеку.

Стек эффективнее обычной памяти в двух отношениях. Команды **PUSH** и **POP** короче команды **MOV**, так как один из операндов косвенно адресуется через регистр **SP**, а инкремент или декремент регистра **SP** с образованием нового целевого адреса производится автоматически.

3.2.2. Понятие подпрограммы

Подпрограммой называется код, к которому можно перейти и из которого можно возвратиться так, как будто код вводится в ту точку, из которой осуществляется переход. Переход к подпрограмме называется вызовом, а соответствующий переход назад называется возвратом.

Возврат всегда производится к команде, находящейся сразу после вызова независимо от того, где находится вызов. Если к одной и той же подпрограмме делается несколько вызовов, возврат после каждого вызова осуществляется к команде, находящейся после данного вызова. Следовательно, в памяти нужно хранить только одну копию подпрограммы, даже если она вызывается несколько раз. При вложении вызовов подпрограмм каждый возврат производится в соответствующую вызывающую, а не в старшую по иерархии подпрограмму.

Подпрограммы являются основным средством разделения кода программы на модули. Хотя понятие модуля шире понятия подпрограммы, тем не менее подпрограммы, так же как и модули, можно разрабатывать, тестировать и документировать по отдельности. Кроме того, единожды написанные подпрограммы можно использовать в множестве программ, что позволяет в итоге сделать большую программу много более компактной. При разработке больших программ подпрограммы являются существенным средством для создания упорядоченного и легко сопровождаемого исходного кода.

Основной недостаток подпрограмм заключается в необходимости написания дополнительного кода для объединения их так, чтобы они могли взаимодействовать друг с другом.

При вызове подпрограммы необходимо выполнить следующие три требования:

– в отличие от других команд переходов вызов подпрограммы должен запомнить адрес следующей команды, чтобы можно было осуществить возврат в нужное место вызывающей программы;

– используемые подпрограммой регистры необходимо запомнить до изменения их содержимого, а перед самым выходом из подпрограммы восстановить эти регистры;

– подпрограмма должна иметь средства взаимодействия или разделения данных с вызвавшей ее программой и другими подпрограммами.

3.2.3. Вызовы, возвраты и определение подпрограмм

Первое из приведенных выше требований удовлетворяется при наличии специальных команд вызова и возврата: **CALL** и **RET**. Команда **CALL** не только осуществляет переход по указанному адресу, но и включает в стек адрес возврата (т. е. адрес команды, непосредственно следующей за командой **CALL**). Команда **RET** просто извлекает адрес возврата из стека. При этом предполагается, что если подпрограмма обращалась к стеку, то указатель стека возвращен на прежнее значение и действия со стеком не разрушили адреса возврата.

Если оператор вызова оказывается обращением вперед, то перед операндом следует ввести атрибутный оператор **NEAR PTR** или **FAR PTR**, чтобы указать внутрисегментный прямой вызов или межсегментный прямой вызов соответственно. Если команда **CALL** является внутрисегментным (межсегментным) вызовом, то и команда **RET** должна быть внутрисегментным (межсегментным) возвратом. Это объясняется тем, что внутрисегментный вызов включает в стек только содержимое регистра **IP**, т. е. смещение адреса возврата, а межсегментный вызов включает в стек и регистр **IP**, и регистр **CS**. Соответственно возврат должен извлекать из стека одно или два слова.

В команде **RET** допускается необязательный 16-битный непосредственный операнд. Этот операнд, если он имеется, прибавляется к содержимому регистра **SP** после извлечения из стека адреса возврата. О том, как это использовать, будет сказано далее.

Подпрограммы ограничиваются в исходном коде директивой **PROC** со следующим форматом

Имя_процедуры PROC Атрибут

в начале подпрограммы и директивой **ENP** со следующим форматом

Имя_процедуры ENDP

в конце подпрограммы. Имя подпрограммы является идентификатором, используемым для вызова, а атрибутом служит одно из слов **NEAR** или **FAR**. Атрибут необходим для определения типа команды **RET**. В случае **NEAR** команда **RET** извлекает из стека одно слово в регистр **IP** (внутрисегментный возврат), а в случае **FAR** извлекается еще одно слово в регистр **CS** (межсегментный возврат).

Подпрограмма может находиться:

– в том же сегменте кода, что и вызывающий ее оператор;

– в сегменте кода, отличном от сегмента, содержащего вызывающий ее оператор, но в том же исходном модуле, что и вызывающий оператор;

– в исходном модуле и сегменте, которые отличаются от содержащих вызывающий оператор.

В первом случае атрибутом может быть **NEAR**, но при условии, что все вызовы находятся в том же сегменте кода, что и процедура. В последних двух случаях атрибутом должен быть **FAR**. Если подпрограмме придан атрибут **FAR**, то все ее вызовы должны быть межсегментными, даже если вызов осуществляется из того же сегмента кода. В противном случае возврат извлекает из стека два слова, хотя вызов включал одно слово. Наконец, в третьем случае имя процедуры необходимо объявить в операторах **EXTRN** и **PUBLIC** (см. далее).

3.2.4. Запоминание и восстановление регистров

При программировании на языках высокого уровня подпрограмма обычно не может изменить значения переменных вызывающей программы, если вызывающая программа этого явным образом не допускает. При программировании на Ассемблере это не так: переменные вызывающей программы хранятся часто в тех же регистрах, что и локальные переменные, используемые подпрограммой. Поэтому, если подпрограмма изменяет регистр, значение которого вызывающая программа установила перед вызовом подпрограммы, но который она использует после обращения к ней, то программа будет работать неправильно.

Так как вызывающая программа и подпрограмма разделяют одни и те же регистры, при вызове подпрограммы необходимо запомнить регистры, а перед возвратом в вызывающую программу восстановить их. Для запоминания и восстановления содержимого регистров, модифицируемых

подпрограммой, практически всегда используют стек. К сожалению, для этого требуется существенное время и большой объем кода.

Можно запоминать не все регистры, а только те из них, которые модифицирует подпрограмма. Однако если подпрограмма не слишком мала и подвержена изменениям, то целесообразно автоматически запомнить все регистры - это облегчит сопровождение такой подпрограммы. Исключением являются регистры, используемые для возврата результата в вызывающую программу.

Другая возможность заключается в том, чтобы ввести правило, что вызывающая программа никогда не рассчитывает на сохранение регистров подпрограммой и все функции по их сохранению выполняет сама. То есть запоминание возможно произвести непосредственно перед вызовом подпрограммы, а восстановление сразу после возврата. Однако данный способ обычно не применяется, поскольку если эти задачи решает подпрограмма, то необходимый код записывается только один раз, а если такой код является частью вызывающей программы, то его приходится повторять при каждом вызове.

Нужно разумно относиться к сохранению регистров и обеспечить, чтобы при вызове каждой подпрограммы из-за регистров не возникало конфликтов. Наилучшим подходом является аккуратное комментирование каждой подпрограммы с обязательным указанием, какие регистры она использует.

Необходимо уделять внимание как отслеживанию сохранения регистров, так и максимально эффективному их использованию. При программировании на Ассемблере это одинаково важно. Языки высокого уровня выполняют эту работу за программиста, но они не позволяют получать такие быстрые и компактные программы, какие можно получить с помощью языка Ассемблера.

3.2.5. Взаимодействие подпрограмм

При разработке подпрограмм практически всегда требуется обеспечить возможность выполнения функции, реализуемой подпрограммой, не над конкретной переменной или не с конкретным значением, а с нужной в данный момент (вызова) переменной и с нужным в данный момент (вызова) значением. Таким образом, необходимо обеспечить передачу подпрограмме параметров, которые и будут представлять собой данные, обрабатываемые подпрограммой.

Параметр может быть передан по ссылке или по значению. В первом случае передается адрес области памяти (переменной вызывающей про-

граммы), содержащей соответствующий параметр. Этот способ позволяет подпрограмме изменять значение параметра так, что эти изменения останутся в силе после возврата из подпрограммы и будут видны вызывающей программе. Во втором случае передается непосредственное значение параметра. Этот способ исключает возможность изменения подпрограммой переменных вызывающей программы.

Технически передачу параметров можно организовать тремя способами: через глобальные переменные (в этом случае никакой передачи данных, собственно, и нет), через регистры и через стек. Последний способ самый сложный, но и самый гибкий, именно он всегда используется при трансляции программ с языков высокого уровня.

При передаче параметров через стек вызывающая программа сначала включает в стек фактические параметры (для параметров, передаваемых по ссылке, - их адреса, для параметров, передаваемых по значению, - их значения), выполняя последовательность команд **PUSH**, а затем осуществляет вызов подпрограммы. Сразу после вызова содержимое стека выглядит так, как показано на рис. 3.1.

