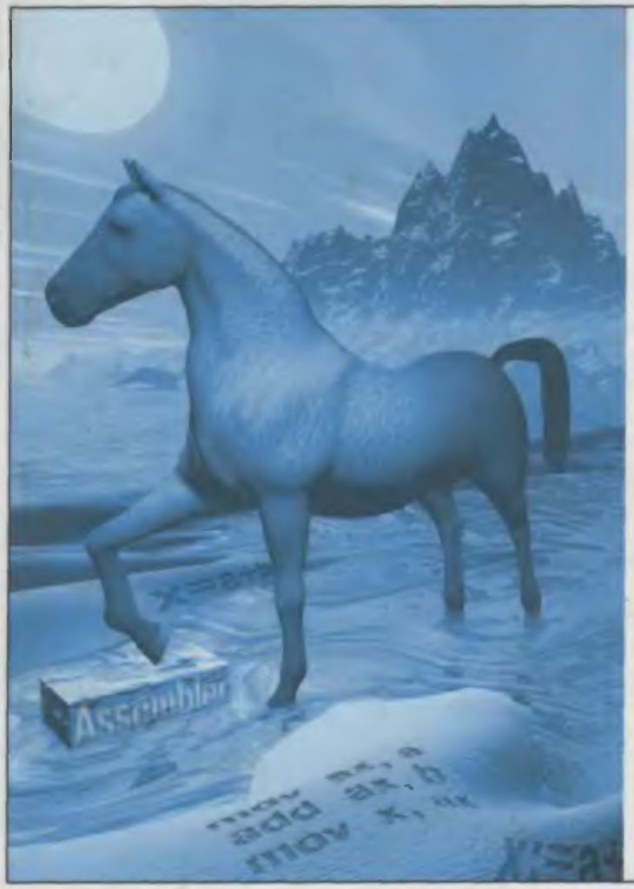
Надежда Григорьевна Голубь

Искусство программирования на АССЕМБЛЕРЕ

лекции и упражнения вторая редакция

Воспользуйтесь советами программиста-профессионала

Разберитесь в особенностях работы компьютера



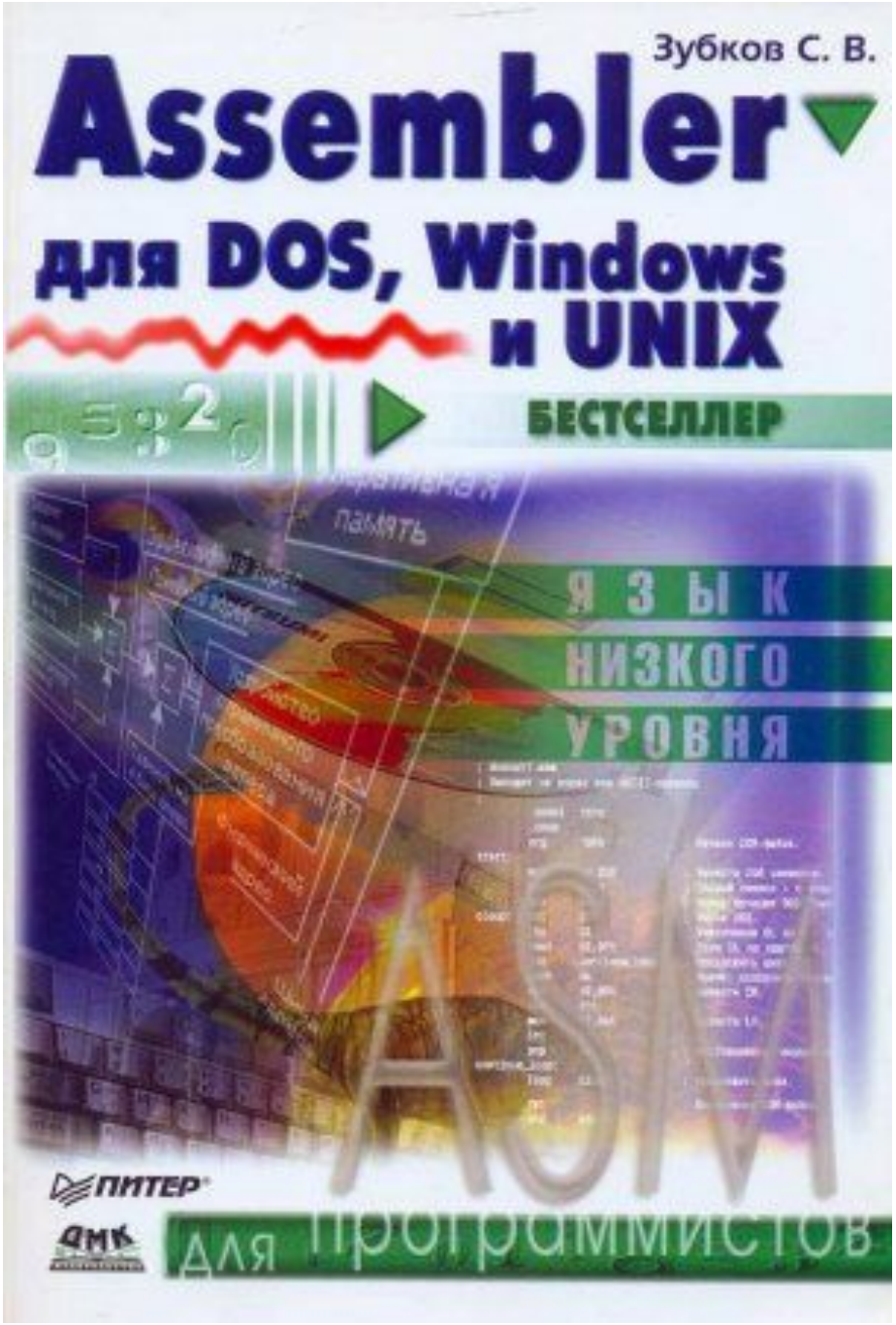
Подробно рассмотрены

- Системы счисления
- Внутреннее представление данных в компьютере IBM PC
- Принципы программирования на Ассемблере: команды, директивы, регистры, распределение оперативной памяти, способы адресации
- Соглашения о стыковке разноязычных модулей: Паскаль и Ассемблер, C/C++ и Ассемблер, особенности и хитрости их программирования
- Основные принципы тестирования и отладки программ
- Основные команды целочисленного программирования на Ассемблере
- Команды сопроцессора
- Машинный код
- Особенности 16- и 32-разрядного программирования
- Принципы организации ввода-вывода информации
- Программирование в DOS и Windows

издательство DiaSoft

книга-почтой интернет-магазин
www.diasoft.kiev.ua


Классика программирования
Питер Абель
АССЕМБЛЕР
язык и программирование
для
IBM® PC
пятое издание
"BEK"
Prentice Hall


Зубков С. В.
Assembler
