ПЕРВАЯ ПРОГРАММА НА АССЕМБЛЕРЕ

**Цель работы:** ознакомиться с основными требованиями к программам на языке Ассемблера.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Первым этапом создания программы является написание текстового файла на специальном языке - Ассемблере. Этот файл создается любым текстовым редактором.

В дальнейшем этот текстовый файл будет обработан специ­альной программой-транслятором для преобразования в машинный код.

В этой и последующих лабораторных работах приведены правила написания программ на языке Ассемблера фирмы Microsoft - MASM. Turbo Assembler, на котором будут выполняться лабораторные работы, по умолчанию работает в режиме MASM

Программа, написанная на языке Ассемблера, состоит из строк. Максимальная длина строки - 80 символов. Строки отделяются друг от друга символом «Enter».

Основной формат строки.

[метка] команда [операнд\_1][,операнд\_2] [;комментарий]

В квадратных скобках указаны необязательные поля.

Метка (если имеется), команда и первый операнд отделя­ются друг от друга по крайней мере одним символом пробела или табуляции.

МЕТКА - это символьное имя или идентификатор, помечающее данную строку для последующего обращения. Символьное имя может содержать в принципе любое количество символов, но транслятор распознает только первые 31 сим­вол. По этому символьное должно быть уникально по первому 31 символу. Допустимые символы:

буквы: от A до Z и от a до z (в режиме, принятом по умолчанию, транслятор не различает прописные и строчные буквы);

цифры: от 0 до 9;

специальные символы: знак вопроса (?),

точка (.) - только первый символ,

коммерческое «а» (@),

подчеркивание (\_),

доллар ($).

Первым символом в символьном имени должна быть буква или специаль­ный символ.

Имена регистров зарезервированы. Их допускается исполь­зовать только для обозначения регистров.

КОМАНДА обозначает действие, которое должен произвести процессор или транслятор с языка Ассемблер. В ка­честве команд могут использоваться команды процессора (собс­твенно команды) и директивы (псевдокоманды) Ассемблеру. Команды процессора на этапе трансляции преобразуются в машинный код и выполняются в процессе выполнения программы. Директивы указывают программе – транслятору, что ей нужно делать. Они выполняются на этапе трансляции и в машинный код не преобразуются.

ОПЕРАНД определяет элемент, над котором производит дей­ствие команда.

Команда может иметь один или два операнда или не иметь их совсем.

RET - команда без операнда;

INC CX - команда с одним операндом;

ADD AX,12 - команда с двумя операндами.

Если команда имеет два операнда, они разделяются запя­той (,). При использовании двух операндов первым указывается операнд-приемник, вторым - операнд-источник. То есть резуль­тат операции заносится в первый операнд.

Примеры операндов

MOV AX,BX ;переслать содержимое BX в AX

MOV AX,WORDA ;переслать содержимое ячейки памяти под

;именем WORDA в AX

MOV AX,[BX] ;переслать содержимое памяти по адресу ука­занному в BX в

;AX

MOV AX,25 ;переслать число 25 в регистр AX

КОММЕНТАРИЙ начинается с точки с запятой (;). Может за­нимать всю строку или следовать за командой в той же строке. Комментарий появляется только в листингах и не приводят к ге­нерации машинных кодов. В комментарии допускаются любые сим­волы основной клавиатуры.

ДИРЕКТИВЫ (или псевдокоманды).

Это команды, которые позволяют управлять процес­сом трансляции и формирования листинга.

Директива SEGMENT. Используется для описания используе­мых в программе сегментов. Любая программа содержит по край­ней мере один сегмент - сегмент кода. Могут присутствовать также сегмент стека и сегмент данных, а также дополнительный сегмент. Формат директивы:

Имя Директива Операнд

имя SEGMENT [атрибуты]

; содержимое сегмента

имя ENDS

Имя сегмента должно обязательно присутствовать, быть уникальным и соответствовать соглашениям для символьных имен в Ассемб­лере. Директива EDS означает конец сегмента. Обе директивы SEGMENT и ENDS должны иметь одинаковые имена. Атрибуты сегмента на данном этапе нам понадобятся только для описания сегмента стека.

Сегмент стека определяется следующим образом:

имя SEGMENT PARA STACK 'Stack'

DB 30 DUP(?)

имя ENDS

Директива PROC. Сегмент кода содержит выполняемые ко­манды программы. Он может содержать в себе одну или несколь­ко процедур, определяемых директивой PROC. Сегмент, содержа­щий только одну процедуру, имеет вид

имя\_сегмента SEGMENT

имя\_процедуры PROC FAR

; текст процедуры

RET ; команда выхода из процедуры

имя\_процедуры ENDP

имя\_сегмента ENDS

Имя процедуры должно обязательно присутствовать, быть уникальным и соответствовать соглашениям по именам в Ассемб­лере. Оператор FAR показывает, что данная процедура может вызываться из другого сегмента.

Директива ASSUME. Процессор использует регистр SS для адресации стека, регистр DS - для адресации сегмента данных и регистр CS - для адресации сегмента кода. Ассемблеру необ­ходимо сообщить назначение каждого сегмента.

Директива Операнд

ASSUME SS:имя\_сегм\_стека,DS:имя\_сегм\_данных,CS:имя\_сегм\_кода

где имя\_сегм\_стека, имя\_сегм\_данных, имя\_сегм\_кода - имена соответственно сег­ментов стека, данных и кода, присвоенные им директивой SEGMENT.

Операнды могут записываться в любой последовательности. Регистр ES также может присутствовать среди операндов. Если программа не использует регистр ES, то его можно опустить или указать ES:NOTHING.

Директива END. Полностью завершает всю программу.

Директива Операнд

END [точка входа в программу]

Операнд может быть опущен, если программа не предназна­чена для выполнения (ассемблируется только для определения данных или должна быть скомпонована с другим (главным) моду­лем). Для обычной программы с одним (главным) модулем опе­ранд содержит имя, указанное в директиве PROC, которое было обозначено, как FAR.

ИНИЦИАЛИЗАЦИЯ ПРОГРАММЫ.

Существуют два основных типа загрузочных модулей: EXE и COM. Рассмотрим требования к EXE-программам. Система DOS имеет четыре требования для инициализации ассемблерной EXE-программы.

1. Необходимо указать Ассемблеру, какие сегментные ре­гистры каким сегментам соответствуют (осуществляется директивой ASSUME).

2. Требуется сохранить в стеке адрес, находящийся в ре­гистре DS.

Загрузочному модулю в оперативной памяти непосредствен­но предшествует 256-байтовая (100Н) область, именуемая пре­фиксом программного сегмента (PSP), в которой DOS хранит свою служебную информацию. После загрузки ЕХЕ-программы в оперативную память регистр DS содержит адрес начала (базу) PSP.

Пользовательская программа должна сохранить этот адрес, по­местив его в стек. Позже команда RET возьмет этот адрес из стека для возврата в DOS.

3. Необходимо записать в стек ноль.

4. Необходимо загрузить в DS адрес сегмента данных. Загрузчик DOS устанавливает правильные адреса стека в регистре SS и сегмента кодов в регистре CS. Поскольку прог­рамма загрузчика использует регистр DS для других целей, не­обходимо явно инициализировать регистр DS.

Выход из программы и возврат в DOS сводится к использо­ванию команды RET.

Команда RET обеспечивает выход из пользовательской про­граммы и возврат в DOS, используя для этого адрес, записан­ный в стек в начале программы из регистра DS. Делает это она следующим образом. Команда RET для процедуры типа FAR извлекает из стека два верхних слова и помещает их в регистры IP и CS. Верхние два слова в стеке содержат ноль и адрес начала PSP. Таким образом, ноль попадает в IP, а адрес начала PSP – в CS. Таким способом команда RET передает управление по адресу база\_PSP: 0, то есть на нулевую ячейку PSP, а в этой ячейке содержится команда системного возврата в DOS.

Пример программы.

SEGST SEGMENT PARA STACK 'Stack'

DB 30 DUP(?)

SEGST ENDS

;-------------------------------------

SEGDT SEGMENT

A DB 5 ;эти данные не используются в приведенной программе

B DW 300 ; и приведены для иллюстрации применения сегмента

;данных

SEGDT ENDS

;-------------------------------------

SEGCOD SEGMENT

ASSUME CS:SEGCOD,SS:SEGST,DS:SEGDT

BEGIN PROC FAR

PUSH DS ;сохранить содержимое DS в стеке

SUB AX,AX ;обнулить регистр AX

PUSH AX ;занести содержимое AX (то есть 0) в стек

MOV AX,SEGDT ;занести адрес начала сегмента данных в

;регистр AX

MOV DS,AX ;занести содержимое AX в регистр DS

;---------------------------------------------------------------

SUB BX,BX ;обнулить регистр BX

ADD BX,10 ;прибавить к содержимому BX 10

MOV CX,20 ;занести в CX 20

ADD BX,CX ;прибавить CX к BX

SUB BX,CX ;вычесть CX из BX

;-------------------------------------------------------------------

RET

BEGIN ENDP

SEGCOD ENDS

END BEGIN

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Приведите формат строки языка Ассемблер.

2. Что будет находиться в регистрах AX и BX после вы­полнения следующих команд:

MOV AX, 20

MOV BX, 10

ADD AX, BX

3. Перечислите требования для инициализации EXE-прог­раммы.

4. Что содержит сегментный регистр DS после загрузки EXE-программы в оперативную память.

5. Как осуществляется возврат из программы в DOS.

ТРАНСЛЯЦИЯ, КОМПОНОВКА И ОТЛАДКА ПРОГРАММЫ.

**Цель работы:** освоить процессы трансляции, компоновки и отладки программы, написанной на языке Ассемблер.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Процесс создания программы на языке Ассемблер включает в себя следующие этапы (рис.1).

Редактирование

Трансляция

Компоновка

Отладка

Любой текстовый редактор

Tasm.exe

Tlink.exe

Td.exe

Name.asm

Name.obj

Name.exe

(name.com)

Рис.1. Этапы разработки программы на языке Ассемблера.

Первым этапом создания Вашей программы является написа­ние ТЕКСТОВОГО файла на языке АССЕМБЛЕР (в дальнейшем этот файл будем называть ИСХОДНЫМ текстовым файлом). Выполняется этот этап при помощи любого **текстового** редактора, например, редактора Norton Commander. Для файлов, написанных на языке Ассемблер, принято расширение .asm.

На втором этапе Вы должны выполнить ТРАНСЛЯЦИЮ Вашего текстового файла в объектный код. В процессе трансляции текст Вашей программы преобразуется в машинный код, на этом этапе еще не привязанный к конкретным физическим адресам. Объектный код Вашей программы в результате трансляции будет помещен в OBJ-файл - файл с расширением .obj. Для трансля­ции используется программа-ТРАНСЛЯТОР tasm.exe. Для трансляции Вашей программы введите командную строку вида:

[путь]**tasm**[опции] sours[,object][,listing][,xref]

где sours, object, listing, xref - соответственно пути и имена исходного файла, OBJ-файла, файла-листинга и XREF-фай­ла (файла перекрестных ссылок).