…	
Параметр 1	
…	
Параметр n	
Регистр CS до вызова - если был межсегментный вызов	
Регистр IP до вызова	**<SP**
…	
База стека	**<SS**

Рис. 3.1. Содержимое стека после вызова подпрограммы

Для доступа к переданным параметрам в подпрограмме обычно используют регистр **BP** как базовый регистр при косвенной, относительной и индексной адресации, причем необходимо помнить, что стек растет в сторону младших адресов. Поскольку вызовы подпрограмм часто бывают вложенными в другие подпрограммы, то перед тем, как использовать регистр **BP** для доступа к параметрам, подпрограмма должна сначала сохранить регистр **BP** (чаще всего в стеке), чтобы перед возвратом в вызывающую подпрограмму была возможность восстановить его прежнее значение. Таким образом, первые команды, с которых обычно начинается любая подпрограмма, это пара команд

```
push   bp
mov    bp, sp
```

после выполнения которых содержимое стека выглядит так, как показано на рис. 3.2.

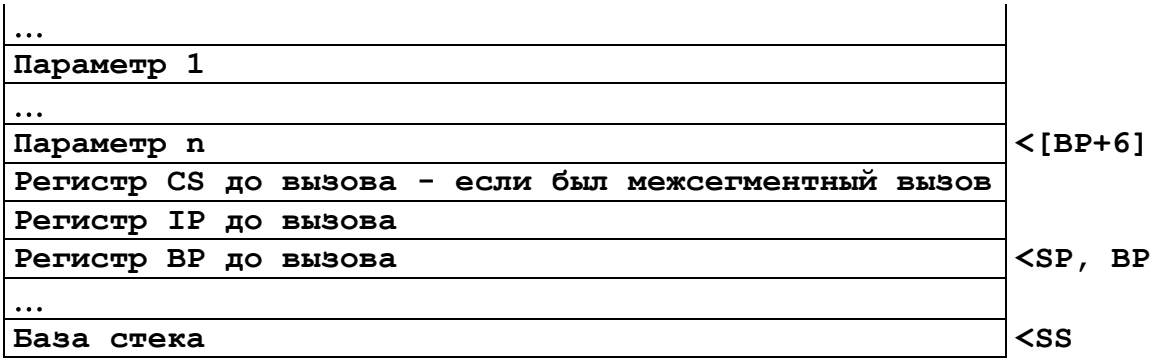


Рис. 3.2. Содержимое стека после сохранения регистра **BP**

Обращение к переданным параметрам производится через регистр **BP** с применением регистровой относительной адресации, например так (см. рис. 3.2.):

```
mov     ax, WORD PTR [bp+6]    ; занести в регистр AX
                               ; параметр n
```

Естественно, перед возвратом необходимо восстановить значение **BP** с помощью команды **POP**:

```
pop     bp
```

При возврате из подпрограммы необходимо освободить стек от переданных параметров. Обычно эту задачу выполняет подпрограмма, для чего достаточно команде **RET** указать необязательный 16-битный операнд, равный количеству байтов, переданных подпрограмме. Это значение будет прибавлено процессором к содержимому регистра **SP** сразу после того, как будет извлечен адрес возврата.

Как правило, подпрограмме необходимо возвратить вызывающей программе одно или несколько значений-результатов. Чаще всего для этого используют регистры общего назначения. С целью упрощения сопровождения программы лучше пользоваться теми или иными соглашениями, определяющими, в каких именно регистрах располагаются возвращаемые значения. В качестве таких соглашений лучше использовать те, что приняты в каком-либо из языков высокого уровня.

В случае, если подпрограмма возвращает несколько значений, может возникнуть ситуация нехватки регистров или, что хуже, ситуация, когда подпрограмма разрушает важную для вызывающей программы информацию в соответствующих регистрах. Последнее, в свою очередь, может привести к возникновению трудноуловимых ошибок. Чтобы этого избежать, следует всегда сводить к минимуму число значений, возвращаемых в

регистрах. Полезным в данном случае представляется соглашение, используемое в языках высокого уровня: в регистрах возвращается только одно значение, остальные сохраняются в ячейках памяти, на которые ссылаются переданные в качестве параметров указатели.

3.2.6. Локальные данные и локальные метки

Еще одним важным моментом при разработке подпрограмм является то, что часто подпрограмме необходимо использовать память для сохранения промежуточных результатов вычислений, реализуемых подпрограммой, причем желательно, чтобы эта память использовалась только в те моменты времени, когда выполняется данная подпрограмма, а все остальное время эта память была доступна для других подпрограмм.

Для хранения локальных переменных подпрограммы практически всегда пользуются стеком, что позволяет учесть указанные требования. Причем для этого не нужно выполнять команды **PUSH** и **POP** - достаточно обращаться к соответствующим ячейкам памяти, используя регистр **BP** (содержащий то же значение, что и **SP**) и соответствующий режим адресации, например так:

```
mov    WORD PTR [bp-2], 10    ; занести число 10 в слово,
                              ; расположенное сразу после
                              ; сохраненного регистра BP
```

Содержимое стека здесь можно представить так, как показано на рис. 3.3.

…	
Параметр 1	
…	
Параметр n	
Регистр CS до вызова - если был межсегментный вызов	
Регистр IP до вызова	
Регистр BP до вызова	<SP, BP
10	<[BP-2]
…	
База стека	<SS

Рис. 3.3. Содержимое стека при использовании локальных данных

Поскольку при таком обращении к стеку указатель стека **SP** не изменяет своего значения, то не нужно заботиться об освобождении памяти, использованной для локальных данных, перед возвратом в вызывающую программу.

При разработке множества подпрограмм часто бывает, что небольшие участки кода разных подпрограмм выполняют похожие функции. А если две части кода выполняют похожие функции, то из этого следует, что, скорее всего, они содержат одинаковые метки (при организации циклов и условных переходов). При разработке небольших программ уникальность меток обеспечивается сравнительно легко, однако в больших программах это может оказаться затруднительным.

Здесь возникает необходимость в метках такого типа, область действия которых ограничена отдельной подпрограммой, что позволило бы гарантированно избежать конфликтов по именам с другими метками. Метки такого типа называются локальными.

Имена локальных меток всегда начинаются с пары символов "@" и ограничены по области действия командами, заключенными между двумя нелокальными метками (т. е. имена которых не начинаются с пары символов "@"). При использовании локальных меток всегда следует помнить о том, что любая нелокальная метка ограничивает область действия локальной метки.

Следующий пример показывает правильное использование локальных меток во фрагменте подпрограммы вычисления факториала.

```
Fac     PROC NEAR               ; подпрограмма вычисления x!
@@POP   EQU 2                   ; количество переданных байт
@@X     EQU [bp+4]              ; параметр x
   push   bp                    ; сохраняем bp
   mov    bp, sp                ; адресация к стеку - через bp
   mov    ax, 1                 ; в dx:ax - результат
   xor    dx, dx                ;
   mov    cx, @@X               ; в cx - счетчик циклов
   cmp    cx, dx                ;
   jz     @@Done                ;
@@Continue:                     ;
   ...                          ;
   loop   @@Continue            ;
@@Done:                         ;
   mov    sp, bp                ;  восстанавливаем sp
   pop    bp                    ;  восстанавливаем bp
   ret    @@POP                 ;  возврат из подпрограммы
Fac     ENDP                    ;
```

А вот пример ошибки при использовании локальных меток:

```
SomeNonLocalLabel:              ; Первая нелокальная метка
   ...                          ; Код
@@SomeLocalLabel:               ; Локальная метка
```

```
   ...                              ; Код
AnotherNonLocalLabel:               ; Другая нелокальная метка -
                                    ;    метка @@SomeLocalLabel
                                    ;    стала недоступной
   ...                              ; Код
   jle    @@SomeLocalLabel          ; Ошибка: условный переход
                                    ;    к неизвестной метке
```

Такая ошибка возможна, например, если в теле цикла **while** используется условный оператор **if** или в аналогичных случаях. Код, подобный приведенному выше, вызовет ошибку времени компиляции.

3.2.7. Подпрограммы и стек

При разработке подпрограмм необходимо учитывать объем памяти, выделенной под стек. Если в программе используется множество подпрограмм, вызывающих одна другую и использующих много локальных данных, то возможно переполнение стека. Это особенно вероятно при разработке рекурсивных подпрограмм. В таких случаях необходимо для стека выделять достаточный объем памяти.

3.3. Пример разработки подпрограммы

Задача. Разработать программу вычисления числа сочетаний из **m** по **n**, используя формулу:

$$C(m, n) = m!/(n!\cdot(m - n)!),$$

где **m**, **n** - натуральные числа, $n \le m$.

Определим объем памяти для хранения исходных данных (**m** и **n**) и результата вычисления (**C(m, n)**). Тем самым мы заодно выясним тот набор значений исходных данных, при которых программа будет работать корректно.

Чтобы иметь возможность провести деление числителя (**m!**) на знаменатель (**n!**) в формуле для **C(m, n)** непосредственно командой процессора **DIV**, числитель, рассматриваемый как беззнаковое целое, должен помещаться по крайней мере в двойное слово, т. е.

$$m! \le 2^{32} - 1 = 4\ 294\ 967\ 295,$$

а знаменатель, также рассматриваемый как беззнаковое целое, - по крайней мере в слово, т. е.

$$n!\cdot(m-n)! \le 2^{16} - 1 = 65\ 535.$$

Если **m** фиксировано, то при **m = n** знаменатель максимален, и тогда последнее неравенство перепишется так: $n! \le 65\ 535$, откуда получаем,

что **n ≤ 8** (**8! = 40 320**, **9! = 362 880**). А поскольку мы предположили, что **m = n**, то и **m ≤ 8**. Ясно, что при **m ≤ 8** и **n ≤ m** знаменатель формулы **C(m, n)** никогда не превзойдет **8! ≤ 65 535**. Ясно также, что при **m ≤ 8** условие для числителя формулы **C(m, n)** безусловно выполняется.

