РОСЖЕЛДОР
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Ростовский государственный университет путей сообщения»
(ФГБОУ ВО РГУПС)

А.В. Чернов, А.В. Тишина

АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ

Учебно-методическое пособие
для выполнения лабораторных, практических и
самостоятельных работ

Часть 2

Основы программирования
на языке ассемблера

Ростов-на-Дону
2017

УДК 681.31(75.8)

Чернов, А.В.

Архитектура информационных систем: [Электронный ресурс] учебно-методическое пособие. В 3 ч. Ч. 2. Основы программирования на языке ассемблера /А.В. Чернов, А.В. Тишина; ФГБОУ ВО РГУПС. – Ростов н/Д, 2017. – 51 с.

Учебно-методическое пособие предназначено для изучения студентами организации и принципов функционирования памяти, микропроцессора, организации ввода – вывода, а также приобретение навыков низкоуровневого программирования на языке ассемблера. Приводятся теоретические и методические рекомендации к выполнению работ по изучаемым темам. Рассмотрены принципы программирования и базовые приемы и на примере микропроцессоров семейства Intel 80x86.

Предназначено для выполнения лабораторных, практических работ и самостоятельных работ студентами направления подготовки 09.03.02 «Информационные системы и технологии» и 09.03.01 «Информатика и вычислительная техника» при изучении дисциплин «Архитектура информационных систем», «Архитектура вычислительных систем» и «Архитектура вычислительных и автоматизированных систем».

Одобрено к внесению в «Электронный университет» кафедрой «Вычислительная техника и автоматизированные системы управления».

ВВЕДЕНИЕ

Современный специалист в области создания программного обеспечения для вычислительной техники и автоматизированных систем должен обладать достаточными знаниями по использованию средств вычислительной техники в организации и управлении процессами разработки программного обеспечения. Низкоуровневое программирование позволяет четко усвоить принципы работы вычислительных машин и систем, а также их функциональных блоков, более рационально использовать их вычислительную мощность при разработке конкретного вида программного обеспечения, с учётом его особенностей.

Целью проведения лабораторных, практических и самостоятельных работ является изучение студентами организации и принципов функционирования памяти, микропроцессора, организации ввода – вывода, а также приобретение навыков низкоуровневого программирования на языке ассемблера.

Для выполнения работ требуются IBM совместимый персональный компьютер, операционная система Windows XP и выше, программный пакет ассемблера TASM.

1. Основные теоретические положения по программированию на языке ассемблера

1.1 Организация памяти для хранения программ

Согласно принципам Джона фон Неймана, электронная вычислительная машина (ЭВМ) выполняет вычисления в соответствии с программой, которая располагается в памяти ЭВМ. Любая программа включает в себя команды (операторы) и данные (операнды). Программа выполняется с целью получения результирующих данных на основе преобразования исходных, с возможным формированием промежуточных данных. В соответствии с концепцией хранимой в памяти программы, и команды и данные располагаются в единой памяти и представлены в двоичных кодах. Память представляет собой набор ячеек, каждая из которых имеет свой уникальный номер – адрес. Команды и данные хранятся в ячейках, и их местоположение в памяти определяется адресами соответствующих ячеек. Поскольку команды и данные на уровне кодов неотличимы друг от друга, то для различия команд и данных используется их размещение в различных областях памяти – сегментах.

Сегмент - это прямоугольная область памяти, характеризующаяся начальным адресом и длиной. **Начальный адрес (адрес начала сегмента)** – это номер (адрес) ячейки памяти, с которой начинается сегмент. **Длина сегмента –** это количество входящих в него ячеек памяти. Сегменты могут иметь различную длину. Все ячейки, расположенные внутри сегмента, перенумеровываются, начиная с нуля. Адресация ячеек внутри сегмента

ведется относительно начала сегмента; адрес ячейки в сегменте называется **смещением** или **эффективным адресом - ЕА** (относительно начального адреса сегмента). На рисунке 1.1 представлены примеры сегментов, указаны адреса ячеек в памяти, входящих в сегмент, и их смещения (в квадратных скобках) относительно начала сегмента (0100h).

В общем случае программа, размещенная в памяти, может иметь следующие сегменты: сегмент данных для хранения операндов, сегмент кода для хранения операторов программы и сегмент стека – дополнительную память для временного размещения информации. Начальные адреса сегментов помещаются микропроцессором в соответствующие сегментные регистры, о которых пойдет речь в дальнейшем.

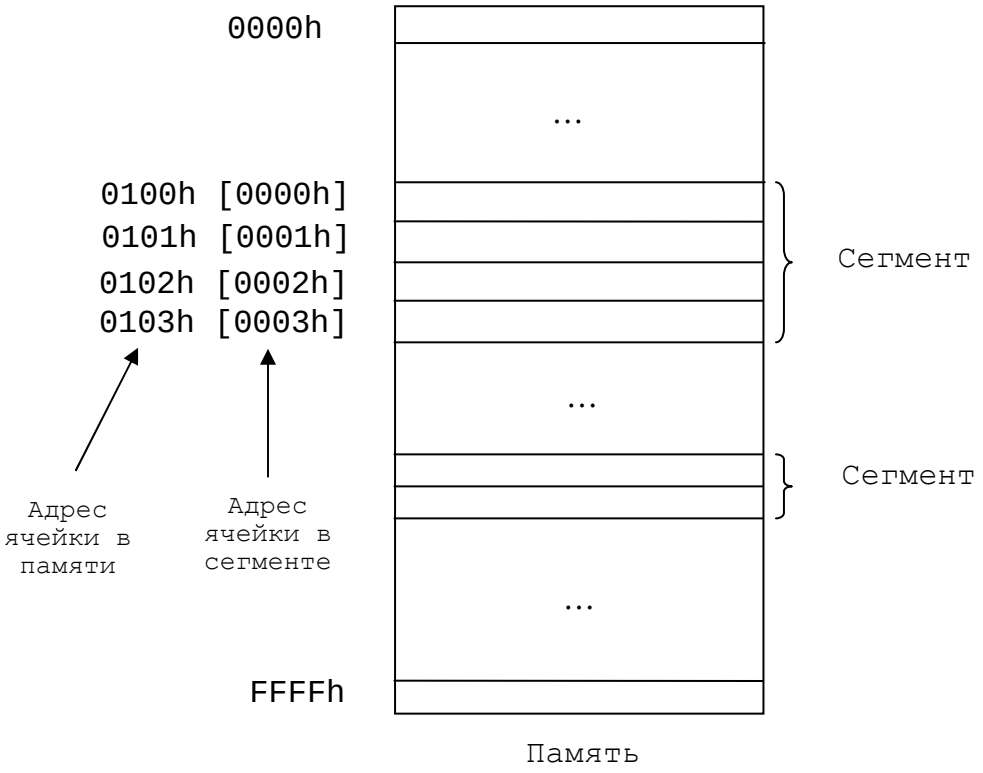


Рисунок 1.1 – Размещение сегментов в памяти

В зависимости от структуры данных и их местонахождении доступ к ним может осуществляться различными способами. Режимы адресации данных рассмотрим на примере микропроцессора Intel 8086.

1.2 Режимы адресации данных микропроцессора Intel 8086

1) **Непосредственный**. Данное длинной 8 или 16 бит является частью команды. Например: выражение-константа: 12; 1010B; 08AH; ‘AB’; XX-YY-5.

```
mov ah,12 ;переслать в регистр ah значение 12
add ax,08AH ;прибавить к содержимому регистра ax
            ;значение 08AH
```

2) **Прямой**. 16-битный эффективный адрес данного является частью команды. Например: переменная ± выражение- константа: CNT; CNT +5; ARRAY -5. Это означает, что соответствующие метки (переменные) определены предварительно в сегменте данных.

```
mov ah,cnt ;переслать в регистр ah значение по адресу cnt
           ;относительно начала сегмента данных
add ax,cnt+5 ;прибавить к содержимому регистра ax
             ;значение по адресу, вычисляемому как сумма
             ;адреса cnt и 5
```

3) **Регистровый**. Данное содержится в определяемом командой регистре (AX,BX,CX,DX,SI,DI,SP,BP).

Например: регистр: AX; BH; DL.

```
mov ah,al ;переслать в регистр ah значение из регистра al
add ax,dx ;прибавить к содержимому регистра ax значение
          ;из регистра dx
```

4) **Регистровый косвенный**. Эффективный адрес данного находится в базовом регистре BX или индексном регистре SI,DI:

$$EA = \begin{Bmatrix} (BX) \\ (SI) \\ (DI) \end{Bmatrix}.$$

Например: [регистр]: [BX].

```
mov ah,[bx] ;переслать в регистр ah значение по адресу,
            ;который находится в регистре bx
```

Таким образом, если в регистре `bx` находится значение 6, то в регистр `ah` пересылается информация, находящаяся по смещению `0006h` относительно начала сегмента данных.

5) **Регистровый относительный**. Эффективный адрес равен сумме 8 или 16 битного смещения и содержимого базового или индексного регистров:

$$EA = \begin{Bmatrix} (BX) \\ (BP) \\ (SI) \\ (DI) \end{Bmatrix} + \begin{Bmatrix} 8-offset \\ 16-offset \end{Bmatrix}.$$

Например: переменная + [регистр±выражение константа]: CST[BX]; MES[SI+10H]; [BX-1].

```
mov ah,cst[si] ;переслать в регистр ah значение по адресу,
               ;который определяется как сумма смещения
               ;метки cst относительно начала сегмента
               ;данных и регистра si
```

При этом метка `cst` должна быть предварительно описана в сегменте данных. Если `cst` имеет смещение `0002h` относительно начала сегмента данных, а в `si` записано значение `7h`, то эффективный адрес данного относительно начала сегмента данных составит `0009h` (`0002h+7h`).

6) **Базовый индексный**. Эффективный адрес равен сумме содержимого базового и индексного регистров, определяемых командой:

$$EA = \begin{Bmatrix}(BX)\\(BP)\end{Bmatrix} + \begin{Bmatrix}(SI)\\(DI)\end{Bmatrix}.$$

Например: [базовый регистр][индексный регистр] : [BX][DI].

```
mov ah,[bx][di] ;переслать в ah значение по адресу,
                ;равному сумме содержимого bx и di
```

Если в регистре `bx` записано значение `5h`, а в регистре `di` – значение `3h`, то в регистр `ah` пересылается информация по смещению `8h` (`5h+3h`) относительно начала сегмента данных.

7) **Относительный базовый индексный**. Эффективный адрес равен сумме 8 или 16 битного смещения и базово - индексного адреса:

$$EA = \begin{Bmatrix}(BX)\\(BP)\end{Bmatrix} + \begin{Bmatrix}(SI)\\(DI)\end{Bmatrix} + \begin{Bmatrix}8-offset\\(16-offset)\end{Bmatrix}.$$

Например: переменная + [базовый регистр±выражение константа] [индексный регистр±выражение константа]: E[BX+5][SI-2]; DATA[BX][SI]; [BX+2][SI].

```
mov ah,data[bx][si] ;переслать в регистр ah значение по
                    ;адресу, равному сумме смещения
                    ;метки data относительно начала
                    ;сегмента, содержимого bx и
                    ; содержимого регистра si
```

Если метка `data` имеет смещение `0002h`, в регистре `bx` записано значение `8h`, а в регистре `si` – значение `5h`, то в регистр `ah` передается значение, находящееся по смещению `0015h` относительно начала сегмента данных.

1.3 Режимы адресации переходов микропроцессора Intel 8086

Команды программы размещаются в сегменте кода, начальный адрес которого хранится в специальном сегментном регистре CS. При выполнении текущей команды микропроцессор должен знать адрес следующей команды, которая поступит на исполнение. Этот адрес хранится в специальном регистре IP. Таким образом, содержимое регистров CS и IP однозначно определяет местонахождение команды.

Последовательное выполнение команд микропроцессором может быть изменено благодаря командам условного и безусловного переходов, а также

командам вызова процедур. Рассмотрим виды переходов и способы вычисления адресов команд при выполнении этих переходов.