При запуске tasm без параметров на экран будет выведен формат командной строки для tasm и опции транслятора.

Файл перекрестных ссылок при работе с небольшими программами обычно не требуется.

Например, для исходного файла name.asm командная строка может выглядеть так:

tasm name.asm, name.obj, name.lst

Предполагается, что исходный файл и файл tasm.exe располагаются в одном каталоге. Там же будут размещены и результирующие файлы. В противном случае необходимо указывать путь к файлам. При сохранении порядка перечисления файлов их расширения можно опустить. Если вы уверены в отсутствии ошибок, осуществить трансляцию можно командной строкой

tasm name

При этом будет сгенерирован только объектный файл.

Транслятор преобразует исходный файл в машинный код и выдает на экран сообщение об встретившихся ошибках. При трансляции выявляются **синтаксические** ошибки в Вашей програм­ме, например, задание недопустимой команды процессора, исполь­зование одинаковых или недопустимых имен, неправильный фор­мат директивы Ассемблера и т.п. На первых этапах работы с языком Ассемблер программист делает много синтаксических ошибок. Для большего удобства идентификации места возникновения ошибки можно воспользоваться файлом листинга. Файл-листинг является текстовым файлом и для его просмотра можно пользоваться соответствующими средствами MS DOS и Norton Commander.

Ниже показан листинг программы, приведенной в лабора­торной работе N1.

Turbo Assembler Version 4.0 01/07/03 21:20:02 Page 1

name.asm

1 0000 SEGST SEGMENT PARA STACK 'Stack'

2 0000 1E\*(??) DB 30 DUP(?)

3 001E SEGST ENDS

4 ;-------------------------------------

5 0000 SEGDT SEGMENT

6 0000 05 A DB 5 ;эти данные не используются в приведенной программе

7 0001 012C B DW 300 ; и приведены для иллюстрации применения сегмента

8 ;данных

9 0003 SEGDT ENDS

10 ;-------------------------------------

11 0000 SEGCOD SEGMENT

12 ASSUME CS:SEGCOD,SS:SEGST,DS:SEGDT

13 0000 BEGIN PROC FAR

14 0000 1E PUSH DS ;сохранить содержимое DS в стеке

15 0001 2B C0 SUB AX,AX ;обнулить регистр AX

16 0003 50 PUSH AX ;занести содержимое AX (то есть 0) в стек

17 0004 B8 0000s MOV AX,SEGDT ;занести адрес начала сегмента данных в

18 ;регистр AX

19 0007 8E D8 MOV DS,AX ;занести содержимое AX в регистр DS

20 0009 2B DB SUB BX,BX ;обнулить регистр BX

21 000B 83 C3 0A ADD BX,10 ;прибавить к содержимому BX 10

22 MOW CX,20 ;занести в CX 20

\*\*Error\*\* name.asm(22) Illegal instruction

23 000E 03 D9 ADD BX,CX ;прибавить CX к BX

24 0010 2B D9 SUB BX,CX ;вычесть CX из BX

25 0012 CB RET

26 0013 BEGIN ENDP

27 0013 SEGCOD ENDS

28 END BEGIN

Turbo Assembler Version 4.0 01/07/03 21:20:02 Page 2

Symbol Table

Symbol Name Type Value

??DATE Text "01/07/03"

??FILENAME Text "name "

??TIME Text "21:20:02"

??VERSION Number 0400

@CPU Text 0101H

@CURSEG Text SEGCOD

@FILENAME Text PR\_MET

@WORDSIZE Text 2

A Byte SEGDT:0000

B Word SEGDT:0001

BEGIN Far SEGCOD:0000

Groups & Segments Bit Size Align Combine Class

SEGCOD 16 0013 Para none

SEGDT 16 0003 Para none

SEGST 16 001E Para Stack STACK

Turbo Assembler Version 4.0 01/07/03 21:20:02 Page 3

Error Summary

\*\*Error\*\* name.asm(22) Illegal instruction

При набивке программы в двадцать второй строке была сделана ошибка: вместо MOV было набито MOW.

В правой части листинга расположен исходный текст Вашей программы. В левой части листинга приводится:

- в первой колонке - порядковый номер строки Вашей программы;

- во второй колонке - смещение относительно начала соот­ветствующего сегмента адреса команды, расположенной в данной строке;

- в третьей колонке - машинный код данной команды. Обратите внимание, что никакие директивы Ассемблера не генерируют машинный код.

В конце листинга приводится таблица идентификаторов. Первая часть содержит идентификаторы: так называемые предопределенные идентификаторы, имена полей данных в сегменте данных (A и B) и метки, назначенные командам в сегменте кодов (в нашем примере - BEGIN). Вторая часть таблицы содержит определенные в программе сег­менты с их размерами в байтах, выравниванием и классом.

В случае возникновения ошибки после строки с ошибкой ассемблерной программы будет выведено сообщение об ошибке и ее причине.

Для исправления ошибок, выявленных на этапе трансляции, Вам необходимо вернуться к первому этапу - текстовым редак­тором исправить ошибки и повторить трансляцию. Процесс повторяется до тех пор, пока не будут устранены все ошибки. Только в этом случае будет создан объектный файл.

Третьим этапом создания программы является КОМПОНОВКА. На этапе компоновки OBJ-файл, созданный при трансляции исходного файла, преобразуется в исполняемый EXE-файл. При этом завершается формирование адресов, которые были не опре­делены на этапе трансляции, объединяется, если это необходимо, несколько OBJ-фай­лов в единый EXE-файл.

Для компоновки программы введите командную строку вида:

[путь]tlink objfiles, [exefiles], [mapfiles], [libfiles], [defiles]

где objfiles и exefiles пути и имена соответственно объектного и ехе-файла. Остальные файлы для работы на нашем этапе не нужны.

При запуске tlink без параметров на экран будет выведен формат командной строки для tlink и опции компоновщика.

Например, для компоновки нашей программы командная строка будет выглядеть следующим образом:

tlink name.obj, name.exe

При этом предполагается, что все файлы находятся в одном каталоге.

При сохранении порядка следования файлов расширения можно опустить.

Если вы уверены в отсутствии ошибок, то можете ввести командную строку в виде:

tlink name

Типичной ошибкой на этом этапе является неправильное определение сегмента стека.

В случае возникновения ошибок компоновки необходимо вернуться на этап редактирования, исправить ошибки, опять странслировать программу и опять скомпоновать.

Когда устранены все ошибки, компоновщик сформирует ехе-файл – файл с расширением .ехе.

EXE-файл уже является выполняемой программой, которую можно запускать на выполнение командами MS DOS.

Для проверки правильности выполнения условий работы программы под управлением MS DOS запустите вашу ехе-программу на выполнение. Если программа не «висит», то условия инициализации ехе-программы соблюдены.

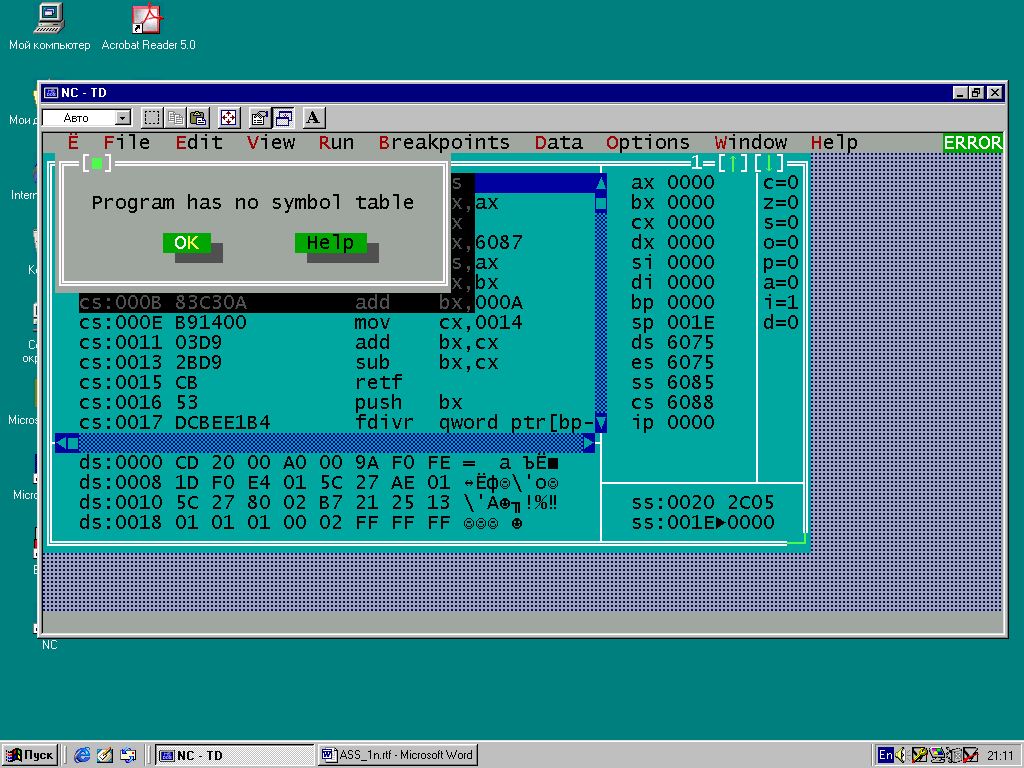
Может возникнуть необходимость создать выполняемый файл не в виде EXE-файла, а в виде COM-файла. Особенности COM-фай­ла и правила его создания будут рассмотрены в лабораторной работе N3.

Итак, Вы создали Вашу программу. Теперь ее необходимо ОТЛАДИТЬ, то есть проверить, правильно ли она работает. На этапе отладки выявляются логические ошибки в Вашей програм­ме, которые не могут быть выявлены на этапе трансляции и компоновки. Особенно сложно отладить программу, которая не "общается" с пользователем - не выводит информацию на экран. Для отладки ассемблерных программ служит специальная прог­рамма - отладчик. Одним из таких отладчиков является программа td.exe. Для запуска этой программы введите командную строку вида:

[путь]td.exe [путь]имя\_файла.exe

Имя\_файла - имя того файла, который Вы хотите отладить. Отлаживать можно только выполняемые файлы (EXE- и COM-фай­лы).

Вид экрана после запуска отладчика представлен на рис. 2



В

Б

Д

Г

А

Рис.2. Вид экрана после загрузки отладчика.

Нажмите «Enter» (так как символьную таблицу мы не создавали) и верхнее сообщение исчезнет.