Легко видеть, что при **m = 8** результат **C(m, n) ≤ 65 535** при любом **n ≤ 8**, так что при делении числителя на знаменатель формулы **C(m, n)** прерывания **int0** не возникнет.

Таким образом, окончательно получим, что **m ≤ 8**, **n ≤ m**. Если **m** и **n** удовлетворяют полученным ограничениям, то ни при каком сочетании значений **m** и **n** не произойдет ни переполнения при вычислении знаменателя формулы **C(m, n)**, ни прерывания **int0** при вычислении самого **C(m, n)**. Для представления **m** и **n** в памяти достаточно выделить по байту, а для **C(m, n)** - слово.

```
const
   m = 8;
   n = 5;

var
   Cmn: word;

function Fac(x: byte): dword
var
   i:   byte;

begin
   Result := 1;
   for i := x downto 2 do Result := Result*i
end;

begin
   Cmn := Fac(m)/(Fac(n)*Fac(m-n))
end.
```

```
;-------------------------------------------------------
_Data_  SEGMENT                 ;
   m    DW 8                    ; мощность множества
   n    DW 5                    ; размер сочетания
   Cmn  DW ?                    ; результат Cmn
_Data_  ENDS                    ;
                                ;
_Stack_ SEGMENT STACK 'STACK'   ;
        DW 3 DUP (?)            ;
_Stack_ ENDS                    ;
                                ;
```

```
_Code_  SEGMENT                 ;
        ASSUME DS:_Data_        ;
        ASSUME SS:_Stack_       ;
        ASSUME CS:_Code_        ;
Fac     PROC NEAR               ; подпрограмма вычисления x!
@@POP   EQU 2                   ; количество переданных байт
@@X     EQU WORD PTR [bp+4]     ; параметр x
   push   bp                    ; сохраняем bp
   mov    bp, sp                ; адресация к стеку - через bp
   mov    ax, 1                 ; в dx:ax - результат
   xor    dx, dx                ;
   mov    cx, @@X               ; в cx - счетчик циклов
   cmp    cx, dx                ;
   jz     @@Done                ;
@@Continue:                     ;
   mul    cx                    ; умножение 32-разрядного
   xchg   si, ax                ;    числа на 16-разрядное
   mov    di, dx                ;
   xchg   ax, dx                ;
   mul    cx                    ;
   add    ax, di                ;
   xchg   dx, ax                ;
   xchg   ax, si                ;
   loop   @@Continue            ;
@@Done:                         ;
   mov    sp, bp                ; восстанавливаем sp
   pop    bp                    ; восстанавливаем bp
   ret    @@POP                 ; возврат из подпрограммы
Fac     ENDP                    ;
                                ;
Begin:                          ;
   mov    ax, _Data_            ; загрузка регистра DS
   mov    ds, ax                ;
   mov    bx, n                 ; вычисляем n!
   push   bx                    ;
   call   Fac                   ;
   neg    bx                    ; вычисляем (m - n)!
   add    bx, m                 ;
   push   bx                    ;
   xchg   bx, ax                ;
   call   Fac                   ;
   mul    bx                    ; перемножаем результаты
   xchg   bx, ax                ;
   push   m                     ; вычисляем m!
   call   Fac                   ;
   div    bx                    ; получаем Cmn
   mov    Cmn, ax               ;
   mov    ax, 4C00H             ; завершение программы
```

```
   int    21H                    ;
_Code_  ENDS                     ;
                                 ;
        END Begin                ;
```

3.4. Макросредства Турбо Ассемблера

Достоинства подпрограмм заключаются в том, что они экономят память и время программирования благодаря многократному использованию кода и обеспечивают модульность, упрощающую отладку и сопровождение программ. Недостаток подпрограмм объясняется возникающими при их связывании накладными расходами, которые иногда превышают затраты на решение самой задачи. В этом случае использование подпрограммы не дает экономии памяти, а время выполнения программы значительно увеличивается.

Для таких ситуаций, когда очень похожие, но короткие фрагменты кода используются во многих местах программы, необходимо средство, обеспечивающее модульность подпрограмм, но позволяющее избежать накладных расходов по связыванию. Таким средством являются макрокоманды.

3.4.1. Макрокоманды

Макрокомандой называется поименованный фрагмент кода, который можно использовать в нескольких точках исходного модуля, помещая там лишь имя макрокоманды - макровызов. При этом код, соответствующий указанной макрокоманде, будет компилироваться так, как если бы он непосредственно присутствовал в точке вызова макрокоманды, и столько раз, сколько макровызовов было сделано.

Таким образом, макрокоманды не требуют накладных расходов на связывание и при этом обеспечивают модульность, однако код макрокоманды дублируется при каждом ее вызове. Говорят, что имя макрокоманды расширяется до полного кода этой макрокоманды, поэтому для описания подстановки кода макрокоманды вместо ее имени часто используется термин "макрорасширение". Следует помнить, что расширение макрокоманд выполняется во время компиляции, а не во время выполнения программы.

Макрокоманда определяется директивой **MACRO**, которая имеет следующий формат:

```
Имя_макроса MACRO [Имя_параметра [{, Имя_параметра}]]
   [{Операторы Ассемблера}]
ENDM
```

Определение макрокоманды может содержать любой допустимый оператор Ассемблера, т. е. любую команду, директивы определения данных и сегментов и прочее. Следующий пример макрокоманды реализует эффективное умножение значения в регистре **AX** на 4 с сохранением результата в **DX:AX**:

```
Multiply MACRO
   xor    dx, dx
   shl    ax, 1
   rcl    dx, 1
   shl    ax, 1
   rcl    dx, 1
ENDM
```

Чтобы обеспечить возможность применения кода макрокоманды в различных условиях следует использовать макрокоманды с параметрами. Формальные параметры макрокоманды указываются при ее описании как операнды директивы **MACRO**. При вызове макрокоманды с параметрами ее фактические параметры необходимо указывать как операнды макровызова. Перед расширением каждого макровызова происходит подстановка фактических параметров вместо формальных.

В случае, когда в тексте макрокоманды имя формального параметра смешивается с окружающим его текстом, Турбо Ассемблер не может определить, является ли соответствующая подстрока именем формального параметра или нет. Поэтому во избежание ошибок следует в таких случаях имя формального параметра ограничивать парой символов амперсанда ("&"). Если при макрорасширении Турбо Ассемблер обнаруживает подстроку, окруженную амперсандами, то сначала проверяется, является ли она именем формального параметра. Если да, то эта подстрока заменяется значением соответствующего фактического параметра, в противном случае амперсанды игнорируются.

Следующий пример макрокоманды можно использовать для создания инициализированного массива указанной длины:

```
InitArray MACRO ArrayName, ArrayLength, InitValue
   ArrayName DB ArrayLength DUP (&InitValue&)
ENDM
```

При описании макрокоманд можно использовать макровызовы других макрокоманд, а также описывать макрокоманды внутри других макрокоманд. В последнем случае вложенная макрокоманда доступна извне, как если бы она не была вложенной. В любом случае опережающие ссылки на макрокоманды не допускаются: все макрокоманды должны быть определены до их макровызова. Это обусловлено тем, что компилятор не может точно определить, сколько байт необходимо зарезервировать для макро-

расширения до того, как будет известно точное описание соответствующей макрокоманды.

В макрокомандах можно использовать директиву **EXITM**. Эта директива указывает Турбо Ассемблеру, что нужно прекратить расширение текущей макрокоманды. Если же текущая макрокоманда является вложенной по отношению к другой макрокоманде, то расширение внешней макрокоманды продолжается.

3.4.2. Локальные метки

При необходимости использования метки в макрокоманде возникает проблема дублирования меток в процессе макрорасширения. Одно из решений - использование в качестве метки имени формального параметра, который заменяется различными фактическими при макрорасширении. Однако в таком случае программист вынужден самостоятельно контролировать уникальность меток, что утомительно.

В Турбо Ассемблере для решения указанной проблемы предлагается директива **LOCAL**, которая позволяет при определении макрокоманды описать имена меток, область действия которых ограничивается данной макрокомандой. Директива **LOCAL** имеет следующий формат:

```
LOCAL Имя_метки [{, Имя_метки}]
```

и должна следовать сразу за директивой **MACRO**. В ходе макрорасширения имена локальных меток заменяются на специальные, генерируемые компилятором, которые гарантированно уникальны и имеют вид **??XXXX**, где **XXXX** - это число в шестнадцатеричной записи из диапазона от **0H** до **0FFFFH**. Это означает, что метки, определяемые программистом, не должны начинаться с символов "??", поскольку в противном случае есть вероятность конфликта имен с метками, генерируемыми Турбо Ассемблером.