1) **Внутрисегментный прямой**. Эффективный адрес перехода равен сумме смещения и текущего содержимого IP. В команде условного перехода смещение только 8 бит (короткий переход). Например: метка±выражение константа: Labl1+27.

```
jmp lab11 ;выполняется безусловный переход на команду,
          ;которая помечена идентификатором lab11
lab11: mov ah, bh
```

К текущему содержимому регистра IP прибавляется смещение метки, которое вычисляется относительно значения регистра IP. Если текущее значение IP равно `0005h`, а смещение метки относительно текущего значения IP составляет `0004h`, то эффективный адрес перехода (смещение следующей выполняемой команды) равен `0009h`.

2) **Внутрисегментный косвенный**. Эффективный адрес перехода есть содержимое регистра или ячейки памяти, указанных в любом режиме кроме непосредственного. Значение регистра IP заменяется соответствующим содержимым регистра или ячейки памяти. Допустим только для безусловных переходов.

```
jmp [bx] ;выполняется переход на команду, адрес которой
         ;находится в ячейке памяти по адресу относительно
         ;начала сегмента данных, смещение ячейки памяти
         ; указано в регистре bx
```

Если в регистре `bx` хранится значение `0005h`, то адрес следующей выполняемой команды находится в ячейке памяти по смещению `0005h` относительно начала сегмента данных.

3) **Межсегментный прямой**. В команде указана пара `сегмент:смещение`, содержимое регистра IP заменяется одной частью команды (смещением), а содержимое CS – другой (сегментом). Допустим только в командах безусловного перехода.

Например: метка±выражение константа: BRANCH_EXT.

```
call far quickSort ;вызывается процедура quickSort,
                   ;которая находится в другом сегменте
                   ;кода, и адрес начала этого сегмента
                   ;помещается в cs, а смещение – в ip
```

Таким образом, адрес следующей выполняемой команды определяется смещением первой команды процедуры `quickSort` относительно своего сегмента кода.

4) **Межсегментный косвенный**. Заменяет содержимое IP и CS содержимым двух смежных слов из памяти, определенных в любом режиме

кроме, непосредственного и регистрового. Младшее слово помещается в IP, а старшее – в CS. Допустим только в командах безусловного перехода.

```
call far [bp+4] ;выполняется переход на команду, адрес
                ; которой содержится в четырех ячейках
                ; памяти, адреса которых начинаются со
                ; значения, указанного в в ячейке памяти
                ;по смещению bp+4
```

1.4 Слово состояния микропроцессора Intel 8086

Вычислительная обстановка вовремя выполнения программа определяется с помощью однобитных флагов, хранящихся в специальном регистре микропроцессора – PSW – слово состояния процессора. Все флажки делятся на две группы: флажки условий и флажки управления.

1. **Флажки условий**:

1.1. **Флажок знака SF**. Равен старшему биту результата: ноль – если результат положительный, и единица – если результат отрицательный.

1.2) **Флажок нуля ZF**. Устанавливается в единицу при получении нулевого результата и сбрасывается в ноль, если результат отличается от нуля.

1.3. **Флажок паритета PF**. Устанавливается в единицу, если младшие 8 бит результата содержат четное число единиц, в противном случае он сбрасывается в ноль.

1.4. **Флажок переноса CF**. При сложении (вычитании. устанавливается в единицу, если возникает перенос из младшего бита или заем из старшего бита.

1.5. **Флажок вспомогательного переноса AF**. Устанавливается в единицу, если при сложении (вычитании. возникает перенос (заем) из бита 3. Только для двоично-десятичной арифметики.

1.6. **Флажок переполнения OF**. Устанавливается в единицу, если возникает переполнение, т.е. получение результата вне допустимого диапазона. При сложении флажок устанавливается, если имеется перенос в старший бит и нет переноса из старшего бита и наоборот.

2. **Флажки управления**:

2.1. **Флажок направления DF**. Применяется в командах манипуляции цепочками.

2.2. **Флажок разрешения прерываний IF**. Когда установлен этот флажок, центральный процессор распознает маскируемые прерывания, иначе прерывания игнорируются.

2.3. **Флажок прослеживания (трассировки) TF**. Когда этот флажок установлен, после выполнения каждой команды генерируется внутреннее прерывание.

1.5 Список сокращений и условных обозначений

Прежде чем перейти к рассмотрению регистров микропроцессора и основных команд языка ассемблера, следует ознакомиться с условными

обозначениями, используемыми в дальнейшем. Перечень сокращений и условных обозначений приведен в таблице 1.

Таблица 1 – Перечень сокращений и условных обозначений

Сокращение	**Смысловое значение**	**Сокращение**	**Смысловое значение**
1	**2**	**3**	**4**
OPR	Операнд	DATA8	8- битный непосредственный операнд
SRC	Операнд- источник	DATA16	16- битный непосредственный операнд

Продолжение таблицы 1

1	**2**	**3**	**4**
DST	Операнд- получатель	AX,BX.CX,DX	16- битные регистры
REG	Регистр	AL,AH,BL,BH	8- битные регистры
RSRC	Регистр- источник	CL,CH,DL,DH	8- битные регистры
RDST	Регистр- получатель	IP	Регистр- указатель команды
CNT	Счетчик	SP	Регистр- указатель стека
DISP	Смещение	BP	Регистр базовый
D8	8- битное смещение	SI,DI	Регистры индексные
ADDR	Адрес	CS,SS,DS,ES	Регистры сегментные
EA	Эффективный адрес	DF,IF,TF	Флаги управляющие
SEG	Сегмент	OF,SF,ZF	Флаги условий
DATA	Непосредственный операнд	AF,PF,CF	Флаги условий

1.6 Общий формат ассемблерной команды

`Метка: Мнемоника Операнд, Операнд; Комментарий`

Метка представляет собой идентификатор, то есть последовательность букв и цифр, начинающаяся с буквы. Символы метки могут разделяться знаком подчеркивания. Все имена регистров являются зарезервированными и их использовать в качестве метки нельзя. Они используются для указания соответствующих регистров.

Команда (мнемоника) указывает транслятору с ассемблера какое действие должен выполнить данный оператор. В сегменте данных команда или псевдооператор определяет поле, рабочую область или константу.

Операнды - регистры, метки данных или непосредственные данные.

Комментарий служит для пояснения действий команды или директивы ассемблера. Посте точки с запятой комментарий записывается на одной строке. Для продолжения комментария на последующих стоках он записывается после точки с запятой.

1.7 Определение данных

Формат операторов резервирования и инициализации данных:

```
Переменная  Мнемоника  Операнд,...,
                       Операнд  ; Комментарий
```

Переменной (идентификатором) назначается смещение первого резервируемого байта.

Мнемоника определяет длину каждого операнда:

1. **DB** (определить байт). Диапазон для целых чисел без знака: 0...255, для целых чисел со знаком: -128…127.

2. **DW** (определить слово – два байта). Диапазон для целых чисел без знака: 0...65535, для целых чисел со знаком: -32768…32767.

3. **DD** (определить двойное слово – четыре байта).

Диапазон для целых чисел без знака: 0...4294967295, для целых чисел со знаком: -2147483648…2147483647.

Операнды показывают инициализируемые данные или объем резервируемого пространства. Операнд резервирует место без инициализации. Выражение может содержать константу или символ **?** для неопределенного значения.

Примеры:

```
Data_byte   DB  104
Data_word  DW 100H,FFH,-5
Data_DW    DD  5*25,0FFFDH,1
Data_str      DB ‘H’,’E’,’L’,’L’,’O’
Data_str1    DB ‘HELLO’
```

При определении большого числа ячеек можно применять оператор повторения **DUP (Операнд,...,Операнд)**.

Примеры:

`Arr   DB 30 DUP(1,2)` - зарезервирует 30 однобайтовых ячеек с начальными значениями 1 в нечетных и 2 в четных байтах;

`MM1   DB 15 DUP(14)` - означает, что по адресу MM1 находятся 15 байт содержащих шестнадцатиричную цифру 0EH (14 в десятичной системе).

`Arr_DW    DW 4 DUP(?)` – определяет по адресу Arr_DW 4 ячейки, содержащих произвольную информацию.

Операнды могут задаваться в различных системах счисления: для двоичной после значения операнда ставится символ B, для шестнадцатеричной – символ H, для десятичной – ничего.

Примеры:

```
Data_word  DW 100H
Met  DB 01010011B
ARG  DB 15
```

1.8 Регистры процессора Pentium

Пользовательские регистры. Они называются так потому, что программист может использовать их при написании своих программ. К этим регистрам относятся:

1. восемь 32-битных регистров, которые могут использоваться программистами для хранения данных и адресов (их еще называют регистрами общего назначения -РОН):

1.1. EAX – 32 бита/ AX – 16 бит/ AH/ AL – по 8 бит;

1.2. EBX – 32 бита/ BX – 16 бит/ BH/ BL – по 8 бит;

1.3. EDX – 32 бита/ DX – 16 бит/ DH/ DL – по 8 бит;

1.4. ECX – 32 бита/ CX – 16 бит/ CH/ CL – по 8 бит;

1.5. EBP – 32 бита/ BP – 16 бит;

1.6. ESI – 32 бита/ SI – 16 бит;

1.7. EDI – 32 бита/DI – 16 бит;

1.8. ESP – 32 бита/ SP – 16 бит.

2. шесть регистров сегментов: CS, DS, SS, ES, FS, GS по 16 бит;

3. регистры состояния и управления:

3.1. регистр флагов EFLAGS – 32 бита/ FLAGS – 16 бит;

3.2. регистр указателя команды EIP – 32 бита/ IP – 16 бит.

Микропроцессоры i486 и Pentium имеют в основном 32-разрядные регистры. Их количество, за исключением сегментных регистров, такое же, как и у i8086, но размерность больше, что и отражено в их обозначениях - они имеют приставку E (Extended).

Перечислим регистры, относящиеся к группе регистров общего назначения. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства (АЛУ), то их еще называют регистрами АЛУ:

1. EAX/AX/AH/AL (Accumulator register) - аккумулятор. Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно. Например, при выполнении операций умножения и деления используется для хранения первого числа, участвующего в операции, и результата операции после ее завершения.

2. EBX/BX/BH/BL (Base register) - базовый регистр. Применяется для хранения базового адреса некоторого объекта в памяти (например, массивов).

3. ECX/CX/CH/CL (Count register) - регистр-счетчик. Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды. К примеру, команда организации цикла loop кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра ECX/CX;

4. EDX/DX/DH/DL (Data register) - регистр данных. Так же, как и регистр EAX/AX/AH/AL, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит

неявно. Используется как расширение регистра- аккумулятора при работе с 32- разрядными числами.

Следующие два регистра используются для поддержки цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

1. ESI/SI (Source Index register) - индекс источника. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;

2. EDI/DI (Destination Index register) - индекс приемника (получателя). Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных, как **стек**. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

1. ESP/SP (Stack Pointer register) - регистр указателя стека. Содержит указатель вершины стека в текущем сегменте стека.

2. EBP/BP (Base Pointer register) - регистр указателя базы начального адреса поля памяти, непосредственно отведенного под стек. Предназначен для организации произвольного доступа к данным внутри стека.

Большинство из перечисленных регистров могут использоваться при программировании для хранения операндов практически в любых сочетаниях. Но некоторые команды используют фиксированные регистры для выполнения своих действий. Это нужно обязательно учитывать.

В программной модели микропроцессора имеется шесть сегментных регистров: CS, SS, DS, ES, GS, FS. Фактически в этих регистрах содержатся адреса памяти? с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:

1. **Сегмент кода**. Содержит команды программы. Для доступа к этому сегменту служит регистр CS (code segment register) - сегментный регистр кода. Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор (то есть, эти команды загружаются в конвейер микропроцессора).

2. **Сегмент данных**. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр DS (data segment register) - сегментный регистр данных, который хранит адрес сегмента данных текущей программы.