Выйти из отладчика можно, нажав клавиши Alt-X или выбрав в верхнем меню пункт «файл», а в выпавшем меню пункт «Выход».

В окне «А» выведено содержимое кодового сегмента: в левой колонке смещение команд относительно сегментного регистра CS. Во второй колонке находятся машинные коды команд, в третьей - текст ассемблерной программы.

В окне «Б» находится содержимое регистров процессора, в окне «В» - значения флагов (содержимое флагового регистра), в окне «Г» - содержимое сегмента данных, в окне «Д» - содержимое стека.

Одним из основных действий отладки является ТРАССИРОВКА программы - ее пошаговое выполнение. При трассировке Ваша программа будет выполняться последовательно по одной команде. Для того, чтобы выполнить трассировку, Вам необходимо нажать клавишу «F7». При этом будет выполнена одна команда Вашей программы, начиная с первой. Стрелка перед машинным кодом команды указывает на команду, которая будет выполнена следующей. После выполнения каждой команды остальные окна показывают новое состояние процессора (регистров, флагов, оперативной памяти, стека). Это позволяет проследить за правильностью выполнения программы.

При обнаружении ошибок Вам нужно вернуться к первому этапу, т.е. исп­равить исходный текстовый файл, и затем последовательно пов­торить все описанные выше этапы.

Для более подробного ознакомления с работой программ tasm.exe, tlink.exe, td.exe, Вам нужно обратиться к соот­ветствующей документации.

ПОДГОТОВКА К РАБОТЕ

Изучить краткие теоретические сведения и рекомендованную литературу.

РАБОЧЕЕ ЗАДАНИЕ

Для исходного текстового файла, подготовленного во вре­мя выполнения лабораторной работы N 1, выполнить трансля­цию, компоновку и отладку программы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие существуют этапы создания с ассемблерной программы.
2. Как осуществить трансляцию программы, написанной на Ассемблере.
3. Какие файлы формируются на этапе трансляции.
4. Что собой представляет объектный файл.
5. Что собой представляет файл листинга.
6. Какие ошибки обнаруживаются на этапе трансляции. Как их устранить.
7. Как осуществляется компоновка программы.
8. Какой файл формируется на этапе компоновки.
9. Как проверить правильность выполнения условий работы программы под управлением MS DOS.
10. С помощью какой программы можно отладить вашу программу.
11. Какие окна Вы видите на экране после запуска программы – отладчика.
12. Что такое трассировка программы, как она выполняется.

COM-ФАЙЛЫ.

**Цель работы:** изучить назначение и особенности COM-фай­лов, научиться создавать ассемблерные программы в виде COM-файлов.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Выполняемую программу можно создавать не только в виде EXE-файла, но и в виде COM-файла. Примером командного файла может служить COMMAND.COM. Программы в виде EXE-файла и в виде COM-файла имеют следующие различия.

1. Размер программы. Программа в формате EXE может иметь в принципе любой размер. COM-программа ограничена размером одного сегмента и не превышает 64 Кбайт.
2. Форма представления файла на диске. COM-файл представлен на диске в виде абсолютного образа памяти, то есть так же, как и в оперативной памяти. EXE-файл имеет специальный заголовок размером 512 байт, где MS DOS хранит свою служебную информацию.
3. Сегмент стека. В EXE-программе необходимо определить сегмент стека. В COM-программе стек генерируется автомати­чески. По этому при создании ассемблерной программы, которая будет преобразована в COM-файл, сегмент стека должен быть опущен. Если максимальный размер сегмента (64К) достаточен для размещения и программы и стека, то MS DOS устанавливает регистр SP на конец сегмента (FFFE). Это вершина стека. Если программа занимает весь сегмент, т.е. места для стека недостаточно, то MS DOS устанавливает стек в конце памяти, доступной MS DOS.

4. Сегмент данных. В EXE-программе обычно определяется сегмент данных, и регистр DS инициализируется адресом этого сегмента. В COM-программе все данные должны находиться в ко­довом сегменте. **Таким образом, в COM-программе у Вас будет только один сегмент - сегмент кодов**.

5. Инициализация. В начале EXE-программы выполняется запись в стек значения регистра DS, нуля и инициализация ре­гистра DS. Так как сегменты стека и данных в COM-программе не определены, то эти шаги в ней отсутствуют. При загрузке COM-программы в оперативную память все сегментные регистры содержат адрес префикса программного сегмента (PSP) - 256-байтового (100h) блока, резервируемого MS DOS в оператив­ной памяти перед EXE- и COM-программой. Так как регистр CS также будет установлен на начало PSP, Вам надо установить пару регистров CS:IP на начало Вашей программы. Выполняется это директивой ORG 100h после директивы SEGMENT. Эта директива установит значение указателя команд IP (смещение относительно CS) 100h.

6. Создание COM-файла. Для создания COM-файла необхо­димо в процессе компоновки указать опцию /t.

Командная строка при компоновке будет иметь вид:

[путь]tlink/t objfiles,[comfiles],[mapfiles],[libfiles],[defiles]

При этом выполняемый файл будет по умолчанию иметь рас­ширение .com.

Преобразуем программу из лабораторной работы N1 в вид, необходимый для создания COM-программы.

SEGCOD SEGMENT PARA 'CODE'

ASSUME CS:SEGCOD, SS:SEGCOD, DS:SEGCOD, ES:SEGCOD

ORG 100H

BEGIN: JMP MAIN

A DB 5 ;эти данные не используются в приведенной программе

B DW 300 ; и приведены для иллюстрации применения данных

MAIN PROC NEAR

SUB BX,BX ;обнулить регистр BX

ADD BX,10 ;прибавить к содержимому BX 10

MOV CX,20 ;занести в CX 20

ADD BX,CX ;прибавить CX к BX

SUB BX,CX ;вычесть CX из BX

RET

MAIN ENDP

SEGCOD ENDS

END BEGIN

Обратите внимание, какие были произведены изменения.

1. Сегмент стека и сегмент данных отсутствуют.

2. Директива ASSUME инициализирует все сегментные ре­гистры адресом начала сегмента кодов.

3. Директива ORG 100h устанавливает указатель команд на начало Вашей программы (после PSP).

4. Команда JMP (безусловный переход) служит для обхода области данных.

5. После метки точки входа в Вашу программу (BEGIN) стоит двоеточие. Его назначение будет рассмотрено в лабора­торной работе N 5.

6. Процедура MAIN задана с параметром NEAR (внутрисегментная) так как ее начало уже не является точкой входа в программу.

ПОДГОТОВКА К РАБОТЕ

1. Изучить краткие теоретические сведения и рекомендо­ванную литературу.

2. Преобразовать исходный текст программы из лабораторной работы N 1 для создания COM-файла.

РАБОЧЕЕ ЗАДАНИЕ

Набить подготовленную программу, странслировать ее, скомпоновать, преобразовать в COM-файл, отладить.

Запустить программу на выполнение. Проверить, осущест­вляется ли возврат в MS DOS.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Сколько сегментов в Вашей программе.
2. Где в Вашей программе располагается стек.
3. Какое значение содержит регистр SP.
4. Зачем необходима директива ORG 100h.
5. В чем разница в представлении exe- и com-программы на диске.
6. Сформулируйте основные отличия exe- и com-программы.

ОПРЕДЕЛЕНИЕ ДАННЫХ.

**Цель работы:** ознакомиться с методами определения конс­тант и рабочих областей в ассемблерной программе.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В лабораторной работе N 1 мы задавали необходимые нам данные непосредственно в команде. Например, мы писали

MOV AX,25

В этом случае число 25 становится частью объектного кода команды.

Такой способ представления данных целесообразен, если заданное число (в нашем случае 25) встречается в Вашей прог­рамме один раз. Если же заданное число встречается в прог­рамме многократно, то повторное задание его в различных командах приведет к неоправданному расходу оперативной памя­ти. Память под Ваше число будет выделяться каждый раз, когда Вы его указываете. Кроме того, если возникнет необходимость изменить заданное число, то Вам придется отыскивать его по всей программе. Здесь велика вероятность ошибки. Более целесообразным является запись нужного числа в ячейку оперативной памяти, присвоение этой ячейке символьного имени, а затем обращение к заданному числу по имени. В этом случае память под число будет выделяться один раз и при необходимости изменения этого числа сделать замену придется тоже только в одном месте.

В лабораторной работе N 1 мы использовали для хранения результатов вычислений (промежуточных и окончатель­ных) регистры общего назначения. Такой способ является наи­более быстродействующим. Однако в регистрах можно хранить весьма ограниченный объем информации. Вместо использования регистров для хранения результатов можно выделять в оперативной памяти так называемые рабочие области - определенное количество ячеек оперативной памяти, которым можно присвоить имена и в командах обращаться к ним по именам.

В Ассемблере для описания исходных данных и выделения рабочих об­ластей в оперативной памяти служат директивы ОПРЕДЕЛЕНИЯ ДАННЫХ.

С помощью этих директив Вы можете задавать КОНСТАНТЫ (числа, значения которых определяются в исходном тексте Ва­шей программы) и ПЕРЕМЕННЫЕ (рабочие области определенной длины, значение которых может изменяться в процессе работы Вашей программы).

Основной формат определения данных

[имя] Dn выражение

Имя элемента данных не обязательно, однако, если Вы хо­тите в программе обращаться к этому элементу, Вы должны это имя указать. Имя создается по правилам написания символьных имен (идентификаторов), которые были приведены в лабораторной работе N 1 при описании меток.

Директивы определения данных (Dn) могут использоваться следующие: DB (байт), DW (слово), DD (двойное слово), DQ (учетверенное слово) и DT (десять байт).

В поле «выражение» может находиться:

1. Константа.

Например: ABC DB 25

2. Последовательность констант.

Например:

BCD DB 1,2,3,4,5

Константы в последовательности должны разделяться запя­тыми. Количество констант в последовательности ограничивает­ся длиной строки. Имя (в нашем примере BCD) указывает на пер­вую константу из последовательности (в нашем примере – «1»). Ко второй константе («2») можно обращаться по имени BCD+1 (так как первая константа занимает один байт), к третей - BCD+2 и так далее. Если бы мы использовали в этом примере директиву DW, то к BCD необходимо было прибавлять соответственно 2, 4 и так далее.

3. В качестве выражения может стоять знак вопроса (?) для неопределенного значения переменой:

CDE DB (?)

В этом случае значение переменной CDE не присваивается.

4. Выражение допускает повторение константы или перемен­ной в следующем виде:

[имя] Dn число\_повторений DUP (выражение)

например,

DEF DB 10 DUP(?)

Оператор повторения (DUP) может быть вложенным, например:

EFH DB 10 DUP(4 DUP(2))

В этом случае будут сгенерированы вначале 4 копии конс­танты 2, а затем это значение повторится 10 раз. В результа­те будет сформирована последовательность из сорока двоек.