3.4.3. Блоки повторения

Полезная разновидность макрокоманд - блоки повторения, одним из применений которых является создание инициализированных таблиц в сегменте данных. Блоки повторения оформляются директивами **REPT**, **IRP** и **IRPC**. Директива **REPT** имеет следующий формат:

```
REPT Выражение
   [{Операторы Ассемблера}]
ENDM
```

Код, содержащийся внутри такого блока повторения, компилируется столько раз, сколько указано операндом директивы **REPT**. Примером ис-

пользования может служить создание массива, инициализированного первыми десятью натуральными числами:

```
InitValue = 1
REPT 10
   DW InitValue
   InitValue = InitValue + 1
ENDM
```

Директива **IRP** позволяет использовать в определении блока повторения переменный параметр и имеет следующий формат:

```
IRP Имя_параметра, <Значение [{, Значение}]>
   [{Операторы Ассемблера}]
ENDM
```

Код, содержащийся внутри такого блока повторения, компилируется столько раз, сколько значений параметра указано в списке. Причем каждый раз параметр принимает очередное значение из списка, которое может быть любой строкой или числом. Пример использования, аналогичный предыдущему:

```
IRP InitVal, <1, 2, 3, 4, 5, 6, 7, 8, 9, 10>
   DW InitVal
ENDM
```

Директива **IRPC** по назначению аналогична директиве **IRP**, она имеет следующий формат:

```
IRPC Имя_параметра, Значение
   [{Операторы Ассемблера}]
ENDM
```

Код, содержащийся внутри такого блока повторения, компилируется столько раз, сколько символов содержится в строке, представляющей значение параметра. Причем каждый раз параметр принимает значение очередного символа этой строки.

3.5. Организация модулей

3.5.1. Компоновка модулей и объединение сегментов

При разработке программы различные ее модули оформляются в разных исходных файлах и компилируются раздельно. При этом основной модуль, с которого начинается выполнение программы, обязательно должен заканчиваться директивой **END** с указанием адреса точки входа, а все остальные модули должны заканчиваться директивой **END** без операнда. Объектные модули, полученные при компиляции исходных модулей, должны быть связаны в загрузочный модуль, прежде чем программу мож-

но будет запускать на выполнение. Это связывание осуществляет компоновщик, который:

– ищет подлежащие связыванию объектные модули;

– определяет адреса всех сегментов;

– определяет смещения всех внешних идентификаторов, которые не смог определить компилятор;

– группирует сегменты и строит загрузочный модуль.

При компоновке объектных модулей одноименные сегменты, определенные в различных модулях, объединяются. Директива **SEGMENT** определения сегмента позволяет указать, каким образом компоновщик должен объединять сегменты. Рассмотрим следующие типы объединения.

AT. Тип объединения **AT** сопровождается выражением, вычисление которого дает константу - номер сегмента, что позволяет явно определить точное размещение сегмента в памяти.

COMMON. Если сегменты в различных объектных модулях имеют одно и то же имя и тип объединения **COMMON**, то они перекрываются таким образом, что имеют один начальный адрес. Длина общего сегмента равна максимальной из длин объединяемых сегментов.

PRIVATE. Если сегмент имеет тип объединения **PRIVATE**, то компоновщик не будет объединять его ни с каким другим сегментом, даже если их имена совпадают. Это позволяет определять сегменты, локальные по отношению к модулю, и избавляет от беспокойства о возможных конфликтах с именами сегментов, определенных в других модулях. Данный тип объединения используется по умолчанию.

PUBLIC. Если сегменты в различных объектных модулях имеют одно и то же имя и тип объединения **PUBLIC**, то в загрузочном модуле они сцепляются в один сегмент. Порядок сцепления определяется опциями компоновщика. Суммарный размер всех объединяемых сегментов не должен превышать **64K**.

STACK. Если сегменты в различных объектных модулях имеют одно и то же имя и тип объединения **STACK**, то в загрузочном модуле они сцепляются в один сегмент стека. При загрузке программы на выполнение регистры **SS:SP** будут адресовать самое старшее слово этого сегмента.

3.5.2. Экспорт и импорт идентификаторов

Очевидно, связываемые объектные модули должны иметь возможность обращаться к некоторым переменным или меткам, определенным в других модулях. Когда идентификатор определяется в объектном модуле, он называется локальным относительно этого модуля. Если же идентификатор

определен не в данном модуле, а в одном из других связываемых объектных модулей, то он называется внешним относительно этого модуля.

В многомодульных программах компилятор необходимо заранее информировать обо всех внешне определенных идентификаторах, которые используются в данном модуле, в противном случае компилятор будет считать их неопределенными и зафиксирует ошибку. Кроме того, чтобы разрешить другим объектным модулям обращаться к некоторым идентификаторам, определенным в данном модуле, он должен явно содержать список идентификаторов, к которым разрешено обращение. Таким образом, модуль может содержать два списка: список внешних идентификаторов, к которым обращается данный модуль (оформляется директивой **EXTRN**), и список локальных идентификаторов, к которым разрешено обращение из других модулей (оформляется директивой **PUBLIC**). Директивы **EXTRN** и **PUBLIC** имеют следующий формат:

```
EXTRN Идентификатор:Тип [{, Идентификатор:Тип}]
PUBLIC Идентификатор [{, Идентификатор}]
```

Здесь идентификаторы представляют собой имена переменных или меток. Так как компилятор для генерации правильного объектного кода должен заранее знать типы всех внешних идентификаторов, с каждым идентификатором в директиве **EXTRN** должен быть связан спецификатор типа. Типом переменной может быть **BYTE**, **WORD** или **DWORD**, а типом метки - **NEAR** или **FAR**.

Одна из главных задач компоновщика заключается в проверке того, что каждому идентификатору в директиве **EXTRN** соответствует точно один идентификатор в директиве **PUBLIC**. При компоновке следующих модулей возникнет ошибка, поскольку внешнему идентификатору **Var4** нет соответствия объявлению **PUBLIC**:

```
; Модуль 1

EXTRN Var4:WORD, Proc1:FAR
PUBLIC Var1

_Data_  SEGMENT
   Var1 DW 10
_Data_  ENDS

_Code_  SEGMENT
        ASSUME DS:_Data_
        ASSUME CS:_Code_
Begin:
   mov    ax, _Data_
   mov    ds, ax
```

```
; Модуль 2

EXTRN Var1:WORD
PUBLIC Proc1

_Stack_ SEGMENT STACK 'STACK'
        DW 4 DUP (?)
_Stack_ ENDS

_Code_  SEGMENT
        ASSUME SS:_Stack_
        ASSUME CS:_Code_
Proc1   PROC FAR
   push   bp
   mov    bp, sp
```

```
   push   Var4                      mov    ax, Var1
   call   FAR PTR Proc1             div    WORD PTR [bp+6]
_Code_  ENDS                        mov    sp, bp
                                    pop    bp
        END Begin                   ret    2
                                 Proc1   ENDP
                                 _Code_  ENDS

                                         END
```

Как было сказано ранее, полный адрес есть сумма адреса сегмента и смещения. Смещения локальных идентификаторов может ввести и вводит компилятор, но смещения внешних идентификаторов и все сегментные номера должны вводиться компоновщиком. Смещения внешних идентификаторов могут быть определены, только когда найдены все соответствующие объектные модули и просмотрены их таблицы экспортируемых идентификаторов. Назначение сегментных номеров осуществляется только после того, как в процессе компоновки точно определено место каждого сегмента в памяти.

Как и в случае локальных переменных, для обращения к внешним переменным необходимо вручную загрузить в сегментный регистр (как правило, **ES**) соответствующий номер внешнего сегмента. При этом нужно связать данный сегментный регистр с именем соответствующего внешнего сегмента с помощью директивы **ASSUME**, в противном случае при каждом обращении к внешней переменной нужно будет использовать явное переопределение сегмента.

3.5.3. Включаемые файлы

Часто оказывается желательным включить один и тот же блок исходного кода в несколько исходных модулей. В этом случае чрезвычайно удобной оказывается директива **INCLUDE**, которая имеет следующий формат:

```
INCLUDE Имя_файла
```

Директива **INCLUDE** указывает компилятору обратиться к соответствующему включаемому файлу и компилировать его, как если бы все строки этого файла были записаны прямо в исходном модуле. При достижении конца включаемого файла компилятор возвращается к строке в исходном модуле, следующей за директивой **INCLUDE**, и возобновляет компиляцию с этой строки. Допускается вложенность включаемых файлов на произвольную глубину. Другими словами, включаемые файлы также могут содержать директивы **INCLUDE**.

Если в операнде директивы **INCLUDE**, определяющим имя включаемого файла, указан путь доступа к файлу, то компилятор будет искать файл только в указанном каталоге. Если же имя файла задано без указания пути, то компилятор сначала ищет файл в текущем каталоге, и если он там отсутствует, то поиск продолжается в каталогах, заданных в параметре командной строки **/I**. Если включаемый файл не будет найден, то компилятор зафиксирует ошибку.