3. **Сегмент стека**. Этот сегмент представляет собой область памяти, называемую стеком. Работу со стеком микропроцессор организует по следующему принципу: последний записанный в эту область элемент

выбирается первым. Для доступа к этому сегменту служит регистр SS (stack segment register) - сегментный регистр стека, содержащий адрес сегмента стека.

4. **Дополнительный сегмент данных**. Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре DS. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре DS, при исфпользовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах ES, GS, FS (extension data segment registers).

В микропроцессор включены несколько регистров, которые постоянно содержат информацию о состоянии, как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. К этим регистрам относятся:

1. Регистр флагов EFLAGS/FLAGS;
2. Регистр указателя команды EIP/IP.

Младшая часть регистра EFLAGS полностью аналогична регистру FLAGS (слово состояния процессора, описанного выше) для i8086.

Регистр EIP/IP содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра EIP/IP.

1.9 Команды языка ассемблера

В персональных ЭВМ форматы команд достаточно разнообразны. Имеются команды с одним или двумя операндами.

Например:

1) формат команды **“регистр - регистр”** (2 байта);

2) формат команды **“регистр - память”** (2 - 4 байта);

3) формат команды **“регистр - непосредственный операнд”** (3 - 4 байта);

4) формат команды **“память- непосредственный операнд”** (3-6 байтов).

1.9.1 Команды передачи данных

Предназначены для пересылок данных, адресов и непосредственных операндов в регистры или в ячейки памяти. Их описание представлено в таблице 2.

Таблица 2 - Формат команд передачи данных

Название команды	**Мнемоника и формат команды**	**Описание действия**
Передать	MOV DST,SRC	(DST)←(SRC)
Загрузить эффективный адрес	LEA DST,SRC	(REG)← (SRC)
Загрузить в DS указатель	LDS DST,SRC	(REG)← (SRC) (DS) ← (SRC+2)
Загрузить в ES указатель	LES DST,SRC	(REG)← (SRC) (ES) ← (SRC+2)
Обменять	XCHG OPR1,OPR2	(OPR1) ←(OPR2)

Ни один из флажков не изменяется. Что касается режимов адресации, то получатель не может быть непосредственным и не может быть CS. В командах LEA, LES, LDS операнд REG не может быть сегментным регистром, а источник не может иметь непосредственный или регистровый режим. В команде MOV один из операндов должен быть регистром. В команде XCHG хотя бы один из операндов должен быть регистром, но ни один из операндов не может быть сегментным регистром.

1.9.2 Команды двоичных сложений и вычитаний

Применяются для выполнения арифметических операций над двоичными, упакованными и неупакованными двоично-кодированными десятичными числами. Их описание представлено в таблице 3.

Таблица 3 - Формат команд двоичных сложений и вычитаний

Название команды	**Мнемоника и формат команды**	**Описание действия**
Сложить	ADD DST,SRC	(DST)←(SRC) + (DST)
Сложить с переносом	ADC DST,SRC	(DST)← (SRC) + (DST)+(CF)
Вычесть	SUB DST,SRC	(DST)←(DST) - (SRC)
Вычесть с заемом	SBB DST,SRC	(DST)←(DST)-(SRC)-(CF)

Модифицируются все флажки условий. Один из операндов должен находиться в регистре. Другой операнд может иметь любой режим адресации.

1.9.3 Однооперандные команды двоичной арифметики и команды сравнения

Применяются для увеличения или уменьшения на единицу операнда и для сравнения двух операндов. Их описание представлено в таблице 4.

Таблица 4 - Формат одноопернадных команд двоичной арифметики и команды сравнения

Название команды	**Мнемоника и формат команды**	**Описание действия**
Инкремент	INC OPR	(OPR)←(OPR) + 1
Декремент	DEC OPR	(OPR)←(OPR) - 1
Изменить знак	NEG OPR	(OPR)←(OPR) - (OPR)
Сравнить	CMP OPR1,OPR2	(OPR1) - (OPR2)

Команда CMP по действию аналогична команде SUB, но она только устанавливает значения флагов в зависимости от полученного результата вычитания, но не изменяет содержимого операндов.

Модифицируются все флажки условий, но команды INC и DEC не воздействуют на флажок CF. Относительно режимов адресации, в командах INC, DEC, NEG не допускается непосредственный режим. В команде CMP один из операндов должен быть регистром. Другой операнд может иметь любой режим адресации, но OPR1 не может быть непосредственным значением.

1.9.4 Команды умножения и деления двоичных чисел

Применяются для выполнения операций умножения и деления над двоичными, упакованными и неупакованными двоично-кодированными десятичными числами. Их описание представлено в таблице 5.

Таблица 5 – Формат команд умножения и деления

Название команды	**Мнемоника и формат команды**	**Описание действия**
Умножить без знака	MUL RC	См. таблицу 4.6
Умножить со знаком	IMUL SRC	См. таблицу 4.6
Делить без знака	DIV SRC	См. таблицу 4.7
Делить со знаком	IDIV SRC	См. таблицу 4.7

Особенностью операций умножения и деления является наличие всего одного операнда(сомножителя или делимого). Второй операнд задан неявно; его местоположение фиксировано и зависит от размера операндов. Знаки результатов в операциях со знаком определяются по алгебраическим правилам.

Варианты размеров сомножителей, мест размещения второго операнда и результата для операции умножения представлены в таблице 6, а для операции деления – в таблице 7.

Если результат по размеру совпадает с размером сомножителей, то флаги CF и OF после завершения операции раны нулю, в противном случае - устанавливаются в единицу. Это значит, что результат вышел за пределы

младшей части произведения и состоит из двух частей, что необходимо учитывать при дальнейшей работе. Остальные флаги не определены.

Таблица 6 – Расположение операндов при умножении

Первый сомножитель	**Второй сомножитель**	**Результат**
Байт	AL	16 битов в AX; AL – младшая часть результата, AH – старшая часть результата (AX) ← (SRC) * (AL)
Слово	AX	32 бита в паре DX:AX; AX – младшая часть результата, DX – старшая часть результата (DX:AX) ← (SRC) * (AX)
Двойное слово	EAX	64 бита в паре EDX:EAX; EAX – младшая часть результата, EDX – старшая часть результата (EDX:EAX) ← (SRC) * (EAX)

Таблица 7 – Расположение операндов при делении

Делимое	**Делитель**	**Частное**	**Остаток**
Слово (16 бит) в регистре AX	Байт в регистре или ячейке памяти	Байт в регистре AL (AL)← (AX)/ (SRC)	Байт в регистре AH (AH)← (AX)/ (SRC)
Двойное слово (32 бита), в DX – старшая часть в AX – младшая часть	Слово (16 бит) в регистре или ячеек памяти	Слово (16 бит) в регистре AX (AX) ← (DX:AX)/ (SRC)	Слово (16 бит) в регистре DX (DX) ← (DX:AX)/ (SRC)
Учетверенное слово (64 бита), в EDX – старшая часть, в EAX – младшая часть	Двойное слово (32 бита) в регистре или ячейке памяти	Двойное слово (32 бита) в регистре EAX (EAX)← (EDX:EAX)/ (SRC)	Двойное слово (32 бита) в регистре EDX (EDX)← (EDX:EAX)/ (SRC)

Делитель может находиться в регистре или в памяти и иметь размер 8, 26 или 32 бита. Местонахождение делимого фиксировано. Результатом команды деления являются частное и остаток от деления. После выполнения операции деления содержимое флагов не определено, но возможно возникновение прерывания с номером ноль (так называемое «деление на ноль») в случаях, когда делитель равен нулю или частное не входит в отведенную для него разрядную сетку.

Что касается режимов адресации, то операнды источники не могут быть непосредственными значениями, а все другие режимы адресации допустимы. Операнды – получатели строго фиксированы.

1.9.5 Логические команды

Логические команды выполняют логические операции над битами операндов. Размерность операндов должна быть одинакова. Логические команды наиболее часто используются для селективных установок, инвертирования, сброса или проверки бит в операнде- получателе в соответствии с двоичным набором операнда-источника. Такие действия часто встречаются в операциях над битами регистров и данных ввода-вывода. При этом операнд источник называют маской, а сама операция называется маскированием. Описание логических команд представлено в таблице 8.

Таблица 8 - Формат логических команд

Название команды	**Мнемоника и формат команды**	**Описание действия**
Инвертировать	NOT OPR	(OPR)←not OPR
Объединить по «ИЛИ»	OR DST,SRC	(DST)←(DST) or (SRC)
Объединить по «И»	AND DST,SRC	(DST)←(DST) and (SRC)
Сложить по MOD2 («исключающее ИЛИ»)	XOR DST,SRC	(DST)←(DST) xor (SRC)
Проверить	TEST OPR1,OPR2	OPR1 and OPR2

Команда NOT не воздействует на флажки. Остальные команды сбрасывают OF и CF, оставляют AF не определенным и устанавливают CF, ZF, PF по обычным правилам.

Касательно режимов адресации, в команде NOT не допускается непосредственный операнд. В остальных командах один из операндов должен быть регистром. Другой операнд может иметь любой режим адресации.

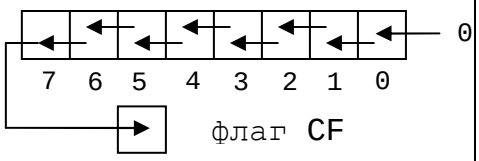
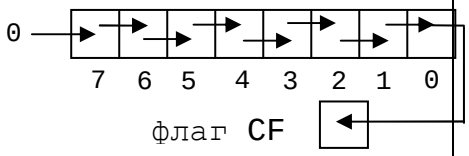
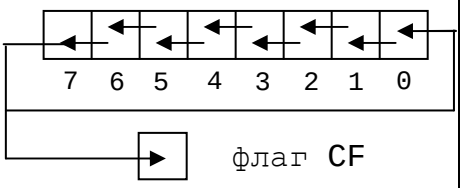
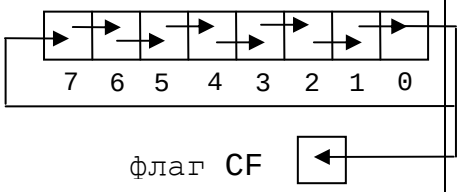
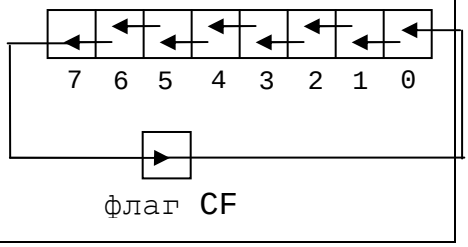
1.9.6 Команды сдвигов и циклических сдвигов

Эти команды также обеспечивают манипуляции над отдельными битами, перемещая биты операнда влево или вправо на определенное число битов, в зависимости от кода операции. Количество битов, на которое выполняется сдвиг, определяется счетчиком сдвигов CNT. Значение счетчика может задаваться статически (непосредственно во втором операнде) или динамически (в регистре CL перед выполнением команды сдвига). Счетчик сдвигов может содержать значение в диапазоне от 0 до 31; современные процессоры могут выполнять 64- разрядные сдвиги.

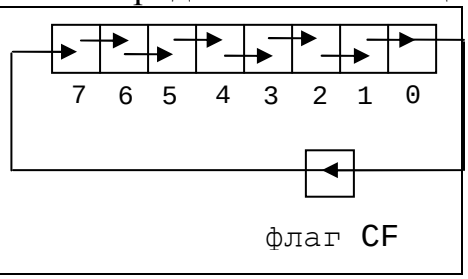
В командах сдвига влево с правой стороны операнда «вдвигаются» нули, а старшие биты «выдвигаются» с левой стороны и теряются, но последний из них сохраняется во флаге CF. Команды сдвига вправо аналогичным образом сдвигают биты вправо. Но арифметический сдвиг вправо не помещает слева нули, а дублирует в старшие биты знак операнда. Команды циклического сдвига отличаются от команд сдвига тем, что операнд считается «кольцом», в

котором выдвигаемые с одной стороны вдвигаются с другой стороны. Их описание представлено в таблице 9.