Различайте инициализированные и неинициализированные данные. В случае неинициализированных данных (в поле «выражение» стоит знак вопроса – «?») в оперативной памяти выделяется ячейка соответствующего размера и никакое значение ей не присваивается. Для инициализированных данных (все остальные значения в поле «выражение») также выделяется ячейка памяти заданного размера и ей присваивается указанное значение.

Константа, задаваемая в поле «выражение» может предс­тавлять собой символьную строку или числовую константу.

СИМВОЛЬНАЯ СТРОКА используется для представления текс­та. Она заключается в апострофы или кавычки, например

'Лабораторная работа N 4'

или

«Лабораторная работа N 4»

Ассемблер переводит каждый символ символьной строки в соответствующий ASCII-код. Символьная строка определяется директивой DB, в которой указываются символы в нормальной последовательности слева направо.

NAME1 DB 'Лабораторная работа N 5'

ЧИСЛОВЫЕ КОНСТАНТЫ применяются для арифметических зна­чений и для адресов памяти. Числовая константа указывается в директиве определения данных непосредственно, без кавычек или апострофов. Ассемблер преобразует все числовые константы в двоичные (в листинге приводится шестнадцатиричное представление двоичных констант). При работе с отладчиком обратите внимание, что числовые константы размером больше байта записываются в оперативную память в обратной последовательности байтов - справа налево (по более младшему адресу – более младший байт). Так например, шестнадцатиричную константу размером в слово 1234h вы увидите записанной в памяти как 34 12.

Числовые константы могут задаваться в следующих форма­тах.

ДЕСЯТИЧНЫЙ ФОРМАТ допускает десятичные цифры от 0 до 9 и обозначается последней буквой D. Десятич­ный формат принимается по умолчанию, то есть если никакая буква после числа не указана, то это десятичный формат. Де­сятичный формат преобразуется Ассемблером в двоичный, кото­рый в листинге записывается в шестнадцатиричном формате.

Например,

NAME2 DB 12D

или

NAME2 DB 12

ШЕСТНАДЦАТИРИЧНЫЙ ФОРМАТ допускает цифры от 0 до 9 и буквы от А до F и обозначается в конце буквой H. Так как Ассемблер полагает, что с буквы начинаются символьные имена, то первым знаком шестнадцати­ричной константы должна быть цифра от 0 до 9. Если Вам нуж­но, например, задать константу A0C1H, то можно записать ее в виде 0A0C1H.

NAME3 DB 0CH

ДВОИЧНЫЙ ФОРМАТ допускает цифры 0 и 1 и обозначается в конце буквой B.

NAME4 DB 11001010B

ВОСЬМИРИЧНЫЙ ФОРМАТ допускает цифры от 0 до 7 и обозна­чается последней буквой Q или O. В настоящее время восьми­ричный формат используется редко.

NAME5 DB 10Q

Существует еще десятичный формат с плавающей точкой, но он используется в основном для задания данных для сопроцес­сора.

При записи шестнадцатиричных констант и букв, обозначающих формат, можно использовать как строчные, так и прописные буквы.

При задании символьных строк и числовых констант следу­ет помнить, что, например, '14' и 14 представляют собой две разные константы. Первая преобразуется в два байта ASCII-ко­да, соответствующие символам 1 и 4. Вторая преобразуется в двоичное число 1110, в шестнадцатиричном представлении 0E.

ДИРЕКТИВА ОПРЕДЕЛЕНИЯ БАЙТА (DB) - определяет байт. Сим­вольное выражение в директиве DB может содержать символьную строку любой длины (ограничивается только длиной ассемблер­ной строки). Числовая константа в директиве DB может содер­жать одну или более однобайтовых констант. Один байт в лис­тинге представляется двумя шестнадцатиричными цифрами. Наи­большее положительное шестнадцатиричное число в одном байте - 7F, все большие числа от 80 до FF представляют отрицатель­ные значения. В десятичном формате эти пределы выражаются числами +127 и -128.

Примеры использования директивы определения байта:

NAME6 DB ?

NAME7 DB 'Задание N 5'

NAME8 DB 12H

NAME9 DB 112

NAME10 DB 01101100B

NAME11 DB 10,11,12,13,14

NAME12 DB 2 DUP(10)

ДИРЕКТИВА ОПРЕДЕЛЕНИЯ СЛОВА (DW) определяет константы и переменные, имеющие длину в одно слово - два байта. Символь­ное выражение в DW ограничено двумя символами. Числовое выражение в DW может содержать одну или более двухбайтовых констант.

Два байта представляются четырьмя шестнадцатиричными цифра­ми. Наибольшее положительное шестнадцатиричное число в двух байтах - 7FFF, числа от 8000 до FFFF представляют отрица­тельные значения. В десятичном коде эти пределы выражаются числами +32767 и -32768.

В поле «выражение» директивы DW мо­жет быть подставлено имя константы или переменной, определен­ной ранее, например:

NAME13 DW NAME8

В этом случае под именем NAME13 будет сгенериро­вана константа, равная адресу (смещению) константы NAME8. Это так называемая адресная константа.

Примеры использования директивы определения слова:

NAME14 DW ?

NAME15 DW 23,64

NAME16 DW '12'

NAME17 DW NAME15

NAME18 DW 0110111100001010B

NAME19 DW 1F0BH

ДИРЕКТИВА ОПРЕДЕЛЕНИЯ ДВОЙНОГО СЛОВА (DD) определяет элементы, которые имеют в длину два слова - четыре байта. Размер числовых констант, используемых в выражении не должен превышать четырех байт (восемь шестнадцатиричных цифр). Наи­большее положительное шестнадцатиричное число, которое может быть использовано в директиве DD, - 7FFFFFFF. Диапазон отри­цательных чисел - от 80000000 до FFFFFFFF. В десятичном коде диапазон допустимых чисел в этой директиве - от +2147483647 до -2147483648. Символьное выражение в директиве DD ограничено двумя символами. Ассемблер преобразует их в ASCII-код и вы­равнивает их справа как показано в приведенной ниже строке листинга

0010 00 00 32 33 NAME20 DD '23'

Примеры использования директивы определения двойного слова:

NAME21 DD 1F00FFFFH

NAME22 DD 'AB'

NAME23 DD ?

NAME24 DD 45,47,49

NAME25 DD NAME23-NAME21

ДИРЕКТИВА ОПРЕДЕЛЕНИЯ УЧЕТВЕРЕННОГО СЛОВА (DQ) опреде­ляет элементы, имеющие длину четыре слова (восемь байт). Ис­пользуемые в директиве числовые константы не должны превы­шать размер восемь байт. Обработка Ас­семблером символьных строк в директиве DQ аналогична обра­ботке в директивах DW и DD.

Примеры использования директивы определения учетверен­ного слова:

NAME26 DQ 245H

NAME27 DQ 100

ДИРЕКТИВА ОПРЕДЕЛЕНИЯ ДЕСЯТИ БАЙТ (DT) определяет эле­менты данных, имеющие длину десять байт. Предназначена для определения так называемых "упакованных десятич­ных" числовых значений, в которых в каждом байте каждая тэтрада содержит двоично-десятичный код соответствующей десятичной цифры. Причем код более старшей цифры располагается в более старшей тэтраде.

Примеры использования директивы определения десяти байт:

NAME28 DT ?

NAME29 DT '23'

Более подробно о возможных значениях поля «выражение» для каждой директивы обращайтесь к документации на соответствующую версию языка Ассемблер.

НЕПОСРЕДСТВЕННЫЕ ОПЕРАНДЫ.

Константа, задаваемая непосредственно в команде, называется непосредственным операндом. Значение непосредственно­го операнда входит в генерируемый для этой команды объектный код. Допустимая длина непосредственного операнда зависит от длины пер­вого операнда в команде. Например, в команде

MOV AL, XX

непосредственный операнд, который мы обозначили как ХХ, дол­жен иметь длину один байт, так регистр AL является однобай­товым регистром.

В команде

MOV AX, XX

непосредственный операнд может иметь длину слово (два бай­та).

При несоблюдении этого правила транслятор сгенерирует сообщение об ошибке.

ДИРЕКТИВА EQU.

Директива EQU присваивает определенное значение символьному имени (идентификатору). Например, если Вы задали

KOL EQU 10

то каждый раз, когда транслятор в тексте Вашей программы встретит идентификатор KOL, он вместо него подставит значение 10. Так транслятор преобразует директиву

NAME30 DW KOL DUP(?)

в директиву

NAME30 DW 10 DUP(?)

Имя, связанное с некоторым значением при помощи дирек­тивы EQU, может использоваться в качестве операнда в коман­де. Например,

MOV AX, KOL

Транслятор заменит имя KOL на значение 10, создавая не­посредственный операнд, как если бы было задано

MOV AX, 10

В отличие от директив определения данных директива EQU не выделяет место под константу в оперативной памяти, а только указывает транслятору на необходимость замены идентификатора, стоящего в левой части директивы, на значение, стоящее в правой части директивы. С помощью директивы EQU можно присваивать идентификатору как символьные, так и числовые значения.

ПОДГОТОВКА К РАБОТЕ

1. Изучить краткие теоретические сведения и рекомендованную литературу.

2. Написать текст программы, соответствующий вашему ва­рианту. Ваша программа должна содержать только сегмент данных и заканчиваться директивой END.

РАБОЧЕЕ ЗАДАНИЕ

1. Набить подготовленную программу, странслировать ее. При трансляции создать листинг. Объектный файл не создавать. Изучить полученный листинг. Проверить, как в объектном коде представляются различные константы.
2. Изменить текст лабораторной работы N 1, заменив непосредственные операнды, используемые в программе, на константы, заданные в сегменте данных, а в программе обратившись к ним по именам. Странслировать измененную программу, скомпоновать и проверить отладчиком.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем разница между инициализированными и неинициализированными данными.
2. В чем отличие представления в памяти двух констант: NAMEA DB 42 и NAMEB DB '42'.
3. Какое значение подставится вместо имени NAMED в вы­ражении

NAMEC DW 20

NAMED DW NAMEC

1. В сегменте данных приведена запись

NAMEE DW 1, 2, 3, 4

Как обратиться в программе ко второй константе этой строки.

5. В какой системе счисления по умолчанию задаются числовые константы.

ОРГАНИЗАЦИЯ ВЕТВЯЩИХСЯ ПРОЦЕССОВ

**Цель работы**: ознакомиться с механизмом организации вет­вящихся процессов и командами Ассемблера, используемыми для этой цели.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В лабораторной работе N 1 Вы написали программу, ко­манды которой выполняются последовательно в том порядке, в котором Вы их записали. Такая программа представляет собой **линейный** процесс. В настоящей работе мы рассмотрим, как организовать ВЕТВЯЩИЙСЯ процесс - программу, порядок выполнения команд которой зависит от результатов ее работы. Примером ветвящегося процесса может служить простейший алгоритм:

1. Заданы два числа A и B.