для DOS, Windows и UNIX
БЕСТСЕЛЛЕР
ЯЗЫК НИЗКОГО УРОВНЯ
ASM
ПИТЕР
ДМК
для программистов

БИБЛИОТЕКА ПРОГРАММИСТА

Юрий Магда

Ассемблер для процессоров Intel Pentium

- Базовая программная архитектура процессоров
- Технологии параллельной обработки данных
- Архитектурно-программные решения для Intel Pentium
- Примеры программного кода

ПИТЕР

Лабораторные работы (5)

- Арифметические операции
  (для 4 типов данных)
- Условные переходы
- Ввод-вывод на АССЕМБЛЕРЕ
  (в DOS и LINUX)
- Обработка массивов
- Сопроцессор

Семейство процессоров 80x86

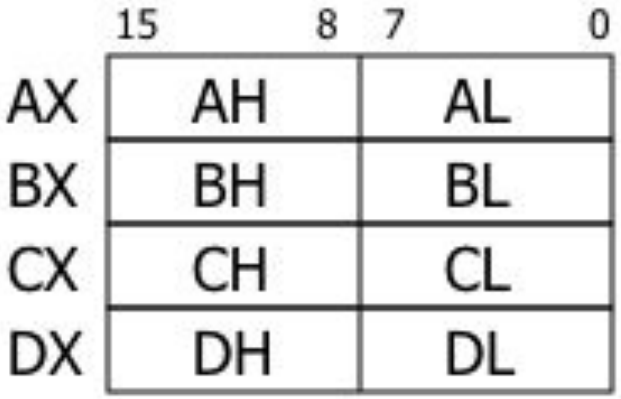
Семейство процессоров 80x86 корпорации Intel включает в себя микросхемы: 8086, 80186, 80286, 80386, 80486, Pentium, Pentium II, Pentium III и т.д. Совместимые с 80x86 микросхемы выпускают также фирны AMD, IBM, Cyrix. Особенностью этих процессоров является преемственность на уровне машинных команд: программы, написанные для младших моделей процессоров, без каких-либо изменений могут быть выполнены на более старших моделях. При этом базой является система команд процессора 8086, знание которой является необходимой предпосылкой для изучения остальных процессоров.