Включаемые файлы полезно использовать для обеспечения доступности макрокоманд в различных модулях программы, а также при совместном использовании констант, объявлений глобальных меток и сегментов данных. Код во включаемых файлах используется редко, однако во включаемых файлах допустимы любые операторы Ассемблера.

3.6. Варианты индивидуальных заданий

Оформите свои индивидуальные задания из работ № 1 и № 2 в виде подпрограмм некоторого модуля. Вызывающие программы оформите в виде отдельных модулей и разместите их в сегменте, отличном от того, в котором расположены подпрограммы. При вызове использовать межсегментный переход.

Оформите одно из заданий в виде макрокоманды и разместите ее во внешнем файле. Включите файл с макрокомандой в вызывающий модуль. Убедитесь, что при использовании нескольких макровызовов объем откомпилированного кода больше, чем при использовании подпрограмм.

3.7. Контрольные вопросы

1) Какие команды предназначены для работы со стеком? Опишите алгоритм их работы.
2) Какие из известных вам команд и зачем изменяют содержимое стека неявно?
3) Можно ли создать сегмент стека размером ровно **64K**? Почему?
4) Какие команды предназначены для организации связи по управлению с подпрограммами? Опишите алгоритм их работы.
5) Какие способы организации связи по данным с подпрограммами вам известны? Раскройте их достоинства и недостатки.
6) Какие режимы адресации поддерживают команды для работы со стеком?
7) Можно ли поместить в стек ровно 1 байт? Почему?

8) Какова роль регистра **BP** при организации подпрограмм? Почему для этих целей используют именно его, а не регистр **SP** или **BX**?

9) Какое значение получает регистр **SP** при запуске программы на выполнение?

10) Объясните назначение директив **NEAR** и **FAR** применительно к подпрограммам. Влияют ли они на машинный код, генерируемый компилятором для команд, в которых использованы эти директивы? Как? Какой режим адресации используется при переходе к **FAR**-подпрограмме по метке?

4. ОБРАБОТКА ПРЕРЫВАНИЙ. ОРГАНИЗАЦИЯ ВВОДА-ВЫВОДА

4.1. Понятие прерывания

Иногда необходимо заставить компьютер автоматически выполнить одну из набора специальных программ, когда в программе или в вычислительной системе возникают определенные условия. Действие, стимулирующее выполнение одной из таких программ, называется прерыванием, а выполняемая программа - процедурой прерывания. Существуют два общих класса прерываний и связанных с ними процедур: внутренние инициируются состоянием или командой процессора, а внешние - сигналом, поступающим в процессор от других компонентов системы. Типичным примером внутреннего прерывания является деление на нуль, внешнего - сигнал от устройства ввода-вывода, требующего обслуживания процессором.

Процедура прерывания аналогична обычной подпрограмме в том отношении, что переход к ней осуществляется из любой другой программы, а после выполнения процедуры прерывания возврат происходит в прерванную программу. Запись процедуры прерывания должна быть такой, чтобы после обработки прерывания прерванная программа продолжалась так, как будто ничего не произошло. Это означает, что необходимо запоминать и восстанавливать регистр флагов и регистры, используемые процедурой прерывания, и что возврат должен происходить к команде, следующей за последней командой, выполненной до прерывания.

Процедура прерывания отличается от обычной подпрограммы тем, что, поскольку может вызываться из различных программ или вообще инициироваться внешними событиями, она не связана с какой-то конкретной программой и ее сегментами. Поэтому взаимодействие по данным с процеду-

рой прерывания невозможно организовать через стек или глобальные переменные: как правило, используют регистры или общие области памяти, непосредственно доступные обеим программам.

Независимо от вида прерывания возникающие при этом действия одинаковы и называются последовательностью прерывания:

– текущее содержимое регистра флагов, а также регистров **CS** и **IP** включается в стек в указанной последовательности;

– в регистры **CS** и **IP** помещается новое содержимое из двойного слова, адрес которого определяется типом прерывания; новое содержимое регистров **CS** и **IP** определяет начальный адрес процедуры прерывания;

– флаги **IF** и **TF** сбрасываются;

– по завершении процедуры прерывания возврат в прерванную программу осуществляется командой, последовательно извлекающей из стека слова в регистры **IP**, **CS** и регистр флагов.

Двойное слово, из которого загружаются регистры **CS** и **IP** при возникновении прерывания, и в котором находится начальный адрес процедуры прерывания, называется вектором прерывания. Каждому типу прерывания назначено число из диапазона 0…255 и адрес вектора прерывания вычисляется процессором путем умножения типа на 4. Следовательно, векторы прерываний размещаются в первых 1024 байтах оперативной памяти и их никогда не следует использовать для других целей. Некоторые из 256 типов прерываний резервируются операционной системой и должны инициализироваться при ее загрузке. Часть типов прерываний резервируется для периферийных устройств, они используются драйверами соответствующих устройств. Первые пять типов прерываний определены явно, т. е. могут быть использованы самим процессором:

– **INT0** возникает из-за ошибки деления, когда результат команды **DIV** или **IDIV** не помещается в регистр-приемник; на это прерывание не влияют флаги **IF** и **TF**;

– **INT1** обеспечивает пошаговый режим выполнения программ и управляется флагом **TF**: если флаг **TF** установлен, то по окончании выполнения каждой команды возникает прерывание **INT1**; это прерывание используется, в основном, при отладке программ и не зависит от флага **IF**;

– **INT2** соответствует немаскируемому внешнему прерыванию и не зависит от флага **IF**; это прерывание инициируется сигналом, подаваемым на вход **NMI** процессора;

– **INT3** также используется при отладке программ, когда применение пошагового режима ограничено: это прерывание инициируется командой **INT**, которую помещают в некоторые ключевые точки программы с це-

лью выдачи информации о состоянии процессора; это прерывание не зависит от флагов **IF** и **TF**;

– **INT4** инициируется командой **INTO**, которая вызывает это прерывание, только если флаг **OF** установлен; эта команда используется при отладке для контроля переполнения при выполнении арифметических операций.

Любые прерывания можно инициировать и программным путем, используя соответствующие команды процессора. Процессор Intel 8086 поддерживает команды, специально предназначенные для работы с прерываниями (табл. 4.1).

Таблица 4.1

Команды процессора Intel 8086 для работы с прерываниями

Мнемоника и формат	Описание	Логика
INT	прерывание с типом 3	**PUSHF;** **TF := 0;** **IF := 0;** **CALL FAR [0CH]**
INT TYPE	прерывание с типом **TYPE**	**PUSHF;** **TF := 0;** **IF := 0;** **CALL FAR [TYPE*4]**
INTO	прерывание при переполнении	**if OF = 1 then** **PUSHF;** **TF := 0;** **IF := 0;** **CALL FAR 10H** **end**
IRET	возврат из обработчика прерывания	**POP IP;** **POP CS;** **POPF**

4.2. Общие принципы ввода-вывода

4.2.1. Порты ввода-вывода

Все периферийные устройства и внешняя память подключаются к системной шине через интерфейсы. Каждый интерфейс имеет набор регистров, называемых портами ввода-вывода, через которые процессор и память взаимодействуют с внешними устройствами. Одни порты предназначены для буферирования данных, другие - для хранения информации о состоянии устройства и интерфейса, которую может проверить процессор, а

третьи для восприятия приказов от процессора, управляющих действиями интерфейса и устройства. Все взаимодействие процессора с внешним миром осуществляется через порты ввода-вывода в интерфейсе. Следовательно, в процессоре должны быть средства для передачи информации в порты и из портов.

В процессоре Intel 8086 адресные пространства памяти и портов ввода-вывода четко разделены. При этом в шине управления предусмотрены специальные линии, которые определяют, к какому адресному пространству относится адрес на шине адреса. Для правильной установки сигналов на эти линии, процессор Intel 8086 поддерживает специальные команды для взаимодействия с портами ввода-вывода.

В процессоре Intel 8086 программное взаимодействие с портами ввода-вывода осуществляется командами ввода **IN** и вывода **OUT** (табл. 4.2). Обе команды могут передавать байт или слово и имеют длинную и короткую формы. Передача байта или слова определяется регистром-приемником (для **IN**) или регистром-источником (для **OUT**): если указан **AL**, то передается байт, если **AX**, то слово. Длинная форма команды (2 байта) предполагает, что адрес порта указан явно во втором (для **IN**) или первом (для **OUT**) байте команды. При этом адрес порта должен находиться в диапазоне 0…255. Короткая форма команды (1 байт) предполагает, что адрес порта указан в регистре **DX**. При этом адрес порта должен находиться в диапазоне 0…65535, что обеспечивает доступ к любому порту в адресном пространстве ввода-вывода.