Таблица 9 - Формат команд сдвигов и циклических сдвигов

Название команды	**Мнемоника и формат команды**	**Описание действия**
Сдвинуть логически влево	SHL OPR,CNT	
Сдвинуть арифметически влево	SAL OPR,CNT	Не сохраняет знака, но устанавливает флаг OF в случае смены знака очередным выдвигаемым битом. В остальном полностью аналогична команде SHL
Сдвинуть логически вправо	SHR OPR,CNT	
Сдвинуть арифметически вправо	SAR OPR,CNT	Сохраняет знак, восстанавливая его после сдвига каждого очередного бита. В остальном – аналогична команде SHR
Сдвинуть циклически влево	ROL OPR,CNT	
Сдвинуть циклически вправо	ROR OPR,CNT	
1	**2**	**3**
Сдвинуть циклически влево через перенос	RCL OPR,CNT	

Продолжение таблицы 9

Сдвинуть циклически вправо через перенос	RCR OPR,CNT	

Команды арифметического сдвига позволяют выполнить «быстрое» умножение и деление операнда на степени двойки. Например, сдвиг числа влево на один разряд аналогичен его умножению на 2 (или на 10 в десятичной системе счисления). В командах циклического сдвига сдвигаемые биты поочередно становятся значением флага переноса CF.

Флажки SF, ZF, PF модифицируются командами сдвига, но команды циклических сдвигов на них не воздействуют. Флажок OF имеет смысл, если только счетчик равен 1. Команды сдвигов влияют на состояние флажка AF, но оно определенного смысла не имеет.

Что касается режимов адресации, то OPR может иметь любой режим адресации, кроме непосредственного.

1.9.7 Команды передачи управления

Эти команды нарушают естественный порядок выполнения команд посредством загрузки в командный счетчик адреса команды, к которой осуществляют переход. Решение о том, какая команда должна выполняться следующей может быть:

1) **безусловным** – из данной точки программы необходимо передать управление не следующей команде, а другой, которая находится на некотором удалении т текущей;

2) **условным** – решение о том, какая команда будет выполняться следующей, принимается на основе анализа некоторых условий или данных.

1. Безусловные переходы.

Безусловные переходы связаны с использованием команды безусловного перехода и команд обращений к процедурам.

1.1. Команда безусловного перехода

Формат команды

JMP [Модификатор] адрес_перехода

Этой командой задаются внутрисегментные и межсегментные переходы.

Описание команды безусловного перехода для внутрисегментных переходов представлено в таблице 10.

Таблица 10 – Формат команды для внутрисегментных переходов

Вариант внутрисегментного перехода	**Мнемоника и формат команды**	**Описание действия**
Прямой короткий переход (расстояние от команды JMP до адреса перехода не превышает -128 или 127 байт)	JMP SHORT OPR	(IP) ← (IP) + 8- битное смещение, определяемое OPR
Прямой переход (на расстояние от 128 байт до 64 Кбайт)	JMP [NEAR] OPR	(IP) ← (IP) + 16- битное смещение, определяемое OPR
Косвенный переход (в команде указывается не сам адрес перехода, а место, где он находится)	JMP OPR	(IP) ← (EA), где EA- эффективный адрес, определяемый OPR

Формат команды для межсегментных переходов описан в таблице 11.

Таблица 11 – Формат команды для межсегментных переходов

Вариант межсегментного перехода	**Мнемоника и формат команды**	**Описание действия**
Прямой переход	JMP FAR PTR OPR	(CS) ← начальный адрес сегмента, определяемого OPR (IP) ← смещение в сегменте из OPR
Косвенный переход	JMP OPR	(IP) ← (EA), где EA- эффективный адрес, определяемый OPR (CS) ← (EA + 2), где EA- эффективный адрес из OPR

Тип перехода определяется типом операнда. Межсегментные передачи управления реализуются только командами безусловных переходов. Не модифицируются все флажки условий. Что касается режимов адресации, то во внутрисегментных прямых переходах применяется относительный режим, в межсегментных прямых переходах - прямой режим. В косвенных переходах не допускается непосредственный режим, а в межсегментных косвенных переходах должна адресоваться память.

1.2. Обращение к процедурам

Процедура (подпрограмма) – это группа команд для решения конкретной подзадачи, обладающая средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку.

Другими словами, это правильным образом оформленная совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Описание процедуры состоит из заголовка, тела и конца процедуры:

имя_процедуры **PROC** [расст-е] – заголовок процедуры

…

команды, директивы ассемблера } - тело процедуры

…

ret

имя_процедуры **ENDP** } - конец процедуры

Имя процедуры – это идентификатор (метка), по которому происходит обращение к процедуре из основной программы или другой процедуры. Необязательный параметр [расстояние] характеризует возможность обращения к процедуре из другого сегмента кода. Моет принимать значения: **NEAR** – для процедур ближнего вызова (описание процедуры находится в том же самом сегменте кода, что и основная программа) и **FAR** – для процедур дальнего вызова (описание процедуры находится в другом сегменте кода). По умолчанию этот параметр равен NEAR.

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. В этом случае процессор воспринимает процедуру как часть исполняемого потока команд и начинает их выполнять. Процедура может размещаться:

1) в начале программы (до первой исполняемой команды);

```
c_s segment
assume cs:c_s
pr1 proc near
…
ret
p1 endp
begin:      ;начало программы
…
end begin
```

2) в конце программы (после команды корректного завершения работы и возвращения управления операционной системе- ОС);

```
c_s segment
assume cs:c_s
begin:
…
mov ah, 4ch
```

```
    int 21h        ;корректное завершение работы и передача управления ОС
    p1 proc near
    …
    ret
    p1 endp
    c_s ends
    end begin
```

3) внутри тела программы или другой процедуры (должен быть предусмотрен обход процедуры с помощью оператора JMP);

```
    c_s segment
    assume cs:c_s
    begin:
    …
    jmp m1
    p1 proc near
    …
    ret
    p1 endp
    m1: …
    mov ah, 4ch
    int 21h        ;корректное завершение работы и передача управления ОС
    c_s ends
    end begin
```

4) в другом модуле – часто используемы процедуры выносятся в отдельный файл, который оформляется как обычный файл ассемблера, а затем подвергается трансляции для получения объектного кода. Впоследствии этот объектный файл с помощью компоновщика можно объединить с файлом, в котором все или некоторые данные процедуры используются.

Специальный механизм вызова процедур, который поддерживается командами **CALL** и **RET**, позволяет сохранять информацию о состоянии программы в точке вызова процедуры. Команда **CALL** осуществляет вызов процедуры, сохраняя предварительно в стеке **адрес возврата** – адрес команды, следующей после команды CALL. При выполнении команды CALL осуществляется передача управления по адресу с символическим именем `имя_процедуры`.

Формат команды CALL:

CALL [Модификатор] имя_процедуры

Модификатор может принимать значения NEAR или FAR, для обращения к процедурам ближнего или дальнего вызовов, соответственно.

Команда **RET** считывает из стека адрес возврата и загружает его в регистр IP/EIP или регистры CS и IP/EIP, возвращая тем самым управление на команду, следующую в программе или другой процедуре за командой CALL.

Для процедур ближнего вызова адрес возврата полностью определяется содержимым регистра IP/EIP, а для процедур дальнего вызовы – содержимым дух регистров: CS и IP/EIP. При этом команда CALL запоминает в стеке сначала значение регистра CS, а затем значение регистра IP/EIP. Важно также отметить, что одна и та же процедура не может быть одновременно и процедурой ближнего вызова, и процедурой дальнего вызова.

Примечание: при использовании процедур программа должна обязательно содержать сегмент стека для обеспечения корректной обработки процедур. Тогда, в общем случае, программа будет иметь три сегмента: сегмент стека, сегмент данных и сегмент кода.

2. Условные переходы

В этих командах, также как и в командах безусловного перехода, нарушается естественный порядок выполнения команд. Но в условном переходе замена содержимого программного счетчика зависит от условия или от состояния флагов (без AF). Команды условно перехода, проверяющие некоторое условие, используются только совместно с командой сравнения CMP, рассмотренной выше, которая задает значение условия перехода. Команды условного перехода, работающие с флагами условий, могут использоваться как совместно с командой CMP (некоторые), которая модифицирует флаги, так и с другими командами, изменяющими флаги (все).

Команды условного перехода и их формат представлены в таблице 12.

Таблица 12 – Формат команд условного перехода

Название команды	**Мнемоника и формат команды**	**Критерий условного перехода (в CMP)**	**Значение флагов для перехода**
1	**2**	**3**	**4**
Перейти, если равно	JE OPR	OPR1 = OPR2	ZF = 1
Перейти, если не равно	JNE OPR	OPR1 <> OPR2	ZF = 0
Перейти, если ниже (меньше)/ не выше или равно (без знака)	JB/ JNAE OPR	OPR1 < OPR2	CF = 1
Перейти, если не ниже (меньше)/ выше или равно (без знака)	JNB/ JAE OPR	OPR1 >= OPR 2	CF = 0

Продолжение таблицы 12

1	**2**	**3**	**4**
Перейти, если ниже или равно/ не выше (без знака)	JBE/ JNA OPR	OPR1 <= OPR2	CF = 1 или ZF = 1
Перейти, если не ниже или равно/ выше (больше) (без знака)	JNBE/ JA OPR	OPR1 > OPR2	CF = 0 и ZF = 0
Перейти, если меньше/ не больше (со знаком)	JL/ JNGE OPR	OPR1 < OPR2	SF <> OF
Перейти, если не меньше/ больше или равно (со знаком)	JNL/ JGE OPR	OPR1 => OPR2	SF = OF
Перейти, если меньше или равно/ не больше (со знаком)	JLE/ JNG OPR	OPR1 <= OPR2	SF <> OF или ZF = 1
Перейти, если не меньше или равно/ больше (со знаком)	JNLE/ JG OPR	OPR1 > OPR2	SF = OF и ZF=0
Перейти, если ноль	JZ OPR	[OPR1 = OPR2]	ZF = 1
Перейти, если не ноль	JNZ OPR	[OPR1 <> OPR2]	ZF = 0
Перейти, если знак установлен	JS OPR	[OPR1 < OPR2]	SF = 1
Перейти, если знак сброшен	JNS OPR	[OPR1 > OPR2]	SF = 0
Перейти, если есть переполнение	JO OPR	-	OF = 1
Перейти, если нет переполнения	JNO OPR	-	OF = 0
Перейти, если паритет установлен	JP OPR	-	PF = 1
Перейти, если паритет сброшен	JNP OPR	-	PF = 0
Перейти, если перенос установлен	JC OPR	-	CF = 1
Перейти, если перенос сброшен	JNC	-	CF = 0

Если проверочное условие удовлетворяется, то действие этих команд заключается в следующем:

(IP) ← (IP)+ 8-битное смещение с расширением знака.

В противном случае программный счетчик не изменяется, и программа продолжается в естественном порядке. Не модифицируются флажки условий. Режимы адресации - относительно программного счетчика (IP). Операнд OPR должен быть в пределах от-128 до127 байт от команды, находящейся за командой перехода.

1.9.8 Команды циклов

Циклы организуются для многократного повторения одной или нескольких команд программы или процедуры. Цикл можно организовать, используя команды условного и безусловного переходов, рассмотренных выше, а можно с помощью специальных команд. Формат команд представлен в таблице 13.

Команда **LOOP** и ее расширения позволяют организовывать циклы, подобные циклам **for** в языках высокого уровня.

Таблица 13 – Формат команд циклов

Название команды	**Мнемоника и формат команды**	**Проверяемое условие**
Зациклить	LOOP OPR	(ECX/ CX) <> 0
Зациклить, пока ноль или равно	LOOPZ/ LOOPE OPR	(ECX/ CX) <> 0 или ZF = 0
Зациклить, по не ноль или не равно	LOOPNZ/ LOOPNE OPR	(ECX/ CX) <> 0 или ZF=1
Переход по CX	JCXZ OPR	(ECX/ CX) = 0

За исключением команды JCXZ, которая не изменяет (ECX/ CX), производится (ECX/ CX)←(ECX/ CX)-1, затем, если проверяемое условие удовлетворяется, (EIP/ IP)←(EIP/ IP)+D8 с расширением знака; в противном случае (IP) не изменяется и программа продолжается в естественном порядке.