Регистры

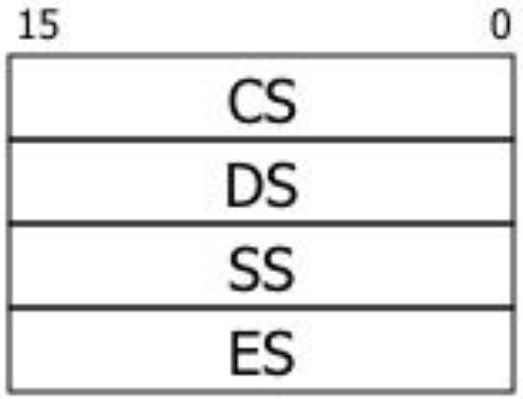
Для того, чтобы писать программы на ассемблере, нам необходимо знать, какие регистры процессора существуют и как их можно использовать. Все процессоры архитектуры x86 (даже многоядерные, большие и сложные) являются дальними потомками древнего Intel 8086 и совместимы с его архитектурой. Это значит, что программы на ассемблере 8086 будут работать и на всех современных процессорах x86.

Все внутренние регистры процессора Intel 8086 являются 16-битными:

РОН



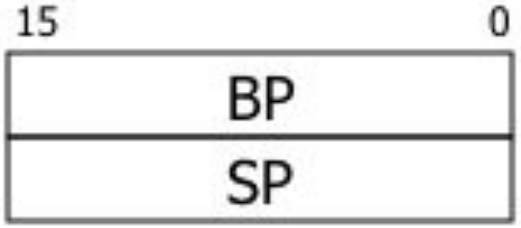
Сегментные регистры



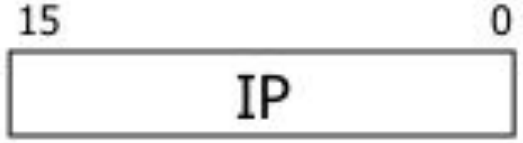
Индексные регистры

Регистр флагов

Регистры-указатели



Указатель команд



Регистры

Процессор 8086 имеет 14 шестнадцатиразрядных регистров, которые используются для управления исполнением команд, адресации и выполнения арифметических операций.

Регистры общего назначения.

К ним относятся 16-разрядные регистры AX, BX, CX, DX, каждый из которых разделен на 2 части по 8 разрядов:

- AX состоит из AH (старшая часть) и AL (младшая часть);
- BX состоит из BH и BL;
- CX состоит из CH и CL;
- DX состоит из DH и DL;

В общем случае функция, выполняемая тем или иным регистром, определяется командами, в которых он используется. При этом с каждым регистром связано некоторое стандартное его значение. Ниже перечисляются наиболее характерные функции каждого регистра:

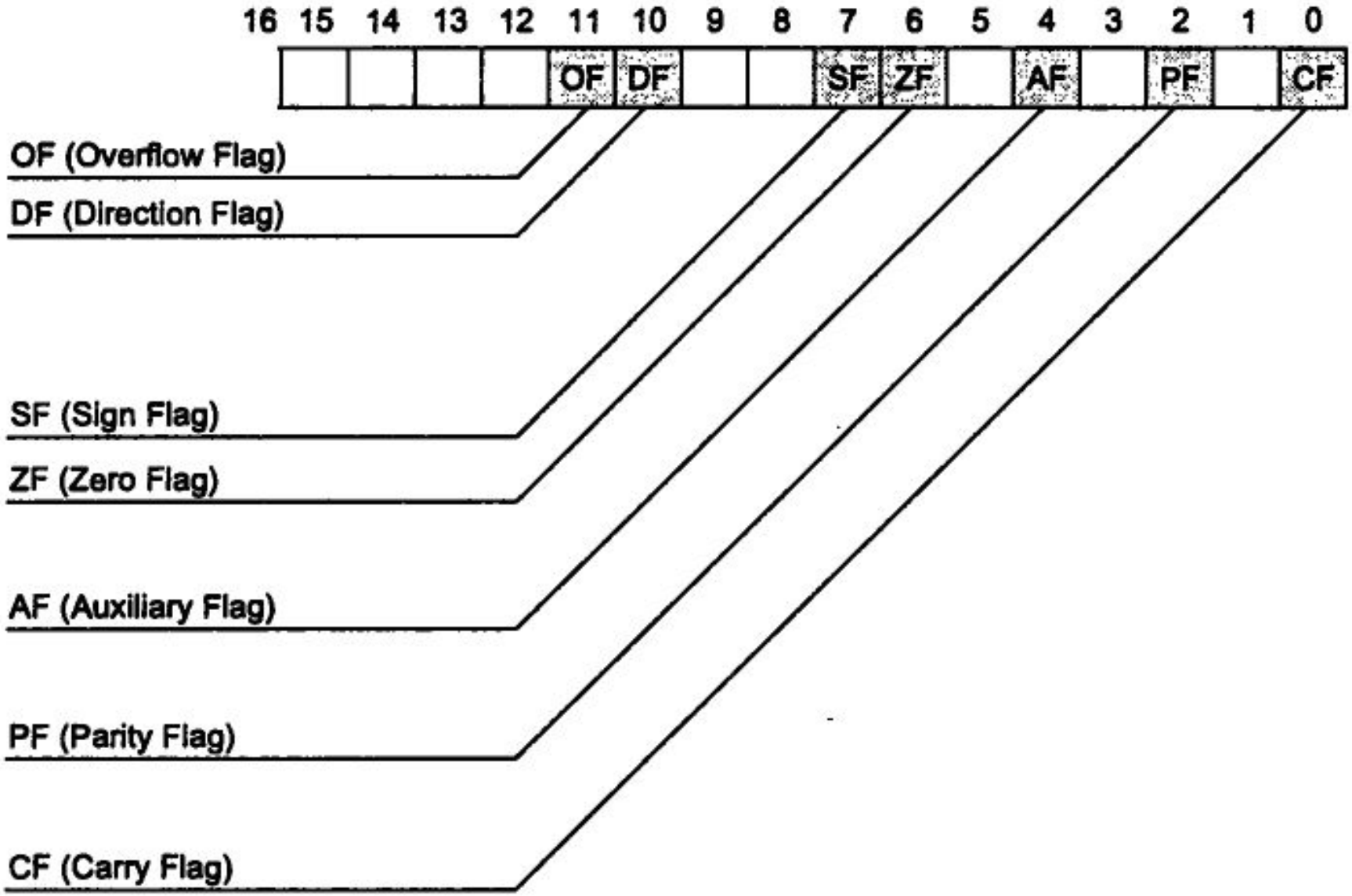
- **регистр AX** служит для временного хранения данных (регистр аккумулятор); часто используется при выполнении операций сложения, вычитания, сравнения и других арифметических и логических операции;
- **регистр BX** служит для хранения адреса некоторой области памяти (базовый регистр), а также используется как вычислительный регистр;
- **регистр CX** иногда используется для временного хранения данных, но в основном служит счетчиком; в нем хранится число повторений одной команды или фрагмента программы;
- **регистр DX** используется главным образом для временного хранения данных; часто служит средством пересылки данных между разными программными системами, в качестве расширителя аккумулятора для вычислений повышенной точности, а также при умножении и делении.
- ***Регистры для адресации****.* В микропроцессоре существуют четыре 16-битовых (2 байта или 1 слово) регистра, которые могут принимать участие в адресации операндов. Один из них одновременно является и регистром общего назначения — это регистр **BX**, или базовый регистр. Три другие регистра — это указатель базы **BP**, индекс источника **SI** и индекс результата **DI**. Отдельные байты этих трех регистров недоступны.

Любой из названных выше 4 регистров может использоваться для хранения адреса памяти, а команды, работающие с данными из памяти, могут обращаться за ними к этим регистрам. При адресации памяти базовые и индексные регистры могут быть использованы в различных комбинациях. Разнообразные способы сочетания в командах этих регистров и других величин называются способами или режимами адресации.