Таблица 4.2

Команды процессора Intel 8086 для работы с портами ввода-вывода

Мнемоника и формат	Описание	Логика
IN AL, PORT	Ввести байт, длинная	**AL := [PORT]**
IN AX, PORT	Ввести слово, длинная	**AX := [PORT]**
IN AL, DX	Ввести байт, короткая	**AL := [DX]**
IN AX, DX	Ввести слово, короткая	**AX := [DX]**
OUT PORT, AL	Вывести байт, длинная	**[PORT] := AL**
OUT PORT, AX	Вывести слово, длинная	**[PORT] := AX**
OUT DX, AL	Вывести байт, короткая	**[DX] := AL**
OUT DX, AX	Вывести слово, короткая	**[DX] := AX**

4.2.2. Способы организации ввода-вывода

Существуют три основные разновидности ввода-вывода: программный ввод-вывод, ввод-вывод по прерываниям и прямой доступ к памяти (ПДП). Первые два опираются на передачи байтов или слов между портами и памятью, причем эти передачи осуществляются через регистр процессора. Таким образом, если необходимо ввести слово из устройства в память, его приходится сначала копировать из соответствующего порта в регистр процессора, а затем - из регистра процессора в память. Для ввода последовательности слов необходимо организовать программный цикл. При выводе действия аналогичны.

Прямым доступом к памяти управляет специальная схема - контроллер ПДП. Контроллер ПДП по шине управления запрашивает у процессора использование шины и осуществляет необходимые передачи самостоятельно, без участия процессора, так что при вводе каждое слово по мере поступления помещается непосредственно в память, а при выводе слова передаются из памяти в устройство минуя процессор.

Программный ввод-вывод

При программном вводе-выводе программа определяет требующие обслуживания интерфейсы, проверяя биты готовности в их регистрах состояния (опрос). При вводе-выводе по прерываниям интерфейс инициирует внешнее прерывание, когда устройство имеет данные для ввода или готово принимать их, а сама операция ввода-вывода реализуется процедурой прерывания.

Программный ввод-вывод заключается в непрерывной проверке интерфейса и выполнении операции ввода-вывода, когда его состояние показывает наличие данных для ввода или когда его буферный выходной регистр готов принимать данные от процессора. При этом часто используется механизм буферирования, когда обработка данных не начинается, пока не будет введено некоторое их количество. В процессе ввода данные сначала временно запоминаются в наборе смежных ячеек памяти, называемых буфером. Как только буфер заполнен, начинается обработка, по окончании которой вся процедура повторяется.

Ввод-вывод по прерываниям

Хотя программный ввод-вывод оказывается простым, он связан со значительными потерями времени на ожидание активного состояния готовности. Это может быть допустимым, если в процессе ввода не нужно выполнять другую обработку, но если эту обработку задерживать нельзя, то дан-

ный подход непригоден. В этом случае используется ввод-вывод по (внешним) прерываниям. Внешнее прерывание инициируется сигналом, подаваемым на вход **INT** или **NMI** процессора. Прерывание, инициируемое сигналом на входе **NMI**, называется немаскируемым: оно вызывает прерывание типа 2 независимо от состояния флага **IF**. Сигналы немаскируемых прерываний обычно формируют схемы, фиксирующие события, требующие немедленной обработки (отказ питания, импульсы таймера и др.). Прерывание на входе **INT** маскируется флагом **IF**: если он сброшен, то прерывание не распознается. Если флаг **IF** установлен и возникает маскируемое прерывание, то процессор через выход **INTA** возвращает в интерфейс сигнал подтверждения и инициирует последовательность прерывания. Сигнал подтверждения заставляет интерфейс выдать в процессор по шине данных байт, который определяет тип прерывания и, следовательно, адрес указателя на процедуру-обработчик. Тип маскируемого внешнего прерывания должен находиться в пределах 5…255.

Когда обслуживания запрашивают несколько интерфейсов, порядок удовлетворения запросов зависит от встроенного в систему механизма приоритетов, который может быть реализован программно или аппаратно.

4.3. Программируемый контроллер прерываний

Наиболее гибкий программно-аппаратный механизм приоритетов реализуется программируемой схемой управления приоритетными прерываниями, которая входит в логику управления шиной. Для работы с микропроцессором Intel 8086 предлагается программируемый контроллер прерываний (ПКП) Intel 8259A, который поддерживает 8 уровней прерываний от восьми различных устройств. Основные функции контроллера:

– фиксация запросов на прерывания от внешних источников;

– программное маскирование поступающих запросов;

– присвоение фиксированных или циклически изменяемых приоритетов входам контроллера, на которые поступают запросы;

– инициация вызова процедуры обработки поступившего аппаратного прерывания.

Количество обслуживаемых внешних источников прерываний может быть увеличено путем каскадирования нескольких контроллеров.

ПКП может находиться в одном из двух состояний: настройки и обслуживания запросов. В состоянии настройки контроллер принимает управляющие слова инициализации (Initialization Command Words, **ICW**), в состоянии обслуживания - операционные управляющие слова (Operation Control Words, **OCW**).

4.3.1. Основные элементы ПКП

Схема управления чтением/записью. Основной функцией этого блока является прием команд от микропроцессора и передача ему информации о состоянии ПКП. Обмен данными с микропроцессором осуществляется по шине данных через специальный 8-разрядный буфер данных. В состав блока входят регистры управляющих слов **ICW** и **OCW**. Схема управляется входами **CS**, **RD**, **WR** и **A0**. Низкий уровень сигнала на входе **CS** разрешает выполнение обмена с ПКП. Низкий уровень сигнала на входе **WR** (Write) разрешает микропроцессору выводить управляющие слова **ICW** и **OCW** для приема их ПКП. Низкий уровень сигнала на входе **RD** (Read) разрешает ПКП передать микропроцессору информацию о состоянии специальных регистров **IRR**, **ISR** и **IMR**, которые описаны ниже.

Все управляющие слова **ICW** и **OCW** принимаются контроллером в виде 9-разрядных значений. Разряды 0-7 передаются через 8-разрядный буфер данных. Старший разряд **A0** устанавливается в 0 или 1 в зависимости от того, через какой из двух возможных портов ввода-вывода (четный или нечетный соответственно) было передано управляющее слово.

Регистр запросов на прерывания (Interrupt Request Register, **IRR**) обслуживается через входы **IR0-IR7** контроллера. Сигнал на одном входов **IR0-IR7** - это запрос на прерывание соответствующего уровня (0-7). В соответствии с сигналом запроса на прерывание схемой управления устанавливается соответствующий бит в регистре **IRR**.

Регистр состояния (In-Service Register, **ISR**) описывает в битах 0-7 прерывания каких уровней (0-7) в данный момент обрабатываются.

Регистр маскирования запросов на прерывания (Interrupt Mask Register, **IMR**) описывает, прерывания каких уровней в настоящий момент замаскированы. Единичное значение бита в **IMR** указывает на то, что прерывание соответствующего уровня при появлении запроса в **IRR** блокируется.

Схема обработки приоритетов определяет, прерывание какого уровня в данный момент является наиболее приоритетным для выполнения.

Схема управления ПКП формирует сигнал запроса на прерывание, поступающий на вход **INT** микропроцессора. Если флаг **IF** процессора установлен, процессор отвечает сигналом по линии **INTA**, после чего сбрасывается в 0 разряд **IRR** и устанавливается в 1 разряд **ISR**, соответствующие уровню обрабатываемого прерывания. После получения второго сигнала подтверждения от процессора по линии **INTA**, ПКП передает на шину данных 8-битовый номер прерывания.

Схема каскадирования отвечает за работу каскада из нескольких контроллеров. При подключении к ведущему контроллеру выход **INT** каждого ведомого подключается к одному из входов **IR0-IR7** ведущего. Далее этот сигнал передается ведущим на вход **INT** процессора. Когда процессор возвращает сигнал **INTA**, ведущий контроллер не только устанавливает бит в **ISR** и сбрасывает бит в **IRR**, но и выдает на свои выходы **CAS0-CAS2** номер уровня прерывания, к которому подключен ведомый, пославший запрос на прерывание. Сигналы по линии **CAS0-CAS2** принимаются всеми ведомыми, однако обрабатываются только тем, который подключен к линии **IR** с соответствующим номером.

4.3.2. Режимы работы ПКП

Возможны несколько режимов обслуживания прерываний.

1) Режим фиксированных приоритетов. В этом режиме контроллер находится сразу после инициализации. Запросы прерываний имеют жесткие приоритеты от 0 до 7 (0 - высший) и обрабатываются в соответствии с приоритетами. Прерывание с меньшим приоритетом никогда не будет обработано, если в процессе обработки прерываний с более высокими приоритетами постоянно возникают запросы на эти прерывания.

2) Автоматический сдвиг приоритетов. В этом режиме дается возможность обработать прерывания всех уровней без их дискриминации. Например, после обработки прерывания уровня 4 ему автоматически присваивается низший приоритет, при этом приоритеты для всех остальных уровней циклически сдвигаются и прерывания уровня 5 будут иметь в данной ситуации высший приоритет и, следовательно, возможность быть обработанными.

3) Программно-управляемый сдвиг приоритетов. Программист может сам передать команду циклического сдвига приоритетов ПКП, задав соответствующее управляющее слово. В команде задается номер уровня, которому требуется присвоить максимальный приоритет. После выполнения такой команды устройство работает так же, как и в режиме фиксированных приоритетов, с учетом их сдвига. Приоритеты сдвигаются циклически, таким образом если максимальный приоритет был назначен уровню 3, то уровень 2 получит минимальный и будет обрабатываться последним.