Не модифицируются все флажки условий. Режимы адресации - относительно IP. Операнд OPR должен быть меткой, которая находится в диапазоне от -128 до127 байт от команды, следующей за командой цикла.

1.9.9 Стековые команды

1. Организация стека

Стек – это область памяти, специально выделяемая для временного хранения данных программы. Для стека в структуре программы предусмотрен отдельный сегмент. Регистры процессора для работы со стеком были рассмотрены выше .

Размер стека зависит от режима работы процессора и ограничивается значением 64 Кбайт в обычном режиме (или 4 Гбайт в защищенном режиме). В каждый момент времени доступен только один стек, начальный адрес сегмента которого содержится в регистре SS. Для перехода к другому стеку необходимо загрузить в SS его адрес. Запись и чтение данных в стеке осуществляется в соответствии с принципом LIFO (Last Input First Output - Последним Пришел Первым Ушел). По мере записи данных в стек он растет в сторону младших адресов памяти.

Концептуальная схема организации стека представлена на рисунке 2. Регистры SS, ESP/ SP и EBP/ BP используются комплексно, каждый из них имеет свое функциональное назначение. Регистр ESP/ SP всегда указывает на вершину стека, то есть содержит смещение относительно начала стека, по которому в стек был записан последний элемент. Для доступа к элементам не в вершине, а внутри стека используется регистр EBP/ BP – указатель базы кадра стека. Например, при входе в процедуру выполняется передача нужных параметров путем записи их в стек. Если процедура также использует стек, то доступ к этим параметрам становится проблематичным. Выход заключается в том, чтобы после записи параметров в регистр EBP/ BP записать адрес вершины стека ESP/ SP. Значение регистра ESP/ SP в дальнейшем будет изменяться, однако в регистре EBP/ BP хранится адрес, используя который, можно получить доступ к переданным параметрам.

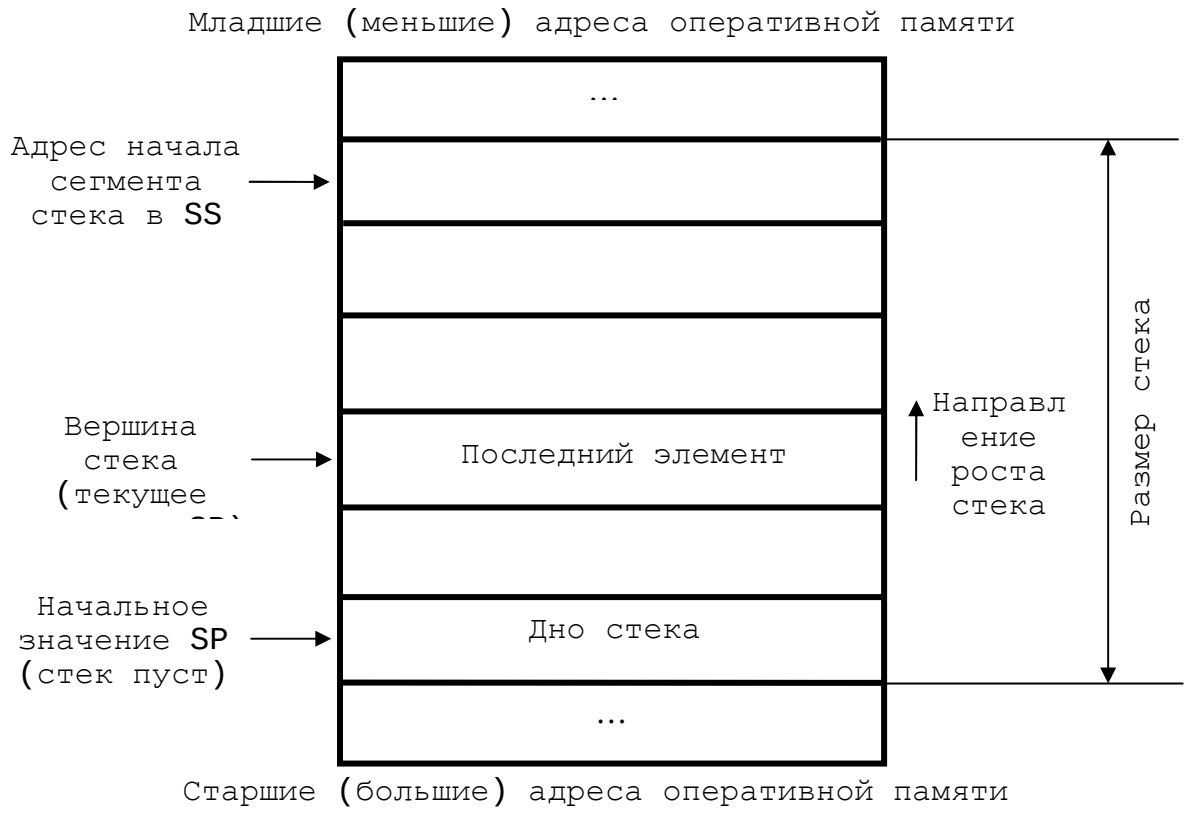


Рисунок 2 – Схема организации стека

Когда стек пуст, то значение регистра ESP/ SP равно адресу последнего байта сегмента (самому старшему адресу ячейки памяти в сегменте), выделенного под стек. Когда стек заполнен, то значение регистра ESP/ SP становится равным значению регистра SS, и дальнейшее добавление элементов невозможно. При помещении элементов в стек адрес вершины стека (содержимое регистра ESP/ SP) уменьшается (смещается в сторону меньших адресов), а при извлечении элементов из стека – увеличивается (смещается в сторону больших адресов).

Для работы со стеком предусмотрены специальные команды, формат которых представлен в таблице 14.

Таблица 14 – Формат стековых команд

Название команды	**Мнемоника и формат команды**	**Описание действия**
Включить в стек	PUSH SRC	(SP) ← (SP) – 2 ((SP + 1):SP) ← (SRC)
Извлечь из стека	POP DST	(DST) ← ((SP + 1):SP) (SP) ← (SP) + 2

Команда PUSH сначала уменьшает значение SP на 2, а затем по адресу (SP+1):SP записывает содержимое источника.

Команда POP сначала записывает содержимое из вершины стека (по адресу (SP+1):SP) по месту, указанному в DST, а затем увеличивает SP на 2.

В качестве операнда- источника SRC и в качестве операнда- приемника DST используются только регистры, кроме однобайтовых и регистра CS. Следует также отметить, что однобайтовые регистры сохранять в стеке нельзя.

2. Использование стека для передачи параметров

Стек часто используется при обращении к процедурам для передачи параметров в процедуры. При передаче управления процедуре процессор автоматически записывает в вершину стека два (для процедур ближнего вызова) или четыре (для процедур дальнего вызова) байта – адрес возврата в вызывающую программу. Если предварительно в стек были записаны переданные процедуре параметры или указатели на них, то они окажутся под адресом возврата.

Ранее уже упоминалось, что для работы со стеком в процессоре предусмотрены три регистра SS, SP и BP. Микропроцессор автоматически работает с регистрами SS и SP в предположении, что они всегда указывают на вершину стека. По этой причине их содержимое изменять не рекомендуется. Для произвольного доступа к данным в стеке используется регистр BP. Для корректной работы с использованием этого регистра содержимое стека должно быть правильно проинициализировано, что предполагает формирование адреса в нем, который бы непосредственно указывал на

переданные данные. Для этого в начала процедуры необходимо включить дополнительный фрагмент кода – **пролог процедуры**. Конец процедуры также должен быть оформлен особым образом для обеспечения корректного возврата из процедуры. Фрагмент кода, выполняющий эти действия, называется **эпилогом процедуры**. При этом нужно откорректировать содержимое стека, убрав из него ставшие ненужными аргументы, переданные и использованные в процедуре. Например, можно использовать последовательность из ***n*** команд вида **POP регистр**. Лучше всего это сделать в вызывающей программе сразу после возврата управления из процедуры.

Типичный фрагмент программы, содержащий вызов процедуры с передачей аргументов через стек, может выглядеть следующим образом:

```
s_s segment stack "stack"
 dw 12 dup(?)
s_s ends
d_s segment
 aa dw 10
d_s ends
c_s segment
 assume ss:s_s,ds:d_s,cs:c_s
begin:
 mov ax,d_s
 mov ds,ax
 push aa   ;запись в стек аргумента
 call pr1 ;вызов процедуры
 pop ax    ;очищаем стек, забирая аргумент в регистр ax
.mov ah, 4ch
 int 21h   ;завершение работы программы

pr1 proc near
 ;начало пролога
 push bp
 mov bp,sp
 ;конец пролога
 mov ax,[bp+4] ;доступ к аргументу по адресу aa для процедуры
               ;в регистре ax будет значение 10
 add ax,158h
 mov dx,ax
 ;начало эпилога
 mov sp,bp ;восстановление значения регистра sp
 pop bp    ;восстановление значения старого bp
           ;до входа в процедуру
 ret       ;возврат в вызывающую подпрограмму
 ;конец эпилога
pr1 endp
```

```
c_s ens
end begin
```

Код пролога состоит из двух команд: первая команда сохраняет содержимое регистра BP в стеке, чтобы исключить затирание находящегося в нем значения в вызываемой процедуре; вторая команда настраивает регистр BP на вершину стека для осуществления прямого доступа к содержимому стека.

Для доступа к последнему аргументу достаточно сместиться от содержимого BP на 4 (22 первых байта занимает адрес возврата, 2 следующих - искомое значение), к предпоследнему аргументу – на 6 и так далее (для процедур ближнего вызова).

Код эпилога процедуры восстанавливает состояние программы до момента входа в процедуру.

Что касается способов передачи параметров в процедуру через стек, то можно передавать либо сами данные (**передача параметров по значению**), либо их адреса (**передача параметров по ссылке**). Например, в приведенной выше программе использовался способ передачи аргументов по значению.

При передаче параметров через стек по значению на их размер накладываются ограничения, связанные с размерностью стека. Кроме того, в этом случае в вызываемой процедуре обрабатываются копии параметров. Таким образом, в приведенном выше примере значение по адресу aa в сегменте данных не изменится, то есть останется равным 10, независимо от выполняемых над этим значением действий в процедуре.

При передаче аргументов по ссылке в вызываемой процедуре обрабатывается не копия, а оригинал передаваемых данных. Поэтому при изменении данных в вызываемой процедуре они автоматически изменяются и в вызывающей программе, поскольку изменения касаются одной области памяти.

В следующем разделе рассмотрим процесс ассемблирования исходной программы (получения файла с расширением .EXE) для последующего ее выполнения.

2 Процесс ассемблирования и выполнения программы

2.1 Получение исполняемого модуля

Перед написанием программы, сначала нужно убедиться в наличии на компьютере файлов пакета TASM:

1) DPMILOAD.EXE;
2) DPMIMEM.DLL;
3) TASM.EXE;
4) TLINK.EXE;
5) TD.EXE (TDHELP.TDH - необязательно).

Для получения исполняемого файла программы необходимо:

1) создать в любом редакторе исходную программу на языке ассемблера, т.е. символьный файл, и сохранить его как файл с расширением .ASM.

2) этот файл транслировать путем ввода в командной строке следующей команды:

```
TASM \Путь\Имя файла.ASM /Z
```

После трансляции на экране появится сообщение:

`Assembling file:` транслируемый файл.

`Error messages:` сообщения об ошибках. (`None` - нет ошибок).

`Warning messages:` предупреждающее сообщение.

`Passes:` количество страниц.

`Remaining memory`: занимаемая память.

Результатом работы транслятора в случае отсутствия ошибок будет файл с расширением - .OBJ - объектный модуль. В противном случае на экране появится перечень ошибок с указание их типа и местоположения. После трансляции можно получить листинг - отпечатанную программу с относительными адресами и машинным кодом.