Регистры сегментов. Имеются четыре регистра сегментов, с помощью которых память можно организовать в виде совокупности четырех различных сегментов. Этими регистрами являются:

- **CS** — регистр программного сегмента (сегмента кода) определяет местоположение части памяти, содержащей программу, т. е. выполняемые процессором команды;
- **DS** — регистр информационного сегмента (сегмента данных) идентифицирует часть памяти, предназначенной для хранения данных;
- **SS** — регистр стекового сегмента (сегмента стека) определяет часть памяти, используемой как системный стек;
- **ES** — регистр расширенного сегмента (дополнительного сегмента) указывает дополнительную область памяти, используемую для хранения данных.
- Эти 4 различные области памяти могут располагаться практически в любом месте физической машинной памяти. Поскольку местоположение каждого сегмента определяется только содержимым соответствующего регистра сегмента, для реорганизации памяти достаточно всего лишь, изменить это содержимое.
- ***Регистр указателя стека***. Указатель стека **SP** – это 16-битовый регистр, который определяет смещение текущей вершины стека. Указатель стека SP вместе с сегментным регистром стека SS используются микропроцессором для формирования физического адреса стека. Стек всегда растет в направлении меньших адресов памяти, т.е. когда слово помещается в стек, содержимое SP уменьшается на 2, когда слово извлекается из стека, микропроцессор увеличивает содержимое регистра SP на 2.

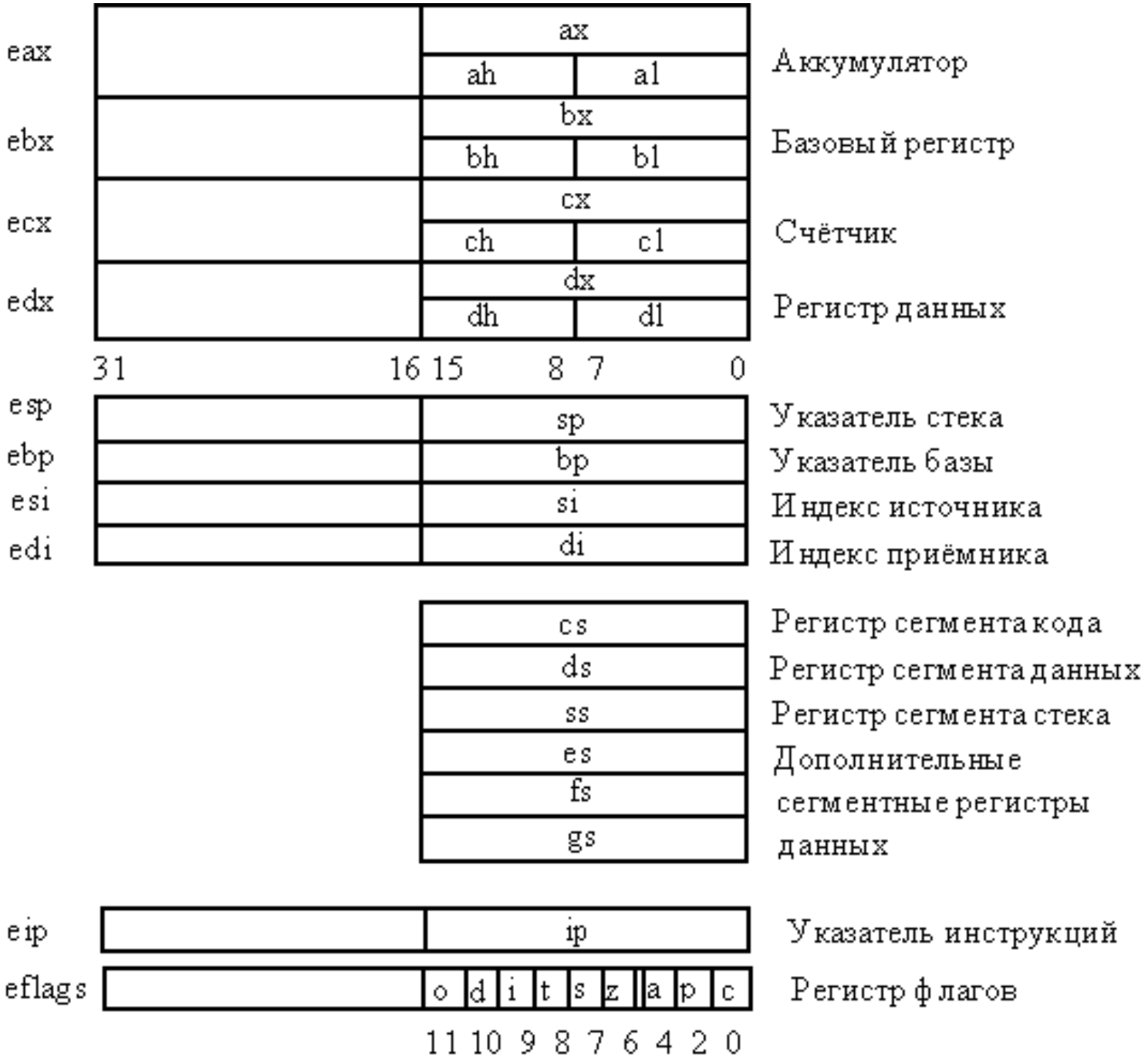
- ***Регистр указателя команд IP***. Регистр указателя команд IP, иначе называемый регистром счетчика команд, имеет размер 16 бит и хранит адрес некоторой ячейки памяти – начало следующей команды. Микропроцессор использует регистр IP совместно с регистром CS для формирования 20-битового физического адреса очередной выполняемой команды, при этом регистр CS задает сегмент выполняемой программы, а IP – смещение от начала сегмента. По мере того, как микропроцессор загружает команду из памяти и выполняет ее, регистр IP увеличивается на число байт в команде. Для непосредственного изменения содержимого регистра IP служат команды перехода.
- ***Регистр флагов***. Флаги – это отдельные биты, принимающие значение 0 или 1. Регистр флагов (признаков) содержит девять активных битов (из 16). Каждый бит данного регистра имеет особое значение, некоторые из этих бит содержат код условия, установленный последней выполненной командой. Другие биты показывают текущее состояние микропроцессора.