4) Автоматическое завершение обработки прерывания. В обычном режиме работы процедура обработки аппаратного прерывания должна перед завершением очистить свой бит в **ISR** специальной командой, иначе новые прерывания не будут обрабатываться ПКП. В этом режиме нужный бит в **ISR** автоматически сбрасывается в тот момент, когда начинается об-

работка прерывания и от процедуры не требуется выдавать команду завершения обработки прерывания (**EOI**). Сложность работы в данном режиме обусловлена тем, что все процедуры обработки аппаратных прерываний должны быть повторно входимыми, т. к. за время их работы могут повторно возникнуть прерывания того же уровня.

5) Режим специальной маски. Данный режим позволяет отменить приоритетное упорядочение обработки запросов и обрабатывать их по мере поступления. После отмены режима специальной маски предшествующий порядок приоритетов уровней сохраняется.

6) Режим опроса. В этом режиме аппаратные прерывания не происходят автоматически. Появление запросов на прерывание должно определяться считыванием **IRR**. Данный режим позволяет так же получить от ПКП информацию о наличии запросов на прерывания и, если они имеются, номер уровня с максимальным приоритетом, по которому есть запрос.

4.3.3. Программирование ПКП

Для вывода информации в ПКП используются 2 порта ввода-вывода. Порт с четным адресом (обычно это порт **20H**) и порт с нечетным адресом (обычно **21H**). Через эти порты могут быть переданы 4 слова инициализации (**ICW1-ICW4**), задающие режим работы ПКП, и 3 операционных управляющих слова (**OCW1-OCW3**).

В порт с четным адресом выводятся **ICW1**, **OCW2** и **OCW3**. **OCW2** отличается от **OCW3** тем, что бит 3 в **OCW2** равен 0, а в **OCW3** равен 1. В то же время бит 4 в **OCW2** и **OCW3** равен 0, а в **ICW1** равен 1. Таким образом по значению, выводимому в порт с четным адресом, однозначно определяется, в какой регистр (**ICW1**, **OCW2** или **OCW3**) заносятся данные.

Порт с нечетным адресом используется для вывода **ICW2**, **ICW3**, **ICW4** и **OCW1**. Неоднозначности интерпретации данных в этом случае так же не возникает, т. к. слова инициализации **ICW2-ICW4** должны непосредственно следовать за **ICW1**, выведенным в порт с четным адресом и выводить в промежутке между ними **OCW1** не следует, оно не будет опознано контроллером.

Инициализация ПКП

Выводом в порт с четным адресом управляющего слова инициализации **ICW1** начинается инициализация ПКП. В процессе инициализации контроллер последовательно принимает управляющие слова **ICW1-ICW4**. При наличии в системе одного контроллера **ICW3** не выводится. Наличие

ICW4 определяется содержанием **ICW1**. При наличии каскада из нескольких ПКП каждый из них инициализируется отдельно.

Формат управляющего слова **ICW1** показан на рис. 4.1.

Рис. 4.1. Формат управляющего слова **ICW1**

Бит 4, равный 1, определяет, что выводится **ICW1**, а не **OCW2** или **OCW3**.

Бит 3 (**LTIM**), равный 0, задает запуск запросов фронтом, при этом действует описанная выше схема: бит в **IRR** сбрасывается при установке соответствующего бита в **ISR**.

Бит 2 (**ADI**) используется только в ПЭВМ на базе микропроцессоров Intel 8080/8085.

Бит 1 (**SNGL**) указывает на наличие в системе одного контроллера (если равен 1) или каскада из нескольких контроллеров (если равен 0). Если этот бит равен 1, то **ICW3** не выводится в процессе инициализации и следом за **ICW2** сразу должно следовать **ICW4**.

Бит 0 (**IC4**) определяет, будет ли выводиться **ICW4**. Если **ICW4** не выводится (бит 0 = 0), то оно автоматически заполняется нулями. Наличие **ICW4** обязательно, т. к. тип микропроцессора 8086/8088 задается в нем значением 1 в одном из битов.

Управляющее слово **ICW2** задает номер вектора прерывания для прерываний уровня 0 (например 8 для IBM PC, у которых по уровню 0 происходят прерывания от таймера). Так как вектора аппаратных прерываний располагаются подряд друг за другом, вывод в **ICW2** значения 8 не только задает восьмой вектор для таймера, но и девятый для прерываний уровня 1, десятый для прерываний уровня 2 и т. д.

Управляющее слово **ICW3** выводится только при наличии каскада и имеет разный формат для ведущего и ведомых контроллеров. **ICW3** ведущего указывает, к каким входам **IR0-IR7** подключены ведомые контроллеры, при этом соответствующие биты устанавливаются в 1. Остальные биты при этом равны 0. Так, **ICW3** следующего вида

A0	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	1	0

задает, что в каскаде имеется 2 ведомых контроллера, подключенных к входам **IR1** и **IR4**.

ICW3 ведомого ПКП в трех младших битах задает номер уровня, на котором работает ведомый контроллер. Для ведомого контроллера, работающего на уровне 1, **ICW3** будет выглядеть следующим образом:

A0	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1

Формат управляющего слова **ICW4** показан на рис. 4.2. Здесь бит 0 (**mPM**) определяет, с каким микропроцессором работает ПКП (0 - 8080/8085, 1 - 8086/8088).

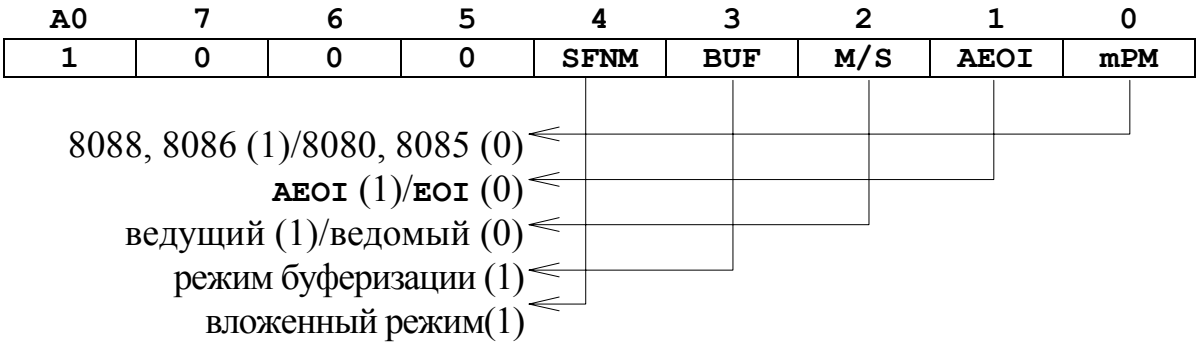


Рис. 4.2. Формат управляющего слова **ICW4**

Бит 1 (**AEOI**), равный 1, задает режим автоматического завершения обработки прерывания, описанный выше. Если этот бит равен 0, действует обычное соглашение: процедура обработки аппаратного прерывания должна сама сбрасывать свой бит в **ISR**.

Бит 2 (**M/S**) игнорируется, если бит 3 **BUF = 0**. При наличии одного контроллера и **BUF = 1** устанавливается в 1. При наличии каскада должен быть равен 1 только для ведущего контроллера.

Бит 4 (**SFNM**) устанавливает специальный вложенный режим, применяемый при каскадировании для определения приоритетов запросов от разных контроллеров.

После инициализации ПКП готов к работе в заданном режиме. Для изменения режимов работы, задаваемых при инициализации требуется переинициализировать его заново.

Управление ПКП

В процессе работы с ПКП без переинициализации можно:

- маскировать и демаскировать аппаратные прерывания;
- изменять приоритеты уровней;
- выдать команду завершения обработки аппаратного прерывания;
- установить/сбросить режим специальной маски;
- перевести контроллер в режим опроса.

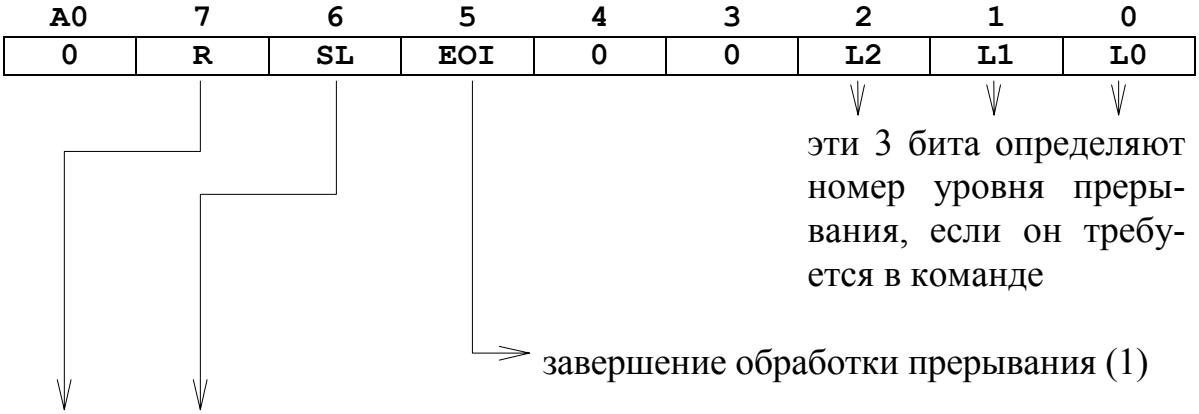
Последнее позволяет считать состояние регистров **ISR** и **IRR**; для этого потребуется вывести в порты ПКП одно из трех операционных слов **OCW1-OCW3**.