3) Транслированный без ошибок файл необходимо обработать компоновщиком т.е. набрать в командной строке следующую команду:

```
TLINK \Путь\ Имя файла.OBJ /V
```

Результатом при отсутствии ошибок будет файл с расширением .EXE или .COM - загрузочный модуль. Эти программы готовы к выполнению на ЭВМ. Их имена можно набрать на клавиатуре и нажать ENTER. Выполнение команд программы можно посмотреть в отладчике.

4) Для работы в отладчике необходимо иметь программу с расширением .EXE или .COM и набрать в командной строке команду:

```
TD \Путь\ Имя файла.EXE
```

Примечание. Для того чтобы посмотреть, какие ключи имеют программы TASM и TLINK, надо набрать их имена в командной строке и нажать клавишу ENTER.

2.2 Работа с отладчиком программ TURBO DEBUGGER

Отладчик TURBO DEBUGGER позволяет по шагам проследить процесс выполнения программы на уровне регистров процессора и ячеек памяти. Внешний вид окна отладчика представлен на рис. 3.

Нижнее меню в отладчике - меню функциональных клавиш.

Значения некоторых функциональных клавиш:

1) **F7** – трассировка программы.

2) **F8** – выполнение программы по шагам т.е. по программе перемещается полоса выбора (синяя), и будет выполнена та команда, на которой эта полоса размещена.

Примечание. Трассировка по F7 отличается от пошагового выполнения по F8 тем, то при наличии подпрограмм при трассировке будет по шагам выполняться не только основная программа, но и каждая подпрограмма, которая вызывается из основной программы. А при пошаговом выполнении по F8 по шагам выполняется только основная программа, а каждая подпрограмма выполняется как единый оператор.

После выполнения команды на экране появляется содержимое регистров, флагов и адрес следующей на очереди команды (соответствующие регистры подсвечиваются белым цветом).

3) **F10** - выход в главное, верхнее меню.

Запускаются команды или с помощью мыши или с помощью клавиш перемещения курсора на клавиатуре. Курсором выбирается нужная команда и нажимается клавиша ENTER или нажимается левая кнопка мыши, если выбор выполнялся с помощью мыши. Выбор группы верхнего меню также может выполняться с помощью мыши или с клавиатуры (ALT+ горячая клавиши соответствующей группы).

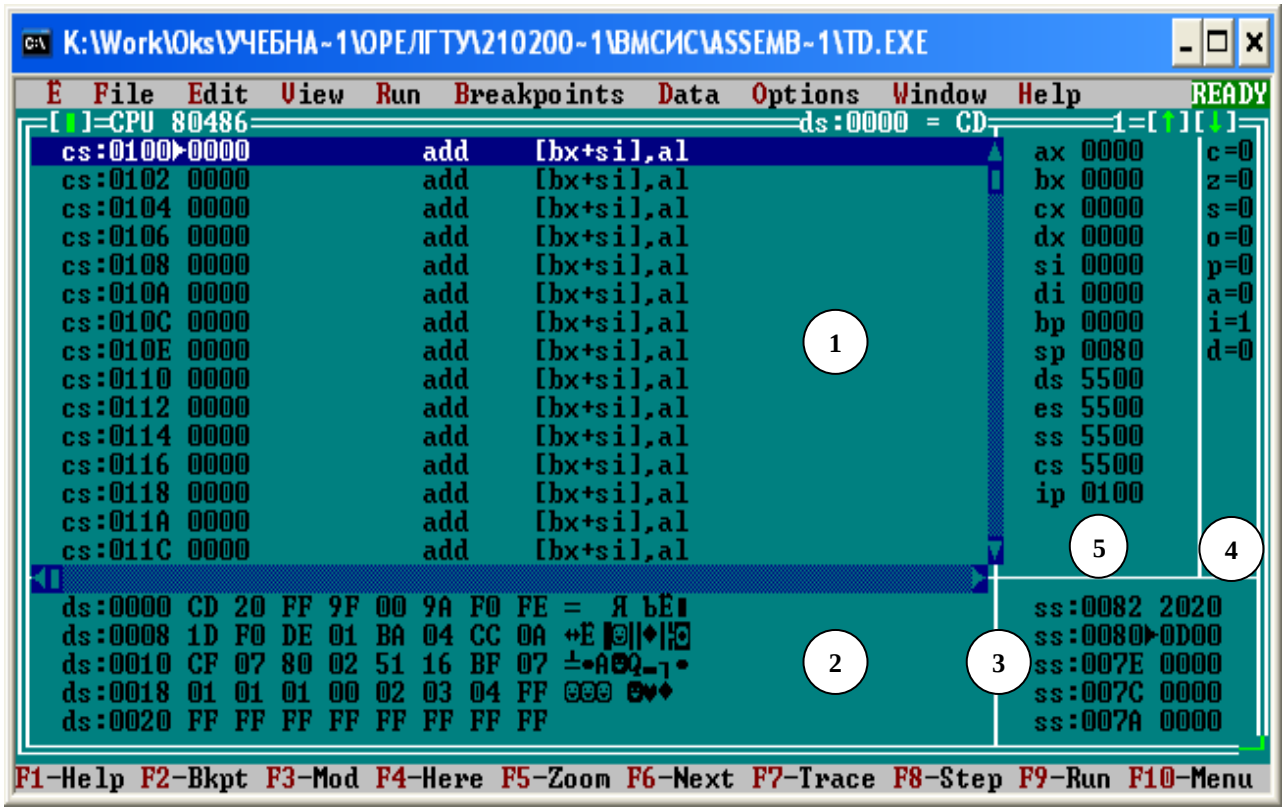


Рисунок 3 – Окно отладчика

В верхнем меню по команде FILE можно открыть любой файл, если он не был указан в команде TD при запуске отладчика.

По команде VIEW появляется еще меню, в котором DUMP - команда получения содержимого памяти по соответствующему адресу заданному в

регистре DS, т.е. содержимое данных определенных в нашей программе. Данные начинаются с нулевого относительного адреса. Эти данные можно изменять.

REGISTERS - после запуска этой команды появляется окно с регистрами, и данные, находящиеся в этих регистрах, можно изменять. Курсором или мышкой выбрать изменяемый регистр и перевести курсор на изменяемое данное, на клавиатуре в появившемся окне набрать новое данное и нажать ENTER.

Выход из отладчика по нажатию ALT+X.

Выход из любой команды по нажатию клавиши ESC.

Закрыть появившееся окно можно или нажать ALT+F3 или надо перевести мышкой курсор в левый угол окна на зеленый квадрат и нажать левую кнопку мыши.

Верхнее и нижнее меню обрамляют отдельные окна, содержащие следующую информацию.

1. О сегменте кода текущей программы. В этом окне отражается смещение команды относительно начала сегмента кода (регистра CS), код команды, мнемоника команды и операнды в шестнадцатеричной системе счисления.

2. О сегменте данных текущей программы. В этом окне отражается смещение данных относительно начала сегмента (регистра DS), их представление в шестнадцатеричном виде и в виде символа таблицы ASCII.

3. О сегменте стека текущей программы. В этом окне отражается смещение вершины стека (регистра SP) относительно начала сегмента стека (регистра SS) и элементы стека в шестнадцатеричном виде.

4. О регистре флагов процессора. В этом окне перечислены все флаги (кроме TF) и в процессе выполнения команд программы отображается их реакция на выполненную команду.

5. О регистрах микропроцессора. В этом окне перечислены регистры процессора и в процессе выполнения команд программы отображается их содержимое после очередной выполненной команды.

Анализируя информацию из этих окон, можно делать вывод о правильности выполнения текущей программы.

Далее рассмотрим, что представляет собой программа на языке ассемблера.

3 Программа на языке ассемблера

3.1 Структура программы

Программа на ассемблере представляет собой совокупность блоков памяти, называемых сегментами памяти. Программа может состоять из одного или нескольких таких блоков-сегментов. Каждый сегмент содержит совокупность предложений языка, каждое из которых занимает отдельную строку кода программы.

Предложения ассемблера бывают четырех типов:

1) **команды** или **инструкции**, представляющие собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд микропроцессора;

2) **макрокоманды** - оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями;

3) **директивы**, являющиеся указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении;

4) **строки комментариев**, содержащие любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором. Комментарий начинается с символа “**;**” и действует в пределах текущей строки.

3.2 Примеры программ

1. Программа сложения двух чисел:

```
s_s segment stack "stack" ;адрес начала сегмента стека
 dw 12 dup(?)          ;зарезервировано в памяти 24 ячейки с любым
                       ;значением
s_s ends               ;адрес  конца сегмента стека
d_s segment            ;адрес начала сегмента данных
aa dw 7145h, 23h       ;данные, т.е. числа 7145h и 23h записаны по
                       ;адресу aa и  aa+2 соответственно, т.к. они
                       ;определены как слова
sum dw 0               ; данное, т.е. число 0 записано по адресу sum
d_s ends               ;адрес  конца сегмента данных
c_s segment            ;начало сегмента кода
 assume ss:s_s,ds:d_s,cs:c_s ;псевдооператор, определяющий,
                             ;каким сегментным регистрам
                             ;соответствуют назначенные
                             ;метками адреса начала сегментов
begin:          ;начало программы
 mov ax,d_s     ;пересылка в регистр сегмента данных (ds) адреса
                ;начала сегмента данных, т.е. метки начала
сегмента
                ;данных
 mov ds,ax      ;пересылку в ds можно сделать только через
                ;промежуточную пересылку в рабочий регистр,
                ; например ax, т.к. из памяти в сегментные
регистры
```

```
                    ; записывать нельзя
    mov ax,aa       ;пересылка в регистр ax содержимого,
находящегося
                    ;по адресу aa, т.е. числа 7145h
    add ax,aa+2     ;сложить число, которое находится в регистре ax с
                    ;содержимым, находящимся по адресу aa+2
                    ;т.е.числа 23h и результат записывается в регистр
ax
    mov sum,ax      ;переслать содержимое ax, т.е. результат в ячейку
                    ;памяти по адресу sum
    mov ah,4ch      ;для правильного завершения программы
                    ;необходимо переслать в регистр ah - байт 4ch
    int 21h         ;и вызвать прерывание равное 21h
c_s ends            ;конец сегмента кода
end begin           ; конец программы. Метки начала и конца
                    ;программы должны совпадать.
```

2. Программа сложения двух чисел с учетом переноса из старшего разряда:

```
s_s segment stack "stack" ;начало сегмента стека
 dw 12 dup(?)             ; зарезервировано в памяти 24 ячейки
s_s ends                  ;конец сегмента стека
d_s segment          ;начало сегмента данных
 aa dw 5435h,4531h; данные, т.е. числа 5435h, 4531h записаны
                   ;по адресу aa и aa+2 соответственно, т.к. они
                   ;определены как слова
 s1 dw 2h        ; по адресу s1 записано число 2
 sum dw ?        ;любое данное записано по адресу sum (это метка)
d_s ends             ;конец сегмента данных
c_s segment          ;начало сегмента кода
 assume ss:s_s,ds:d_s,cs:c_s;
begin:               ;начало программы
 mov ax,d_s
 mov ds,ax
 mov ax,aa       ;пересылка в регистр ax содержимого, находящегося
                 ;по адресу aa, т.е. числа 5435h
 add ax,aa+2     ;сложить число, которое находится в регистре ax с
                 ;содержимым, находящимся по адресу aa+2, т.е.числа
                 ;4531h, результат записывается в ax
 jno kof         ;перейти, если нет переполнения (OF=0)
 mov ax,aa       ;если OF=1 - переполнение, выбрать опять число
 add ax,s1       ;и сложить его с другим
 kof: mov sum,ax ;переслать содержимое ax, т.е. результат в ячейку
```

```
                              ;памяти по адресу sum
    mov ah,4ch      ;для правильного завершения программы необходимо
                              ;переслать в регистр ah 4ch
    int 21h         ;и вызвать прерывание равное 21h
c_s ends            ;конец сегмента кода
end begin           ;конец программы
```

3. Программа сброса 3 и 4 бита в байте:

```
s_s segment stack "stack"
 dw 12 dup(?)
s_s ends
d_s segment
 aa db 75h
 sum db 0h
d_s ends
c_s segment
 assume ss:s_s,ds:d_s,cs:c_s
begin:
 mov ax,d_s
 mov ds,ax
 mov dl,aa   ;пересылка в регистр dl содержимого, находящегося
             ; по адресу aa, т.е. числа 75h
 and dl,11100111b ;сбросить биты 3 и 4, т.е. установит в ноль
 mov sum,dl       ; записать результат в sum
 mov ah,4ch
 int 21h
c_s ends
end begin
```

4. Программа вывода на экран строки символов:

```
s_s segment stack "stack"
 dw 12 dup(?)
s_s ends
d_s segment
 soob dw 'OK',0ah,0dh,'$' ;в первых апострофах то, что выводим,
                          ;затем 0ah -перевод строки,
                          ;0dh- возврат каретки, $- конец строки
d_s ends
c_s segment
 assume ss:s_s,ds:d_s,cs:c_s
begin:
```

```
 mov ax,d_s
 mov ds,ax
 mov ah,9h      ;вывод строки символов на экран - функция 9h
 lea dx,soob    ;в dx - адрес строки для вывода
 int 21h        ;выводим строку на экран
 mov ah,4ch
 int 21h
c_s ends
end begin
```

Примечание. Команда LEA, в отличие от команды MOV записывает в указанный регистр адрес ячейки, где находится данное, а не само данное по этому адресу.