2. Если A>B, вычислить C=A+B.

3. В противном случае, то есть, если A<=B, вычислить C=B-A.

Для организации ветвящихся процессов Вам необходимы ко­манды, которые бы могли передавать управление по адресу ко­манды, не находящейся непосредственно за выполняемой коман­дой. Передача управления может осуществляться вперед для вы­полнения новой группы команд или назад для повторения уже выполненных команд.

Имеются три типа переходов (передачи управления).

Переход типа SHORT (короткий) передает управления в пределах изме­нения адреса на один байт (-128 - +127).

Переход типа NEAR (ближний или внутрисегментный) передает управление в пределах одного сегмента.

Переход типа FAR (дальний или межсегментный) передает управление в другой сегмент.

При переходах SHORT и NEAR изменяется только содержимое указателя команд IP. При FAR-переходе изменяется содержимое указателя команд IP и кодового сегментного регистра CS.

Рассмотрим виды переходов, используемые для передачи уп­равления в программе.

БЕЗУСЛОВНЫЙ ПЕРЕХОД.

Безусловный переход осуществляет передачу управления при любых обстоятельствах.

Для выполнения безусловного перехода в Ассемблере испо­льзуется команда JMP.

Например, приведенная ниже программа осуществляет сло­жение чисел натурального ряда.

codseg segment

assume cs:codseg, ds:codseg, ss: codseg

org 100h

main proc near

sub ax, ax ; в ax накапливается сумма

mov bx, 1 ; в bx формируются числа натурального ряда

a20: add ax, bx

add bx, 1

jmp a20

ret

main endp

codseg ends

end main

Команда JMB A20 передает управление команде с меткой A20:. Двоеточие после метки означает, что тип метки NEAR, то есть управление на эту метку передается внутри сегмента кодов. Обратите внима­ние, при указании метки в команде JMB двоеточие после имени метки не ставится.

Метку можно записывать в одной строке с командой

A20: ADD AX, BX

или в отдельной строке

A20:

ADD AX, BX

Данный цикл не имеет выхода и приводит к бесконечному выполнению - такие циклы обычно не используются.

Рассмотрим листинг данной программы.

Turbo Assembler Version 4.0 08/07/03 21:23:38 Page 1

v\_pr.asm

1 0000 codseg segment

2 assume cs:codseg, ds:codseg, ss: codseg

3 org 100h

4 0100 main proc near

5 0100 2B C0 sub ax, ax ; в ax накапливается сумма

6 0102 BB 0001 mov bx, 1 ; в bx формируются числа натурального ряда

7 0105 03 C3 a20: add ax, bx

8 0107 83 C3 01 add bx, 1

9 010A EB F9 jmp a20

10 010C C3 ret

11 010D main endp

12 010D codseg ends

1. end main

В нашем примере операнд команды JMB (метка A20) соот­ветствует семи байтам от команды, следующей за JMP, до команди, на которую передается управление, в чем можно убедиться по объектному коду команды EBF9. Здесь EB - машин­ный код для короткого перехода JMP , а F9 - отрицательное значение смещения (-7). Команда JMP прибавляет F9 к командному указателю (IP), который содержит адрес команды, следую­щей за JMP (010С). В результате сложения получается адрес перехода (0105). Операнд в команде JMP для перехода вперед будет иметь положительное значение.

Команда JMB для перехода в пределах от -128 до +127 байт имеет тип SHORT. Ассемблер генерирует в этом случае од­нобайтовый операнд в пределах от 00 до FF. Команда JMP пре­восходящая эти пределы, получает тип NEAR, для которого гене­рируется другой машинный код и двухбайтовый операнд. При передаче управления назад

A50: ....

....

JMP A50

транслятор, дойдя до команды JMP, уже знает «расстояние» между командой JMP и меткой A50 и сгенерирует правильный код команды (тип SHORT или NEAR) в зависимости от реального расстояния.

При передаче управления вперед

JMP A90

........

A90:

транслятор, дойдя до команды JMP, не знает типа перехода (NEAR или SHORT) и автоматически генерирует команду JMP для перехода типа NEAR. Для того, чтобы указать транслятору на необходимость генерации команды для короткого перехода (SHORT), следует использовать оператор SHORT:

JMP SHORT A90

.....

A90:

Это позволит сгенерировать более оптимальный код команды.

УСЛОВНЫЙ ПЕРЕХОД

Условный переход представляет собой двухступенчатый про­цесс: сначала проверяется условие, а затем осуществляется пе­реход, если условие выполняется, или продолжение работы, если условие не выполняется.

КОМАНДЫ УСЛОВНОГО ПЕРЕХОДА осуществляют передачу управ­ления в зависимости от результата выполнения предыдущей ко­манды.

Команды условного перехода используют единственный опе­ранд, содержащий адрес (метку), на который должен быть осуще­ствлен переход. Расстояние от команды перехода до заданного адреса должно быть меньше 128 байт (переход типа SHORT).

Команды условного перехода используют состояние одного или нескольких флагов (содержимое флагового регистра) в каче­стве условий перехода. Таким образом, любая команда, которая устанавливает флаг по определенному условию, может быть ко­мандой проверки условия. Чаще других для этой цели использу­ются команды CMP и TEST. Командой перехода может быть любая из 31 команды условного перехода.

ФЛАГОВЫЙ РЕГИСТР. Флаговый регистр хранит в виде бито­вых полей некоторые результаты выполнения операций (признаки нулевого, отрицательного результата, переполнения разрядной сетки и др.). Флаги сохраняют свое значение до тех пор, пока другая команда не изменит их.

Формат флагового регистра

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

X X X X OF DF IF TF SF ZF X AF X PF X CF

Х отмечены неиспользуемые биты.

CF (Carry Flag) - флаг переноса. Содержит значения пе­реноса (0 или 1) из старшего разряда при арифметических операциях и операциях сдвига и циклического сдви­га.

PF (Parity Flag) - флаг четности. Проверяет младшие 8 битов результата операций над данными. Нечетное число битов приводит к установке этого флага в 0, четное - в 1. Не следует путать этот флаг с битом контроля на четность.

AF (Auxiliary Carry Flag) - вспомогательный флаг пере­носа. Устанавливается в 1, если арифметическая операция при­водит к переносу из четвертого справа бита (бита номер 3) в ре­гистровой однобайтовой команде. Данный флаг имеет отношение к арифметическим операция над числами в ASCII-формате и упакованном десятичном формате.

ZF (Zero Flag) - флаг нуля. Устанавливается в качестве результата арифметических команд и команд сравнения. При ну­левом результате - 1, при ненулевом - 0.

SF (Sign Flag) - флаг знака. Устанавливается в соот­ветствии со знаком результата (старшего бита) после арифме­тических операций. При положительном результате - 0, при от­рицательном - 1.

TF (Trap Flag) - флаг трассировки. Если этот флаг уста­новлен в единичное состояние, то процессор переходит в режим пошагового выполнения команд.

IF (Interrupt Flag) - флаг прерывания. При нулевом сос­тоянии этого флага прерывания запрещены, при единичном - разрешены.

DF (Direction Flag) - используется в строковых операци­ях для определения направления обработки строки.

OF (Overflow Flag) - флаг переполнения. Фиксирует ариф­метическое переполнение, т.е. перенос в(из) старший (знако­вый) разряд при знаковых арифметических операциях.

Команда CMP сравнивает два операнда. Операнды в процессе выполнения команды не изменяются. Команда исполь­зуется для проверки соотношений равно, не равно, больше, ме­ньше, больше или равно, меньше или равно. Воздействует на флаги AF, CF, OF, PF, SF, ZF. По своей сути команда CMP сов­падает с командой SUB, за исключением того, что не изменяется операнд-приемник. Нет необходимости проверять все флаги, устанавливаемые командой CMP в отдельности. В следующем при­мере проверяется, содержит ли регистр BX нулевое значение:

CMP BX, 00 ;сравнение BX с 0

JZ B50 ;переход на B50, если BX = 0

... ;действия при BX ≠ 0

...

B50: ... ;точка перехода: действия при BX=0

Если BX содержит нулевое значение, команда CMP устанавливает флаг нуля ZF в единичное состояние и, возможно изме­няет (или нет) другие флаги. Команда JZ (переход, если нуль) проверяет только флаг ZF. При единичном значении ZF, обозна­чающем нулевой результат, команда передает управление на ад­рес, указанный в ее операнде, т.е. на метку B50.

Рассматривая назначение команд условного перехода, сле­дует пояснить характер их использования. Типы данных, над ко­торыми выполняются арифметические операции и операции срав­нения, определяют, какими командами пользоваться: беззнако­выми или знаковыми. Беззнаковые данные используют все биты как биты данных. Характерным примером являются символьные строки и натуральные числа. В знаковых данных самый левый бит представляет собой знак, причем, если его значение равно нулю, то число положительное, а если равно единице - отрица­тельное.

В качестве примера предположим, что регистр AL содержит 11000110, а BL - 00010110. Команда

CMP AL, BL

сравнивает содержимое регистров AL и BL. Если данные рассмат­риваются как знаковые, то значение в BL больше, если как беззнаковые, то значение в AL больше.

КОМАНДЫ ПЕРЕХОДА ДЛЯ БЕЗЗНАКОВЫХ ДАННЫХ.

Мнемоника Описание Проверяемые флаги

JE/JZ Переход, если равно/нуль ZF

JNE/JNZ Переход, если не равно/не нуль ZF

JA/JNBE Переход, если выше/не ниже или равно ZF, CF

JAE/JNB Переход, если выше или равно/ не ниже CF

JB/JNAE Переход, если ниже/не выше или равно CF

JBE/JNA Переход, если ниже или равно/ не выше CF, AF

Любую проверку можно кодировать одним из двух мнемонических кодов. Например, JB и JNAE генерирует один и тот же объектный код, хотя положительную проверку JB легче понять, чем отрицательную JNAE.

КОМАНДЫ ПЕРЕХОДА ДЛЯ ЗНАКОВЫХ ДАННЫХ.

Мнемоника Описание Проверяемые флаги

JE/JZ Переход, если равно/нуль ZF

JNE/JNZ Переход, если не равно/не нуль ZF

JG/JNLE Переход, если больше/не меньше или равно ZF,SF,OF

JGE/JNL Переход, если больше или равно/ не меньше SF,OF

JL/JNGE Переход, если меньше/не больше или равно SF,OF

JLE/JNG Переход, если меньше или равно/ не больше ZF,SF,OF

Команды перехода для условия равно или ноль (JE/JZ) и не равно или не ноль (JNE/JNZ) присутствуют в обоих списках для знаковых и беззнаковых данных. Состояние равно/нуль про­исходит вне зависимости от наличия знака.