16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OF DF SF ZF AF PF CF
OF (Overflow Flag)
DF (Direction Flag)
SF (Sign Flag)
ZF (Zero Flag)
AF (Auxiliary Flag)
PF (Parity Flag)
CF (Carry Flag)

Указанные на рисунке флаги наиболее часто используются в прикладных программах и сигнализируют о следующих событиях:

- OF (флаг переполнения) фиксирует ситуацию переполнения, то есть выход результата арифметической операции за пределы допустимого диапазона значений;
- DF (флаг направления) используется командами обработки строк. Если DF = 0, строка обрабатывается в прямом направлении, от меньших адресов к большим. Если DF = 1, обработка строк ведется в обратном направлении;
- SF (флаг знака) показывает знак результата операции, при отрицательном результате SF = 1;
- ZF (флаг нуля) устанавливается в 1, если результат операции равен 0;
- AF (флаг вспомогательного переноса) используется в операциях над упакованными двоично-десятичными числами. Этот флаг служит индикатором переноса или заема из старшей тетрады (бит 3);
- PF (флаг четности) устанавливается в 1, если результат операции содержит четное число двоичных единиц;
- CF (флаг переноса) показывает, был ли перенос или заем при выполнении арифметических операций.

Легко заметить, что все флаги младшего байта регистра флагов устанавливаются арифметическими или логическими операциями процессора. За исключением флага переполнения, все флаги старшего байта отражают состояние микропроцессора и влияют на характер выполнения программы. Флаги старшего байта устанавливаются и сбрасываются специально предназначенными для этого командами. Флаги младшего байта используются командами условного перехода для изменения порядка выполнения программы.


eax
ax
ah
al
Аккумулятор
ebx
bx
bh
bl
Базовый регистр
ecx
cx
ch
cl
Счётчик
edx
dx
dh
dl
Регистр данных
31
16 15
8 7
0
esp
sp
Указатель стека
ebp
bp
Указатель базы
esi
si
Индекс источника
edi
di
Индекс приёмника
cs
Регистр сегмента кода
ds
Регистр сегмента данных
ss
Регистр сегмента стека
es
Дополнительные
fs
сегментные регистры
gs
данных
eip
ip
Указатель инструкций
eflags
o d i t s z a p c
Регистр флагов
11 10 9 8 7 6 4 2 0

IP (англ. *Instruction Pointer*) — регистр, обозначающий смещение следующей команды относительно кодового сегмента.

IP — 16-битный (младшая часть EIP)

EIP — 32-битный аналог (младшая часть RIP)

RIP — 64-битный аналог

Сегментные регистры — Регистры указывающие на сегменты.