5. Программа, использующая условный переход:

```
d_s segment
 aa dw 10
d_s ends
c_s segment
 assume ss:s_s,ds:d_s,cs:c_s
begin:
 mov ax,d_s
 mov ds,ax;
 cmp aa,10
 jbe met1     ; вызывает переход к метке met1,
              ;если содержимое aa меньше или равно 10
              ;иначе будут выполняться следующие команды
 met1: mov ah,4ch
       int 21h
c_s ends
end begin
```

6. Программа, использующая команды ввода-вывода (программа проверки, установлен ли математический сопроцессор)

```
d_s segment
 sum db 0
d_s ends
c_s segment
 assume ss:s_s,ds:d_s,cs:c_s
begin:
 mov ax,d_s
 mov ds,ax
 mov al,14h ;переписать N регистра (14-хранит информацию о
 out 70h,al ;периферии) в 70h порт (через дополнительную
            ; пересылку в регистр al)
```

```
 in al,71h  ;прочесть содержимое 14 регистра через 71h порт
 test al,10b ;проверить 1-ый бит, который указывает, есть ли
             ;сопроцессор
 jz no_c     ;если не установлен 1 бит, то переход на метку
   .....     ;если установлен (равен 1), то выполняются следующие
    .....    ;команды
 mov sum, al
 no_c: mov sum, ah ;сопроцессора нет
       mov ah,4ch
       int 21h
c_s ends
end begin
```

7. Программа, использующая вызов процедуры:

```
s_s segment stack "stack"
 dw 12 dup(?)
s_s ends
d_s segment
 aa dw 10
d_s ends
c_s segment
 assume ss:s_s,ds:d_s,cs:c_s
begin:
 mov ax,d_s
 mov ds,ax
 call pr1  ;вызов подпрограммы
 call pr1  ;вызов подпрограммы
 mov ah,4ch
 int 21h
pr1 proc near ;начало подпрограммы (ближний вызов)
 push ax       ; записать в стек регистр ax
 mov ax, aa
 pop ax        ;выбрать из стека регистр ax
 ret           ;команда возврата на следующую команду после
               ;вызова процедуры
pr1 endp       ;конец подпрограммы
c_s ends
end begin
```

Программы, имеющие такой вид, компонуются в файл с расширением .EXE. Для инициализации ассемблерной программы необходимо: 1) указать ассемблеру, какие сегментные регистры должны соответствовать сегментам; 2) загрузить в DS адрес начала сегмента данных. Программа в формате EXE может иметь любой размер. Программа с расширением .COM не должна превышать 64 Кбайт памяти.

При определении данных, начинающихся с буквы, необходимо перед буквой ставить цифру 0.

Лабораторная работа № 1.
Линейное исполнение программ. Операции над байтами

Цель работы

Цели лабораторной работы:

1) изучение принципов функционирования памяти и микропроцессора компьютера при последовательном исполнении команд программы;

2) приобретение навыков использования арифметических команд при написании ассемблерных программ;

3) приобретение навыков использования поразрядных логических команд при написании ассемблерных программ;

4) получение представления об особенностях обработки данных разных размерностей и режимах доступа к данным при выполнении арифметических операций.

Контрольные вопросы

1. Понятие сегмента, характеристики сегмента, организация сегмента.
2. На какие сегменты разбита память компьютера? В какие регистры записываются начальные адреса сегментов?
3. Какие регистры микропроцессора используются при выполнении арифметических операций?
4. На какие флаги воздействуют арифметические команды?
5. Какие режимы адресации могут применяться для доступа к данным при выполнении арифметических и поразрядных логических операций?
6. Особенности выполнения операции умножения. Особенности выполнения операции деления. Распределение регистров.
7. Основные логические операции и принципы их выполнения.
8. Правила формирования масок для установки и сброса битов.
9. Каким образом выполняются логические команды над словами?

Задание на лабораторную работу

1. Написать программу на языке ассемблера, которая выполняет арифметические и поразрядные логические операции над байтовыми значениями.

1.1. В сегменте данных определить два байтовых значения 10 и 27. В десятичной системе счисления и однобайтовое произвольное число в двоичной системе счисления.

1.2. В сегменте данных зарезервировать байтовые ячейки для хранения суммы и разности с нулевыми первоначальными значениями, двухбайтовую ячейку для хранения произведения с единичным первоначальным значением, две байтовые ячейки для хранения остатка от деления и частного с произвольными первоначальными значениями.

1.3. Выполнить сложение 10 и 27; полученный результат записать в соответствующую ячейку памяти.

1.4. Выполнить вычитание 10 и 27; полученный результат переслать в соответствующую ячейку памяти.

1.5. Изменить знак второго числа (27) и снова выполнить операцию вычитания 10 и -27.

1.6. Выполнить умножение 10 и -27 с учетом знака; результат записать в соответствующую ячейку памяти. Выполнить умножение 10 и -27 без учета знака.

1.7. Выполнить деление 27 на 10; полученные результаты записать в соответствующие ячейки памяти.

1.8. Переписать его в регистр, установить 2 любых бита в единицу, инвертировать все, сбросить 3 любых бита.

1.9. Полученный результат продублировать в другом регистре, сложить получившиеся значения по модулю два.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TURBO DEBUGGER, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. Как представляется число 27 и -27 в 16-ричной системе счисления?

2.2. Какое значение разности при выполнении вычитания в пунктах 1.4 и 1.5 задания?

2.3. Где размещается результат умножения 10 и -27?

2.4. В чем заключается разность произведения 10 и -27 при умножении со знаком и без учета знака?

2.5. В каких регистрах размещаются результаты деления 27 и 10, и чему равны значения частного и остатка от деления?

2.6. Чему равна маска для установки двух битов в единицу и почему?

2.7. Чему равна маска для сброса трех битов в ноль?

Лабораторная работа № 2.
Адресация межсегментных переходов

Цель работы

Цели лабораторной работы:

1) изучение принципов функционирования памяти и микропроцессора компьютера при выполнении межсегментных переходов;

2) приобретение навыков использования команд сдвига при написании ассемблерных программ;

3) получение представления об особенностях обработки данных и режимах доступа к данным при выполнении операций сдвига над данными.

Контрольные вопросы

1. Виды межсегментных переходов. Способы вычисления адресов переходов.

2. Флаги процессора и их использование в условиях.

3. Команды линейного логического и арифметического сдвигов. В чем заключается разница их выполнения?

4. Особенности выполнения команд циклического сдвига. Сферы применения этих команд.

5. Что указывает директива ASSUME в программе?

6. Как оформляется начало выполнения программы?

Задание на лабораторную работу

1. Написать программу на языке ассемблера, которая выполняет межсегментные переходы и операции сдвига над данными.

1.1. В программе определить 1 сегмент стека, 3 сегмента данных и 3 сегмента кода.

1.2. В сегменте стека зарезервировать 20 байт.

1.3. В первом сегменте данных определить однобайтовое число в двоичной системе счисления. Во втором сегменте данных определить адрес перехода на первый сегмент кода в виде двойного машинного слова. В третьем сегменте данных также определить однобайтовое число в двоичной системе счисления.

1.4. Начать выполнение с третьего сегмента кода и выполнить в нём с помощью команд линейного сдвига умножение на 2 числа из первого сегмента данных, а затем деление на 4 числа из третьего сегмента данных. Команды корректного завершения работы пометить меткой.

1.4. Затем перейти на метку, определенную во втором сегменте данных и выполнить переход в первый сегмент кода. Используя команды циклического сдвига, в регистре BL получить значение третьего бита числа из первого сегмента данных.

1.5. Далее выполнить переход во второй сегмент кода. В нём, используя команды циклического сдвига, в регистре BH получить значение пятого бита числа из третьего сегмента данных.

1.6. Затем перейти на метку конца, определенную в третьем сегменте кода.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TURBO DEBUGGER, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. Каким образом выполняется абсолютный прямой межсегментный переход, абсолютный косвенный межсегментный переход?

2.2. В каких регистрах содержится адрес перехода при прямом переходе, при косвенном переходе?

2.3. В каких регистрах будут находиться результаты умножения на 2 и деления на 4 при выполнении операций линейного сдвига? Чему равны полученные результаты в десятичной системе счисления?

2.4. Чему равны третий и пятый биты анализируемых чисел и какую позицию они занимают в регистрах BL и BH, соответственно?

2.5. Где находятся биты чисел, подвергнутые циклическому сдвигу и чему они равны?

Лабораторная работа № 3.
Команды условного и безусловного переходов.

Цель работы

Цели лабораторной работы:

1) изучение принципов функционирования памяти и микропроцессора компьютера при выполнении ветвлений и циклов;

2) приобретение навыков использования команд условного и безусловного переходов, циклов при написании ассемблерных программ;

3) получение представления об особенностях обработки данных, команд и режимах доступа к данным при организации ветвлений и циклов.

Контрольные вопросы

1. Ветвления в алгоритмах. Реализация ветвлений на языке ассемблера.

2. Команды условных и безусловного переходов. Каким образом вычисляются адреса переходов?

3. Циклы в алгоритмах. Организация циклов на языке ассемблера. Особенности цикла LOOP.

4. В каком регистре находится во время выполнения программы смещение кода? Каким образом вычисляется адрес команды?

5. Какую принципиальную роль играет оператор безусловного перехода JMP при организации ветвлений?

6. Что означает корректное завершение программы?

7. Реальный и защищённый режимы работы процессора (на примере Intel 8086). Вычисление физических адресов ячеек памяти.

Задание на лабораторную работу

1. Написать программу на языке ассемблера, которая реализует ветвления и циклы.

1.1. В сегменте данных определить два числа в шестнадцатеричной системе счисления, размером в один байт каждое.

1.2. Также в сегменте данных описать однобайтовую ячейку для хранения наибольшего общего делителя (НОД. двух чисел с произвольным первоначальным значением.

1.3. Используя команды переходов и цикла, найти НОД двух чисел, описанных в сегменте данных.

1.4. Полученный результат поместить в соответствующую ячейку памяти.

1.5. Используя команды циклического сдвига, переходов и цикла подсчитать количество единиц в НОД.

1.6. Полученное значение поместить в регистр DL.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TURBO DEBUGGER,

описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. Как изменяется содержимое регистра IP при выполнении переходов? Какие виды переходов используются в программе. Что содержится в регистре IP?

2.2. Чему равен адрес следующей команды при выполнении условия для перехода и в противном случае?

2.3. Каким образом организованы циклы в программе?

2.4. Какое значение будет находиться в регистре для НОД после подсчета количества единиц? Сколько раз нужно выполнить команду циклического сдвига, чтобы получить первоначальное значение?