СПЕЦИАЛЬНЫЕ АРИФМЕТИЧЕСКИЕ ПРОВЕРКИ

Мнемоника Описание Проверяемые флаги

JS Переход, если есть знак (отрицательно) SF

JNS Переход, если нет знака (положительно) SF

JC Переход, если есть перенос (аналогично JB) CF

JNC Переход, если нет переноса CF

JO Переход, если есть переполнение OF

JNO Переход, если нет переполнения OF

JP/JPE Переход, если паритет четный PF

JNP/JPO Переход, если паритет нечетный PF

Используя команды условного перехода, можно организо­вать ветвящийся процесс, реализующий алгоритм, приведенный в примере в начале данной лабораторной работы.

1 0000 codesg segment para 'code'

2 assume cs:codesg,ds:codesg,ss:codesg

3 org 100h

4 0100 EB 07 90 main: jmp begin

5 0103 0006 a dw 6

6 0105 0005 b dw 5

7 0107 ???? c dw ?

8 ;пусть переменыые a,b и c находятся соответ-

9 ;ственно в переменных в памяти A, B и C

10 0109 begin proc near

11 0109 A1 0103r MOV AX,A ; занесем A в регистр AX

12 010C 3B 06 0105r CMP AX,B ; сравним AX с B

13 0110 7F 09 JG L1 ; если AX больше B

14 ; перейдем на L1

15 0112 29 06 0105r SUB B,AX ; иначе вычтем из B A

16 0116 A1 0105r MOV AX,B

17 0119 EB 04 JMP SHORT L2; перейдем на метку L2

18 011B 03 06 0105r L1: ADD AX,B ; сложим A и B

19 011F A3 0107r L2: MOV C,AX ; результат занесем в С

20 0122 C3 RET

21 0123 begin endp

22 0123 codesg ends

1. end main

Программу можно минимизировать, например, если вместо операции сравнения сразу выполнить операцию вычитания, кото­рая устанавливает те же флаги.

ПОДГОТОВКА К РАБОТЕ

1. Изучить краткие теоретические сведения и рекомендо­ванную литературу.

1. Написать текст программы в соответствии с заданным ва­риантом.

РАБОЧЕЕ ЗАДАНИЕ

Набить подготовленную программу, странслировать ее, скомпоновать и от­ладить.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. На какое максимальное количество байт можно осуще­ствить переход командами JMP и командами условного перехода.

2. В чем разница организации ветвящихся процессов для знаковых и беззнаковых данных.

3. На какие флаги воздействуют следующие события:

- произошел перенос;

- результат отрицательный;

- произошло переполнение;

* результат нулевой.

ОРГАНИЗАЦИЯ ЦИКЛИЧЕСКИХ ПРОЦЕССОВ

**Цель работы**: ознакомиться с механизмом организации цик­лических процессов и командами Ассемблера, используемыми для этой цели.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

При создании Вашей программы может возникнуть необходи­мость повторить некоторую последовательность команд заданное количество раз. Чтобы не переписывать блок одинаковых команд несколько раз, организуется ЦИКЛИЧЕСКИЙ процес­с, при котором после завершения выполнения блока повторяющихся команд управление передается на его начало до тех пор, пока блок команд не будет выполнен заданное количество раз.

Для организации циклических процессов в Ассемблере су­ществует команда LOOP. Кроме того, циклы можно создавать, используя команды условных переходов. При использовании команды LOOP количество циклов (количество повторений участка Вашей программы) задается в регистре CX. В каждом цикле команда LOOP автоматически уменьшает содержимое CX на 1. Пока значение CX не равно ну­лю, управление передается по адресу, указанному в операнде команды LOOP. Когда CX станет равным нулю, управление пере­дается на следующую за LOOP команду.

В лабораторной работе N 5 при описании команды JMP был приведен пример бесконечно­го циклического процесса. Реально бесконечные циклы не ис­пользуются. Модифицируем эту программу для вычисления суммы десяти чисел натурального ряда.

Пример 1.

1 0000 CODESG SEGMENT PARA 'Code'

2 ASSUME CS:CODESG, SS:CODESG, DS:CODESG

3 ORG 100H

4 0100 BEGIN PROC NEAR

5 0100 B8 0001 SUB AX, AX

6 0103 BB 0001 SUB BX, BX

7 0106 B9 000A MOV CX, 10

8 0109 A20:

9 0109 40 INC BX ; ВX+1

10 010A 03 D8 ADD AX, BX

11 010C E2 FB LOOP A20

12 010E C3 RET

13 010F BEGIN ENDP

14 010F CODESG ENDS

1. END BEGIN

Программа, приведенная в примере, выполняет участок программы находящийся между меткой A20 и командой LOOP 10 раз (число 10 задано ДО начала первого цикла в регистре CX).

Расстояние от команды LOOP до метки, заданной в качест­ве ее операнда, не должно превышать -128 - +127 байт (пере­ход типа SHORT).

Дополнительно существуют две разновидности команды LOOP: LOOPE (LOOPZ) и LOOPNE (LOOPNZ). Обе команды также уменьшают значение CX на 1. Команда LOOPE (LOOPZ) передает управление по адресу операнда, если регистр СХ имеет ненулевое значение и флаг нуля установлен (ZF=1). LOOPNE(LOOPNZ) передает управление по адресу операнда, если регистр CX имеет ненуле­вое значение и флаг нуля сброшен (ZF=0).

В примере 2 показано, как организовать циклический про­цесс с помощью команды условного перехода.

Пример 2.

1 0000 CODESG SEGMENT PARA 'Code'

2 ASSUME CS:CODESG,SS:CODESG,DS:CODESG

3 ORG 100H

4 0100 BEGIN PROC NEAR

5 0100 B8 0001 SUB AX, AX

6 0103 BB 0001 SUB BX, BX

7 0106 BA 000A MOV DX, 10

8 0109 A20:

9 0109 40 INC BX

10 010A 03 D8 ADD AX, BX

11 010C 4A DEC DX ; DX - 1

12 010D 75 FA JNZ A20

13 010F C3 RET

14 0110 BEGIN ENDP

15 0110 CODESG ENDS

1. END BEGIN

Для реализации задания данной лабораторной работы Вам необходимо научиться ум­ножать и делить константу на степень числа 2. Выполнять та­кие операции целесообразно с помощью команд сдвига.

СДВИГ И ЦИКЛИЧЕСКИЙ СДВИГ РАЗРЯДОВ

Ассемблер имеет команды для сдвига и циклического сдвига разрядов в байте или слове. Разряды могут сдвигаться вправо (в сторону младших разрядов) или влево (в сторону старших раз­рядов). Значение выдвинутого за пределы операнда разряда для всех команд сдвига попа­дает во флаг переноса.

Команды сдвига распадаются на две группы. *Логические* команды сдвигают операнд, не считаясь с его знаком; они используются для действий над числами без знака или над нечисловыми значениями. *Арифметические* команды сохраняют старший знаковый бит операнда, они используются для действий над числами со знаком.

На следующем рисунке показано действие 8 команд сдвига и циклического сдвига для 8-разрядных операндов.

SHL (логический беззнаковый сдвиг влево)

7 0

CF 0

SHR (логический беззнаковый сдвиг вправо)

7 0

0 CF

SAL (арифметический сдвиг влево)

7 0

CF 0

SAR (арифметический сдвиг вправо)

7 0

CF

ROL (циклический сдвиг влево)

7 0

CF

ROR (циклический сдвиг вправо)

7 0

CF

RCL (циклический сдвиг влево через флаг CF или с переносом)

7 0

CF

RCR (циклический сдвиг вправо через флаг CF или с переносом)

7 0

CF

Команды SHL и SAL отличаются только воздействием на флаги.

В операнде-приемнике команд сдвига находится значение, подлежащее сдвигу. После выполнения команды в нем же будет находиться результат. В операнде-источнике должно находиться число, на которое нужно сдвинуть разряды. Данное число может быть задано как непосредственное значение "1" или как содер­жимое регистра CL.

Команда SAR сохраняет знак операнда, репродуцируя его при выполнении сдвига. Команда SAL не сохраняет знак, но за­носит "1" во флаг переполнения OF в случае изменения знака операнда.

Пусть AL содержит 0B4h, а CF=1

; AL=1011 0100b

SAL AL,1 ; AL=0110 1000 CF=1

SAR AL,1 ; AL=1101 1010 CF=0

SHL AL,1 ; AL=0110 1000 CF=1

SHR AL,1 ; AL=0101 1010 CF=0

ROL AL,1 ; AL=0110 1001 CF=1

ROR AL,1 ; AL=0101 1010 CF=0

RCL AL,1 ; AL=0110 1001 CF=1

RCR AL,1 ; AL=1101 1010 CF=0

Сдвиг вправо на 1 эквивалентен делению на 2, сдвиг влево на 1 - умножению на 2. Данная особенность может быть ис­пользована для умножения и деления на часто встречающиеся константы, например, на 2n. При двойном сдвиге влево происходит умножение на 4, при тройном - на 8 и так далее.

SHR используется для деления беззнаковых чисел, SAR - для знаковых. Умножение с помощью сдвига является одинаковым для знаковых и беззнаковых чисел, поэтому можно использовать как SAL, так и SHL.

MOV CL,2

SHL AX,CL ;умножить число без знака на 4

SAL AX,CL ;умножить число со знаком на 4

SHR AX,CL ;разделить число без знака на 4

SAR AX,CL ;разделить число со знаком на 4

Каждая из этих команд с учетом загрузки CL выполняется в 6-8 раз быстрее соответствующей команды умножения или деления.

Когда необходимо сдвинуть значение, которое слишком ве­лико для регистра, то можно сдвигать каждую часть отдельно, передавая сдвинутые разряды через флаг переноса. Для передачи переноса из первого регистра во второй должны использоваться команды RCR или RCL.

Например:

SHL AX,1 ;32-разрядное число в регистрах AX и DX

RCL DX,1 ;умножаем на 2

ПОДГОТОВКА К РАБОТЕ

1. Изучить краткие теоретические сведения и рекомендо­ванную литературу.

2. Написать текст программы, соответствующий вашему ва­рианту задания.

РАБОЧЕЕ ЗАДАНИЕ

Набить подготовленную программу, странслировать ее, скомпоновать и от­ладить.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какой тип перехода допускает команда LOOP.
2. Как работает команда LOOP.
3. Где задается количество повторений цикла, организованного при помощи команды LOOP.
4. Каким способом, кроме команды LOOP, можно организовать циклический процесс.
5. В командах сдвига куда попадает последний выдвинутый за пределы разрядной сетки разряд.