Формат операционного слова **OCW1** следующий:

A0	7	6	5	4	3	2	1	0
1	M7	M6	M5	M4	M3	M2	M1	M0

Единичное значение одного из битов **M0-M7** означает, что прерывания соответствующего уровня (**IR0-IR7**) маскируются и не будут обрабатываться контроллером.

Операционное слово **OCW2** предназначено для вывода команды завершения обработки аппаратного прерывания (End Of Interrupt, **EOI**), циклического сдвига и явного изменения приоритетов уровней. Назначение битов **OCW2** показано на рис. 4.3.



0	0	Используется вместе с **EOI = 1**
0	1	Специфицированный **EOI**
1	0	Циклический сдвиг приоритетов влево на 1
1	1	Назначение низшего приоритета уровню **L0 - L2**

Рис. 4.3. Формат операционного слова **OCW2**

Процедура обработки аппаратного прерывания должна перед завершением очистить свой бит в **ISR** выводом команды **EOI**. Существует два варианта команды **EOI**: обычный и специфицированный. Обычный **EOI** очищает бит в **ISR**, соответствующий прерыванию с максимальным приоритетом. Специфицированный **EOI** (**R = 0**, **SL = 1**, **EOI = 1**, **L0-L2** равно номеру уровня прерывания) очищает в **ISR** бит, соответствующий прерыванию с номером, указанным в **L0-L2** независимо от его приоритета.

Команды с битом **R = 1** позволяют изменить приоритеты уровней. Циклический сдвиг приоритетов сдвигает приоритеты влево на единицу, при этом, если после обычного распределения приоритетов выдать команду циклического сдвига, уровень 0 получит низший приоритет, уровень 1 - наивысший, уровень 2 - следующий за ним и т. д. Команда явного назначения низшего приоритета одному из уровней изменяет приоритеты остальных уровней циклически. Таким образом, если задан низший приоритет уровню 5, то уровень 6 получит наивысший.

Операционное слово **OCW3** позволяет установить или отменить режим специальной маски, перевести контроллер в режим опроса и прочитать содержимое **IRR** и **ISR**. Назначение битов **OCW3** показано на рис. 4.4.

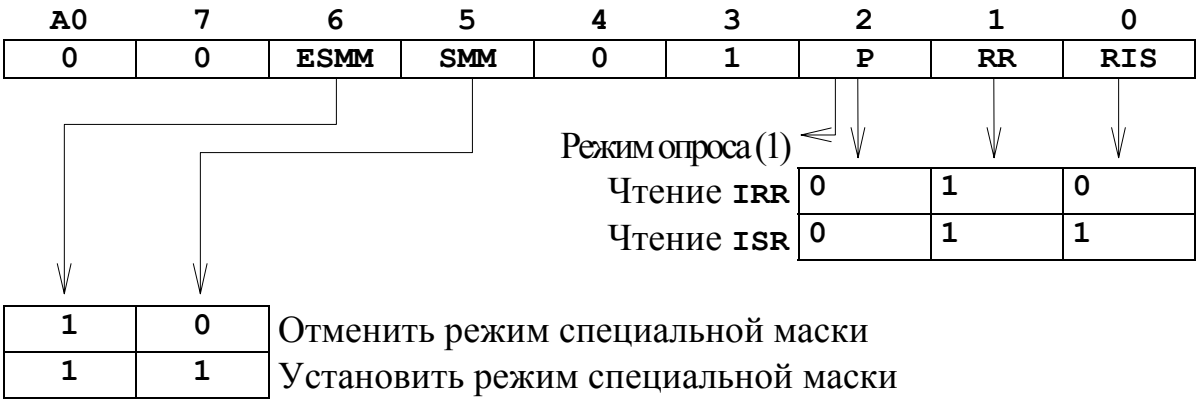


Рис. 4.4. Формат операционного слова **OCW3**

Установка бита **P** переводит контроллер в режим опроса. Если после этого считать данные из порта с четным адресом, в регистр **AL** загрузится байт следующего содержания:

7	6	5	4	3	2	1	0
I	0	0	0	0	L2	L1	L0

Если **I = 1**, значит имеются запросы на прерывания и тогда **L0-L2** - это номер уровня с наивысшим приоритетом, по которому имеется запрос на прерывание.

Если **P = 0**, то можно считать информацию из **ISR** или **IRR**. Для этого необходимо выдать команду чтения **ISR** или **IRR** и затем считать значение из порта с нечетным адресом.

Установка бита **ESMM** позволяет в зависимости от значения бита **SMM** установить или отменить режим специальной маски.

4.4. Варианты индивидуальных заданий

1) Реализуйте на языке Ассемблера процедуры **GetIntVect** и **SetIntVect**, принимающие два параметра: номер прерывания и дальний указатель. Процедура **GetIntVect** должна сохранять в переданный по ссылке указатель адрес процедуры обработки прерывания с указанным номером. Процедура **SetIntVect** должна копировать переданный по значению указатель в соответствующий указанному номеру элемент таблицы векторов прерываний.

2) Напишите собственную процедуру обработки прерывания от клавиатуры. Клавиатуре соответствуют **IRQ1** и **INT9**. Сканкод нажатой клавиши может быть получен чтением порта **60H**. После обработки необходимо установить бит 7 (подтверждение от клавиатуры) порта **61H**, а затем сбросить его.

3) Напишите процедуру, которая программирует ПКП так, чтобы он перестал или снова стал воспринимать прерывания с указанным номером. Запретите прерывания от клавиатуры. Убедитесь в том, что результат достигнут. Затем верните ПКП в исходное состояние и убедитесь, что прерывания от клавиатуры обрабатываются.

4) Переведите ПКП в режим опроса и организуйте программный ввод с клавиатуры с буферизацией в 10 символов. По окончании ввода или по нажатии клавиши **ESC** верните ПКП в исходное состояние.

5) Организуйте чтение клавиатурного ввода по прерыванию с буферизацией в 10 символов. По окончании ввода или по нажатии клавиши **ESC** восстановите прежнее значение вектора прерывания **INT9**.

6) Напишите процедуру обработки прерывания от системного таймера. Таймеру соответствуют **IRQ0** и **INT8**. Организуйте вызов прежнего обработчика прерывания от таймера внутри собственного обработчика. Выясните, отстают ли системные часы, если не выполнять вызов прежнего обработчика.

4.5. Контрольные вопросы

1) Почему при переполнении в ходе выполнения арифметических операций не возникает прерывания? Какая команда специально предназначена для инициирования прерывания в такой ситуации?

2) В чем сходство и различие между обычными подпрограммами и процедурами прерываний?

3) Опишите последовательность прерывания.

4) Укажите назначение прерываний **INT0-INT4**.

5) В чем состоит главный недостаток программного ввода-вывода в сравнении с вводом-выводом по прерываниям? В чем принципиальное отличие ПДП от других способов организации ввода-вывода?

6) Поясните различие между маскируемыми и немаскируемыми прерываниями. Опишите процедуру инициирования маскируемого прерывания.

7) В чем преимущества коротких форм команд **IN** и **OUT** в сравнении с их длинными формами?

8) Почему для передачи параметров подпрограмме прерывания нельзя использовать стек? Как организуется связь по данным с подпрограммой прерывания?

9) Зачем флаги **IF** и **TF** в ходе процедуры прерывания сбрасываются? Влияет ли это на прерванную программу по завершении обработки прерывания? Как?

10) Можно ли для программного инициирования прерывания использовать команду **CALL**? Как? Где это может быть применимо?

Библиографический список

1. Абель П. Язык ассемблера для IBM PC и программирование: Пер. с англ. - М.: Высш. шк., 1992

2. Лю Ю-Чжен, Гибсон Г. Микропроцессоры семейства 8086/8088. Архитектура, программирование и проектирование микрокомпьютерных систем: Пер. с англ. - М.: Радио и связь, 1987.

3. Нортон П., Соухэ Дж. Язык ассемблера для IBM PC: Пер. с англ. - М.: Компьютер, 1992

4. Пильщиков В. Н. Программирование на языке ассемблера IBM PC. - М.: Диалог-МИФИ, 1997

Оглавление

Учебное издание

Головинов Иван Александрович

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА i8086

Практикум

Редактор А. П. Володина

ЛР № 020275. Подписано в печать 26.09.03.
Формат 60x84/16. Бумага для множит. техники. Гарнитура Таймс.
Печать офсетная. Усл. печ. л. 4,18. Уч.-изд. л. 4,77. Тираж 70 экз.
Заказ

Редакционно-издательский комплекс
Владимирского государственного университета.
600000, Владимир, ул. Горького, 87.