Лабораторная работа № 4
Обработка массивов

Цель работы

Цели лабораторной работы:

1) изучение принципов функционирования памяти и микропроцессора компьютера при выполнении операций над массивами данных;

2) приобретение навыков использования команд ассемблера, связанных с обработкой массивов;

3) получение представления об особенностях обработки данных, команд и режимах доступа к данным при обработке массивов.

Контрольные вопросы

1. Массивы и их представление в памяти компьютера.

2. Режимы адресации данных, которые могут применяться для доступа к элементам массива.

3. Описание массивов в сегменте данных.

4. Особенности обработки двумерных массивов в ассемблерных программах. Вычисление эффективного адреса (EA. элемента двумерного массива).

5. Какие режимы адресации данных можно использовать для доступа к элементам двумерного массива?

Задание на лабораторную работу

1. Написать программу на языке ассемблера, которая использует массивы и позволяет вычислить числа Фибоначчи в заданном диапазоне.

1.1. В сегменте данных определить массив из 18 двухбайтовых ячеек с произвольным первоначальным значением, две двухбайтовые ячейки с нулевым первоначальным значением для размещения минимального и максимального элементов массива, соответственно.

1.2. Вычислить первые 18 чисел Фибоначчи и поместить их в массив, обращаясь к нему как к одномерному массиву.

1.3. Рассматривая имеющийся массив как двухмерный размера 3×6 (3 строки, 6 столбцов), найти наименьших из нечётных элементов второй строки и наибольший из чётных элементов четвёртого столбца.

1.4. Полученные результаты поместить в соответствующие ячейки памяти.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TURBO DEBUGGER, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. Как расположены элементы массива в памяти?

2.2. Сколько ячеек памяти отведено под массив?

2.3. Каким образом осуществляется доступ к элементам одномерного массива? Как изменяется индексный регистр, используемый для указания смещения элемента массива?

2.4. В каких регистрах процессора содержатся смещения по строке и смещение по столбцу для двухмерного массива? Каким образом они вычисляются? На что указывает метка начала массива?

Лабораторная работа № 5
Использование подпрограмм

Цель работы

Цели лабораторной работы:

1) изучение принципов функционирования памяти и микропроцессора компьютера при выполнении переходов, связанных с вызовами подпрограмм;

2) приобретение навыков использования команд безусловного перехода для обработки процедур при написании ассемблерных программ;

3) получение представления об особенностях обработки данных, команд и режимах доступа к данным при организации вызовов процедур.

Контрольные вопросы

1. Описание процедур. Варианты размещения процедур в программе.

2. Процедуры и сопрограммы. Особенности передачи управления при вызове процедур и при вызове сопрограмм.

3. Команды вызова процедуры и возврата из неё.

4. Механизмы обработки процедур ближнего и дальнего вызовов. Что представляет собой «адрес возврата» и где он размещается?

5. Обязательно ли наличие сегмента стека в программе, содержащей процедуры, и почему?

Задание на лабораторную работу

1. Написать программу на языке ассемблера, которая выполняет сортировку массива чисел по возрастанию, используя процедуры.

1.1. В сегменте стека зарезервировать 5 двухбайтовых ячеек.

1.2. В сегменте данных определить два массива из 8 однобайтовых ячеек. В первом массиве разместить исходные значения, во втором массиве - нули.

1.3. Используя один из алгоритмов сортировки, выполнить упорядочивание элементов массива по возрастанию.

1.4. Фрагмент программы, соответствующий сортировке, представить в виде процедуры ближнего вызова. Вариант размещения процедуры в программе выбрать самостоятельно.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TURBO DEBUGGER, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. Как расположены процедуры в сегменте кода?

2.2. Что содержит регистр IP при выполнении команды CALL?

2.3. Каким образом изменяется состояние стека при обращении к процедуре ближнего вызова?

2.4. Что представляет собой адрес возврата и чему он равен? Как изменяется состояние стека при возврате из процедуры?

2.5. В какую точку основной программы выполняется возврат из процедуры?

3. Удалить из программы сегмент стека. Запустить программу на исполнение. Объяснить, что происходит при вызове процедуры сортировки.

Лабораторная работа № 6
Обработка структур

Цель работы

Цели лабораторной работы:

1) изучение принципов функционирования памяти и микропроцессора компьютера при выполнении операций над структурами;

2) приобретение навыков использования команд для работы со структурами на примере обработки информации о пациентах;

3) получение представления об особенностях обработки данных, команд и режимах доступа к данным при обработке структур.

Контрольные вопросы

1. Структуры и определение шаблона структуры в программе.

2. Инициализация полей структуры в программе?

3. Режимы адресации для доступа к элементам структуры, для доступа к элементам массива структур.

4. Каким образом вычисляется расстояние до некоторого поля отдельного элемента массива структур?

5. Назначение оператора TYPE?

Задание на лабораторную работу

1. Написать программу на языке ассемблера, которая использует структуры для хранения и обработки информации о пациентах.

1.1. Определить шаблон структуры для хранения информации о пациентах, состоящей из следующих полей:

номер медкарты – dw с нулевым начальным значением;

пол – db с произвольным начальным значением;

год рождения – dw с нулевым начальным значением;

дата поступления – db с шаблоном ‘/ /’;

дата выписки – db с шаблоном ‘/ /’.

1.2. В сегменте данных определить три экземпляра записи, указав конкретную информацию о трёх пациентах.

1.3. В сегменте кода вывести информацию о количестве пациентов, поступивших на конкретную дату.

1.4. Получить сведения о количестве пациентов женского пола, которые были выписаны на определённую дату.

1.5. Найти год рождения пациента по номеру медкарты.

1.6. Найти количество пациентов мужского пола по указанному году рождения.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TURBO DEBUGGER, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. В каком месте программы расположено описание шаблона структуры?

2.2. Как расположены элементы структуры в памяти, массива структур в памяти?

2.3. Сколько байтов отведено под одну структуру, под весь массив структур?

2.4. Каким образом осуществляется доступ к элементам массива структур, к полям отдельной структуры? Как изменяются значения базовых и индексных регистров?

2.5. На что указывает метка начала массива? На что указывает название поля структуры?

Лабораторная работа № 7
Использование стека

Цель работы

Цели лабораторной работы:

1) изучение принципов функционирования памяти и микропроцессора компьютера при выполнении операций со стеком и строками;

2) приобретение навыков использования команд для работы со стеком и строками;

3) получение представления об особенностях обработки данных, команд и режимах доступа к данным при использовании стека.

Контрольные вопросы

1. Память с последовательным доступом. Виды памяти с последовательным доступом.
2. Определение стека. Организация стека.
3. Команды работы со стеком.
4. Какие регистры используются при работе со стеком? Каково их назначение?

Задание на лабораторную работу

1. Написать программу на языке ассемблера, которая использует стек для проверки баланса расстановки скобок в строке символов.

1.1. В сегменте стека определить стек из 20 двухбайтовых ячеек с начальным значением в виде символа «$».

1.2. В сегменте данных определить строку не более 20 символов, содержащую произвольное арифметическое выражение, в котором используются три вида скобок "( )", "[ ]" и "{ }". Последовательность и вложенность скобок может быть любая.

1.3. Также в сегменте данных определить байтовую ячейку для сохранения результата проверки.

1.4. В сегменте кода, используя стек, проверить, все ли скобки закрыты, соответствует ли каждая закрывающаяся скобка открывающейся и нет ли лишних закрывающихся скобок.

1.5. В соответствующую ячейку памяти поместить код результата проверки (0 – скобки расставлены правильно, 1 - несоответствие скобок, 2 – не все скобки закрыты, 3 – лишние закрывающиеся скобки).

1.6. Если возникла ошибка несоответствия скобок, то в регистр dl поместить код скобки, которая ожидается, иначе – 0.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TURBO DEBUGGER, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. С какого адреса в памяти начинается сегмент стека? Какое значение содержится в регистре SP перед началом загрузки элементов в стек? На что указывает регистр SP?

2.2. Как изменяется содержимое регистра SP при помещении одного элемента в стек, при извлечении одного элемента из стека?

2.3. В каком порядке помещаются в стек и извлекаются элементы из стека?

2.4. Что является признаком опустошения стека?

Лабораторная работа № 8
Использование стека и рекурсивных процедур

Цель работы

Цели лабораторной работы:

1) изучение принципов функционирования памяти и микропроцессора компьютера при выполнении операций со стеком для передачи параметров через стек и рекурсивными подпрограммами;

2) приобретение навыков использования команд для работы со стеком и подпрограммами для организации передачи параметров через стек;

3) получение представления об особенностях обработки данных, команд и режимах доступа к данным при организации передачи параметров в подпрограммы через стек.

Контрольные вопросы

1. Какие регистры используются при работе со стеком? Назначение регистра BP.

2. Когда передаваемые в процедуру аргументы записываются в стек? Какое место они занимают в стеке после входа в процедуру?

3. Формат процедуры при использовании передачи параметров через стек. Пролог и эпилог процедуры.

4. Какие действия выполняются в вызывающей программе после возврата из процедуры и для чего?

5. Как вычисляется адрес требуемого параметра в стеке в процедурах ближнего вызова? Как вычислить адрес аргумента в процедуре дальнего вызова?

6. Передача параметров по ссылке и ее особенности. Какой оператор используется для записи в регистр адреса данного, а не самого данного?

7. Передача параметров по значению и ее особенности.

Задание на лабораторную работу

1. Написать программу на языке ассемблера, которая и стек для передачи параметров в процедуру вычисления факториала числа.

1.1. В сегменте данных определить две двухбайтовые ячейки с одинаковым значением: 3h, ниже еще две двухбайтовые ячейки для сохранения результата с произвольным первоначальным значением.

1.2. В сегменте кода описать процедуру ближнего вызова, в которой содержится программа вычисления факториала числа.

1.3. Число, для которого необходимо вычислить факториал должно передаваться в качестве аргумента через стек.

1.4. В сегменте кода в основной программе выполнить дважды вызов процедуры вычисления факториала: в первом случае аргумент передается по значению (из первой ячейки), во втором случае – по ссылке (из второй ячейки).

1.5. Результаты вычисления факториала записать в соответствующие ячейки памяти, определенные в сегменте данных.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TURBO DEBUGGER, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. Каково содержимое стека до входа в процедуру и после?

2.2. Как изменяется содержимое стека при рекурсивных вызовах процедуры и возвратах из рекурсии?

2.3. На что указывает регистр SP после выполнения первой команды эпилога процедуры?

2.4. Каким образом в программе осуществляется передача параметра по значению и по ссылке?

2.5. Какое значение имеют ячейки памяти, отведенные под исходные данные, и почему?

Библиографический список

1. Юров, В.И. Assembler [Текст]/ В.И. Юров.- Учебник для вузов.- 2-е издание.- СПб.: Питер, 2006.- 637 с.: ил.- ISBN: 5-94723-581-1

2. Юров, В.И. Assembler. Практика [Текст]/ В.И. Юров.- Учебник для вузов.- 2-е издание.- СПб.- Питер, 2006.- 399 с.: ил.- ISBN: 5-94723-671-0

3. Абель, П. Язык ассемблера для IBM PC и программирования [Текст]/П. Абель/ Пер. с англ. Ю.В. Сальникова.- М.: Высшая школа, 1992.- 447 с., ил.

4. Пирогов, П.Ю. ASSEMBLER. Учебный курс [Текст]/ П.Ю. Пирогов.- М.: Издатель Молгачева С.В.- Нолидж, 2001.- 848 с.- ил.- ISBN: 5-89251-101-4

Оглавление

Учебное издание

Чернов Андрей Владимирович
Тишина Анджела Викторовна

АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ

Учебно-методическое пособие
для выполнения лабораторных, практических работ и
самостоятельных работ

Часть 2

Основы программирования
на языке ассемблера

«Электронный университет» ФГБОУ ВО РГУПС

Адрес университета:
344038, Ростов н/Д, пл. Ростовского Стрелкового Полка Народного Ополчения, 2.