П Р Е Р Ы В А Н И Я

**Цель работы**: ознакомиться с основными принципами меха­низма прерываний.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

ПРЕРЫВАНИЯ - это события, заставляющие центральный про­цессор прервать выполнение текущей работы и перейти на выполнение программы, называемой ОБРАБОТЧИКОМ ПРЕРЫВАНИЯ. Переход этот осуществляется за малое время с помощью специально раз­работанных аппаратных средств.

Обработчик прерывания определяет причину прерывания, вы­полняет запланированные действия, после чего возвращает уп­равление прерванной программе.

Обычно прерывания вызываются событиями внешними по отношению к ЦП и требующими немедленных действий. Например:

- завершение операции ввода/вывода;

- обнаружение аппаратного сбоя;

- "катастрофа" (отказ питания).

Большинство современных процессоров поддерживают механизм ТИПОВ и УРОВНЕЙ ПРЕРЫВАНИЯ. Каждому типу обычно соответствует ячейка в памяти, называемая ВЕКТРОМ ПРЕРЫВАНИЯ, которая опре­деляет местоположение программы обработчика прерывания данно­го типа. Концепция типов прерывания позволяет назначать пре­рываниям приоритеты. Прерывания группируются по уровням, имею­щим одинаковый приоритет.

ЦП компьютеров, поддерживающих систему прерываний, долж­ны иметь средства блокирования прерываний на время выполнения критических участков программы (избирательного или глобально­го). При обработке прерывания ЦП обычно блокирует все прерывания этого или более низких уровней и разрешает - более высоких.

Процессор 8086 поддерживают 256 типов прерываний, вызываемых событиями трех групп:

- внутренние аппаратные прерывания;

- внешние аппаратные прерывания;

- программные прерывания.

Каждому типу прерывания соответствует свой номер от 0 до 255.

Внутренние аппаратные прерывания или отказы генерируются определенными событиями, возникающими в процессе выполнения программы (деление на 0, неправильный код операции). Закреп­ление номеров прерываний за определенными причинами жестко "зашито" в процессоре.

Внешние аппаратные прерывания инициализируются контрол­лерами периферийного оборудования или сопроцессорами.

Программные прерывания. Любая программа может иницииро­вать синхронное программное прерывание путем выполнения ко­манды Int с указанием номера прерывания, например,

Int 10h

Распределение номеров программных прерываний условно и не закреплено аппаратно.

Нижние 1024 байта системной памяти носят название ТАБЛИ­ЦЫ ВЕКТОРОВ ПРЕРЫВАНИЙ. Вектор занимает 4 байта: сегмент и смещение соответствующего обработчика прерывания.

ЦП, получив сигнал или команду прерывания, помещает в стек содержимое флагового регистра, очищает флаги TF и IF, заносит в стек содержимое регистров CS, IP и блокирует систе­му прерываний. Затем с помощью 8-битового числа - номера пре­рывания, установленного на внутренней шине, извлекает из таб­лицы векторов прерываний адрес (сегмент и смещение) соответствующего обработчика прерывания, заносит их соответственно в CS и IP и таким образом перердает управление обработчику прерывания.

Обычно обработчик разблокирует систему прерываний, сох­раняет регистры, которые будут им использоваться, и обрабаты­вает прерывание. В конце обработчика должна стоять команда IRET - возврат из прерывания. Эта команда восстанавливает первоначальное значение CS и IP и флагового регистра.

В качестве примера рассмотрим, как использовать команд­ные прерывания для вывода информации на экран дисплея.

Все необходимые экранные операции можно выполнить при помощи команды Int 10h, которая передает управление непосредственно в BIOS. ПЕРЕД вызовом прерывания Int 10h необходимо задать в регистре AH номер подфункции вывода. В других регистрах следует задать требуемые этой функции параметры. Номера под­функций, их параметры, а также регистры, в которые нужно их заносить, приведены в справочной информации по BIOS.

Для выполнения некоторых более сложных операций сущест­вует прерывание более высокого уровня Int 21h, которое вызывает функции DOS.

Экран можно представить в виде двухмерного пространства с адресуемыми позициями, в любую из которых может быть уста­новлен курсор или выведен символ. Видеомонитор, например, может иметь 25 строк (нумеруемых от 0 до 24) и 80 столбцов (нумеруемых от 0 до 79). Нумерация строк производится, начи­ная с левого верхнего угла экрана вниз, нумерация столбцов - начиная также с левого верхнего угла экран вправо.

ОЧИСТКА ЭКРАНА.

Запросы и команды остаются на экране дисплея до тех пор, пока не будут смещены в результате прокручивания ("скролинга") или не переписаны на том же месте другими зап­росами или командами. Когда программа начинает свое выполне­ние, экран может быть очищен. Очищаемая область экрана может начинаться в любой позиции экрана и заканчиваться в любой другой позиции с большим номером.

Начальное значение строки и столбца заносится в регистр CX, конечное - в DX, значение 07 - в регистр BH, 0600h - в AX.

В следующем примере выполняется очистка всего экрана.

MOV AX, 0600H ;AH=06 (прокрутка) AL=00 (весь экран)

MOV BH, 07 ;нормальный атрибут (черно/белый)

MOV CX, 0000 ;верхняя левая позиция (CH=00 - номер строки,

;CL=00 - номер столбца)

MOV DX, 184FH ;нижняя правая позиция (DH=18 - номер

;строки, DL=4F - номер столбца)

INT 10H ;передача управления в BIOS

УСТАНОВКА КУРСОРА.

Команда Int 10h включает в себя ус­тановку курсора в любую позицию экрана. Ниже приведен пример установки курсора на 5-ую строку и 12-тый столбец.

MOV AH, 02 ;подфункция установки курсора

MOV BH, 00 ;экранная страница 0

MOV DH, 05 ;строка 05

MOV DL, 12 ;столбец 12

INT 10H ;передача управления в BIOS

ВЫВОД НА ЭКРАН В БАЗОВОЙ ВЕРСИИ DOS.

Вывод на экран в базовой версии DOS требует определения текстового сообщения в сегменте данных, установки в регистре AH значения 09 (вызов функции DOS) и указания команды Int 21h. В процессе выполнения операции конец сообщения опреде­ляется по ограничителю (знак доллара '$'), как показано ниже:

NAMPRMP DB 'Это сообщение','$'

........

MOV AH, 09 ;подфункция отображения строки на экран

LEA DX, NAMPRMP ;загрузка адреса (смещения) сообщения в

;регистр DX

INT 21H ;вызов функции DOS

Знак ограничителя '$' можно записывать непосредственно после символьной строки, как показано в примере, внутри строки 'Это сообщение$' или в следующем операторе DB '$'. Используя данную операцию, нельзя вывести на экран символ доллара. Если символ доллара будет отсутствовать в конце вы­водимого сообщения, то на экран будут выводиться все после­дующие символы, пока код знака '$' не встретится в памяти.

Команда LEA загружает адрес области памяти NAMPRMP в регистр DX для передачи в DOS адреса выводимой информации.

ПОДГОТОВКА К РАБОТЕ

1. Изучить краткие теоретические сведения и рекомендо­ванную литературу.

2. Написать текст программы, которая бы очистила экран дисплея и вывела в центре экрана номер Вашей группы и Вашу фамилию, имя и отчество.

РАБОЧЕЕ ЗАДАНИЕ

Набить подготовленную программу, странслировать ее, скомпоновать и от­ладить.

!!! Помните, что при отладке программы отладчиком td.exe для выполнения команды прерывания не­обходимо использовать команду группового прохода F8.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое прерывание.

2. Что такое вектор прерывания.

3. На какие группы подразделяются типы прерываний про­цессора 8086.

4. Что представляет собой таблица векторов прерываний.

5. Какая последовательность обработки прерывания.

6. Какая команда должна стоять в конце обработчика пре­рываний.

7. Какой номер программного прерывания для вывода на экран средствами BIOS.

ПРОЦЕДУРЫ И МАКРОКОМАНДЫ

**Цель работы**: ознакомиться с понятиями процедуры и мак­рокоманды.

КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

При создании Вашей программы может возникнуть необходи­мость в разных местах программы выполнять одни и те же дейс­твия, возможно с различными данными. Повторяющиеся участки программы можно оформить в виде ПРОЦЕДУРЫ. В этом случае повторяющаяся последовательность кодов будет описана в прог­рамме один раз, а в текс­те основной программы в тех местах, где должен был бы стоять код процедуры, будет стоять команда вызова процедуры CALL. При описании процедуры ей присваивается имя. При вызове про­цедуры оператором команды CALL будет это имя.

Для описания процедуры используется директива PROC. Формат этой директивы приведен в лабораторной работе N 1. Напомним, что в этой лабораторной работе программа содержала одну процедуру, которая была оформлена в виде:

BEGIN PROC FAR

; выполняемые команды

RET

BEGIN ENDP

Оператор FAR информировал систему о том, что данная процедура может вызываться из другого сегмента. Директива ENDP оп­ределяла конец процедуры.

Кодовый сегмент может заключать любое количество проце­дур. Типичная организация многопроцедурной программы приве­дена на рисунке 1.

CODESG SEGMRNT PARA

BEGIN PROC FAR

……………..

CALL B10

CALL C10

RET

BEGIN ENDP

B10 PROC NEAR

…………

RET

B10 ENDP

C10 PROC NEAR

…………

RET

C10 ENDP

CODESG ENDP

END BEGIN

Рисунок 1.

Директивы PROC, используемые для описания процедур с именами B10 и C10 имеют тип NEAR, что указывает на то, что данные процедуры могут вызываться только из того кодового сегмента, в котором они описаны. В дальнейшем тип NEAR может быть опущен, так как это тип, при­нимаемый по умолчанию.

Каждой процедуре должно присваиваться уникальное имя, которое находится в поле имени процедуры (BEGIN, B10, C10). Начинаться описание каждой процедуры должно директивой PROC, заканчиваться директивой ENDP. В конце каждой процедуры долж­на стоять команда RET. Она служит для передачи управления (возврата) в вызывающую процедуру. Если Вы не поставите ко­манды RET в конце Вашей процедуры, то после ее завершения будут выполняться команды, стоящие в тексте программы за процедурой.

Для передачи управления процедурам в основной программе BEGIN используются команды CALL B10 и CALL C10. В результате выполнения команды CALL B10 управление будет передано проце­дуре B10, выполнятся команды, содержащиеся в описании проце­дуры, и по команде RET управление будет передано в основную программу BEGIN на команду, следующую за CALL B10. Команда CALL C10 действует аналогично.

Команда CALL при своем выполнении автоматически заносит в стек адрес команды, следующей за CALL (адрес возврата), и затем передает управление вызываемой процедуре. Для процедуры типа NEAR в стек заносится смещение команды, следующей за CALL, которое берется из регистра IP. Для процедуры типа FAR в стек заносится сегмент и смещение, которые берутся из регистров CS и IP. Команда RET извлекает адрес возврата из стека и помещает его в регистр IP для процедуры типа NEAR и в регистры IP и CS для процедуры типа FAR, тем самым передавая управление в вызывающую программу. Для правильного возврата из процедуры к моменту выполнения команды RET указатель стека должен находиться в том же состоянии, что и сразу после выполнения команды CALL. Помните, что команды PUSH, PUSHF, CALL, INT, INTO заносят информацию в стек, а значит и уменьшают указатель стека. Ко­манды POP, POPF, RET, IRET извлекают информацию из стека и тем самым увеличивают указатель стека. Например, если Вы в процедуре использовали несколько команд PUSH, желательно вы­полнить такое же количество команд POP для восстановления указателя стека или восстановить указатель стека каким либо иным образом.

В примере 1 приведен текст программы, иллюстрирующей воздействие выполняемой программы на стек.

Пример 1.

1 0000 STACKSG SEGMENT PARA STACK 'Stack'

2 0000 20\*(????) DW 32 DUP(?)

3 0040 STACKSG ENDS

4

5 0000 CODESG SEGMENT PARA 'Code'

6 0000 BEGIN PROC FAR

7 ASSUME CS:CODESG, SS:STACKSG

8 0000 1E PUSH DS

9 0001 2B C0 SUB AX,AX

10 0003 50 PUSH AX

11 0004 E8 0001 CALL B10 ;Вызвать B10

12 ; .............

13 0007 CB RET ;Завеpшить пpогpамму

14 0008 BEGIN ENDP

15 ;------------------------------------

16 0008 B10 PROC

17 0008 E8 0001 CALL C10 ;Вызвать С10

18 ; .............

19 000B C3 RET ;Веpнуться в

20 000C B10 ENDP ;вызывающую пpогpамму

21 ;-------------------------------------

22 000C C10 PROC

23 ; .....

24 000C C3 RET ;Веpнуться в вызывающую

25 000D C10 ENDP ;пpогpамму

26 ;-------------------------------------

27 000D CODESG ENDS

28 END BEGIN

Если Вы выполните трассировку данной программы, то уви­дите следующее. Текущей доступной ячейкой стека для занесе­ния или извлечения слова является вершина стека. Первая команда PUSH уменьшает значение SP на 2 и заносит содержимое регистра DS (например, 049F) в вершину стека, то есть по смещению 003E. Вторая команда PUSH также уменьшает SP на 2 и записывает содержимое регистра AX (0000) по адресу 003C. Команда CALL B10 уменьшает значение SP и записывает относительный адрес следующей команды (0007) в стек по адре­су 003A. Команда CALL C10 уменьшает значение SP и записы­вает относительный адрес следующей команды (000B) в стек по адресу 0038. При возврате из процедуры C10 команда RET извлекает 000B из стека (0038), помещает его в указатель команд IP и увеличивает SP на 2. При этом происходит автома­тический возврат по относительному адресу 000B в кодовом се­гменте, то есть возврат в процедуру B10. Команда RET в конце процедуры B10 извлекает адрес 0007 из стека (003A), поме­щает его в IP и увеличивает значение SP на 2. При этом про­исходит автоматический возврат по относительному адресу 0007 в кодовом сегменте. Команда RET по адресу 0007 завершает вы­полнение программы, осуществляя возврат типа FAR.

На рисунке 2 показано воздействие на стек при выполнении каждой команды. Приведено только содержимое стека по адре­сам 0034 - 003F и содержимое регистра SP.

Команда Стек SP

Начальное значение: xxxx xxxx xxxx xxxx xxxx xxxx 0040

PUSH DS (запись 049F) xxxx xxxx xxxx xxxx xxxx 049F 003E

PUSH AX (запись 0000) xxxx xxxx xxxx xxxx 0000 049F 003C

CALL B10 (запись 0007) xxxx xxxx xxxx 0007 0000 049F 003A

CALL C10 (запись 000B) xxxx xxxx 000В 0007 0000 049F 0038

RET (выборка 000B) xxxx xxxx xxxx 0007 0000 049F 003A

RET (выборка 0007) xxxx xxxx xxxx xxxx 0000 049F 003C

Смещение в стеке 0034 0036 0038 003A 003C 003E

Рисунок 2.

Значения сегментных регистров (CS, DS, SS) устанавливаются при загрузке программы в оперативную память.

МАКРОКОМАНДЫ подобны процедурам и представляют собой "мини-программы", которые можно вставлять в исходные программы, просто указывая их имена.

Макрокоманды представляют собой последовательность опе­раторов на языке Ассемблера (команд и директив на языке Ас­семблера), которые могут несколько раз появиться в программе.

Подобно процедурам макрокоманды имеют имена. После того, как макрокоманда задана, ее имя можно использовать в исходной программе вместо последовательности команд.

Хотя и макрокоманды и процедуры предоставляют возмож­ность краткой ссылки на часто используемую последовательность команд, между ними существуют и различия. Коды команд проце­дуры входят в объектный код однократно и процессор в про­цессе выполнения программы передает им управление (то есть вызывает их командой CALL) по мере необходимости. Напротив, коды команд макрокоманды будут включаться в объектный код столько раз, сколько раз вы указываете ее имя. Транслятор заменяет каждое имя макрокоманды на те команды, которые она представляет. Говорят, что транслятор "расширяет" макроко­манду. Следовательно, при выполнении программы микропроцес­сор исполняет команды макрокоманды непосредственно, не пере­давая управление в другое место памяти, как в процедуре. Та­ким образом, ИМЯ МАКРОКОМАНДЫ ПРЕДСТАВЛЯЕТ СОБОЙ ДИРЕКТИВУ ТРАНСЛЯТОРУ; оно служит командой транслятору, а не про­цессору.

По сравнению с процедурами макрокоманды имеют три пре­имущества:

1. Макрокоманды динамичны. За счет изменения входных параметров макрокоманды можно изменять не только объекты, ко­торыми оно манипулирует, но и выполняемые над ними действия. Напротив, в случае процедуры, можно изменять только переда­ваемые ей данные, что делает процедуры гораздо менее гибкими.
2. Применение макрокоманд вместо процедур ускоряет ис­полнение программы, так как процессору не надо выпол­нять команды вызова процедуры и возврата из нее.
3. Макрокоманды можно ввести в библиотеку, из которой программист может извлекать их при составлении других прог­рамм.

Однако, ничего не дается даром. Макрокоманды имеют ос­новной недостаток, которого лишены процедуры: при их при­менении объектные коды программы становятся длиннее (по срав­нению с процедурами), так как макрокоманды расширяются при каждом их появлении и память заполняется повторяющимися по­следовательностями команд.

СОСТАВ МАКРОКОМАНД (МАКРОСОВ)

Каждый макрос имеет три части:

1) Заголовок: директива MACRO, в поле имени которой указано ИМЯ МАКРОСА, а в поле операнда - необязательный СПИ­СОК ПАРАМЕТРОВ.

2) Тело - последовательность операторов Ассемблера (ко­манд и директив), которые задают действия, выполняемые мак­рокомандой.

3) Концевик - директива ENDM, отмечающая конец макроса.

Общий вид записи макрокоманды:

имя MACRO [список\_параметров]

;тело макрокоманды

ENDM

Макрокоманда должна находиться до определения сегмента.

Рассмотрим пример простого макроса по имени INIT1, который инициализирует сегментные регистры для EXE-программы.

INIT1 MACRO ;заголовок

ASSUME CS:CSEG, DS:DSEG, SS:STACKSG, ES:DSEG ;тело

PUSH DS

SUB AX, AX

PUSH AX

MOV AX, DSEG

MOV DS, AX

MOV ES, AX

ENDM ;концевик

Имена, на которые имеются ссылки в макрокоманде (CSEG, DSEG, STACKSG) должны быть определены где-либо в другом месте программы. Макрокоманду INIT1 теперь можно указать в том месте программы, где необходимо инициализировать регистры. Соответствующая ассемблерная строка будет выглядеть просто

INIT1

Ниже показан листинг программы, использующей макроко­манду INIT1. В листинге макрорасширения каждая команда, по­меченная "1", является результатом генерации макрорасширения. В макрорасширении отсутствует директива ASSUME, так как она не генерирует объектный код.

1 INIT1 MACRO ;заголовок

2 ASSUME CS:CSEG,DS:DSEG,SS:STECSG,ES:DSEG ;тело

3 PUSH DS

4 SUB AX, AX

5 PUSH AX

6 MOV AX, DSEG

7 MOV DS, AX

8 MOV ES, AX

9 ENDM ;концевик

10 0000 STSEG SEGMENT PARA STACK 'Stack'

11 0000 20\*(????) DW 32 DUP(?)

12 0040 STSEG ENDS

13 0000 DSEG SEGMENT PARA 'DATA'

14 0000 54 45 53 54 20 4F 46+ MESSGE DB 'TEST OF MACRO-INSTRUCTION',13

15 20 4D 41 43 52 4F 2D+

16 49 4E 53 54 52 55 43+

17 54 49 4F 4E 0D

18 001A DSEG ENDS

19 0000 CSEG SEGMENT PARA 'CODE'

20 0000 BEGIN PROC FAR

21 INIT1

1 22 0000 1E PUSH DS

1 23 0001 2B C0 SUB AX, AX

1 24 0003 50 PUSH AX

1 25 0004 B8 0000s MOV AX, DSEG

1 26 0007 8E D8 MOV DS, AX

1 27 0009 8E C0 MOV ES, AX

28 000B B8 0040 MOV AX, 40H

29 000E BB 0001 MOV BX, 01

30 0011 B9 001A MOV CX, 26

31 0014 BA 0000r LEA DX, MESSGE

32 0017 CD 21 INT 21H

33 0019 CB RET

34 001A BEGIN ENDP

35 001A CSEG ENDS

36 END BEGIN

ПОДГОТОВКА К РАБОТЕ

1. Изучить краткие теоретические сведения и рекомендо­ванную литературу.

РАБОЧЕЕ ЗАДАНИЕ

* + - 1. Вставьте в текст программы к лабораторной работе N 6 в начале и в конце программы вывод на экран сообщения, со­держащего номер Вашей группы, Ваши фамилию, имя и отчество. Выполните это с помощью процедуры.

1. Выполните п.1, но при помощи макрокоманды.
2. Странслируйте обе программы, создайте листинги, сравните их. Скомпонуйте и выполните обе программы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. На каком этапе работы с программой обрабатывается имя макрокоманды. Как это происходит.
2. Что выполняется быстрее, процедура или соответствующая макрокоманда.
3. Как работает команда CALL для процедуры типа NEAR.
4. Как работает команда RET для процедуры типа NEAR.
5. В чем различие работы команды CALL для процедуры типа FAR и процедуры типа NEAR.