CS (англ. *Code Segment*), DS (англ. *Data Segment*), SS (англ. *Stack Segment*), ES, FS, GS

В реальном режиме работы процессора сегментные регистры содержат адрес начала 64Kb сегмента, смещенный вправо на 4 бита.

В защищенном режиме работы процессора сегментные регистры содержат селектор сегмента памяти, выделенного ОС.

CS — указатель на кодовый сегмент. Связка CS:IP (CS:EIP/CS:RIP — в защищенном/64-битном режиме) указывает на адрес в памяти следующей команды.

Регистры данных — служат для хранения промежуточных вычислений.

RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8 — R15 — 64-битные

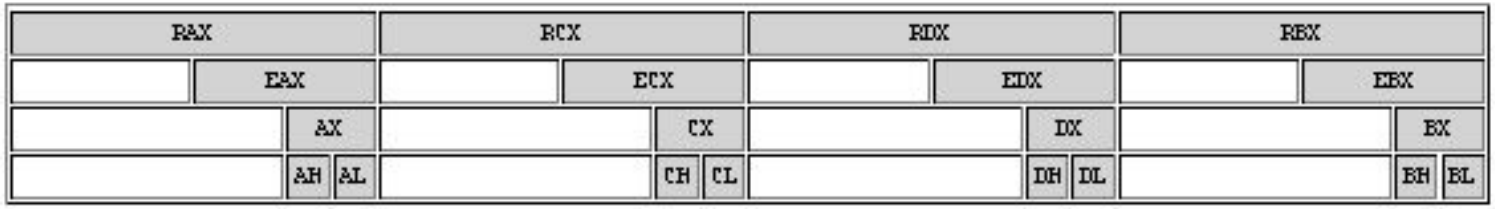
EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, R8D — R15D — 32-битные (extended AX)

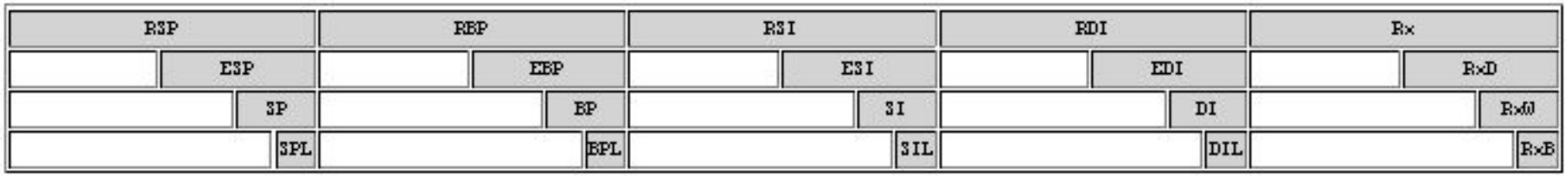
AX, CX, DX, BX, SP, BP, SI, DI, R8W — R15W — 16-битные

AH, AL, CH, CL, DH, DL, BH, BL, SPL, BPL, SIL, DIL, R8B — R15B — 8-битные (половинки 16-ти битных регистров)

например, AH — high AX — старшая половинка 8 бит

AL — low AX — младшая половинка 8 бит





где x — 8..15.

Регистры RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, Rx, RxD, RxW, RxB, SPL, BPL, SIL, DIL доступны только в 64-битном режиме работы процессора.

Регистр флагов FLAGS (16 бит) / EFLAGS (32 бита) / RFLAGS (64 бита) — содержит текущее состояние процессора.

Сегменты, принцип сегментации

Числа, устанавливаемые процессором на адресной шине, являются адресами, то есть номерами ячеек оперативной памяти (ОП). Размер ячейки ОП составляет 8 разрядов, т. е. 1 байт. Поскольку для адресации памяти процессор использует 16-разрядные адресные регистры, то это обеспечивает ему доступ к 65536 (FFFFh) байт или 64К основной памяти. Такой блок непосредственно адресуемой памяти называется сегментом. Любой адрес формируется из адреса сегмента (всегда кратен 16, т.е. начинается с границы параграфа) и адреса ячейки внутри сегмента (этот адрес называется смещением). Для адресации большего объема памяти в процессоре 8086 используется специальная процедура пересчета адресов, называемая вычислением абсолютного (эффективного) адреса.

- Когда процессор выбирает очередную команду на исполнение, в качестве ее адреса используется содержимое, регистра IP. Этот адрес называется исполнительным. Поскольку регистр IP шестнадцатиразрядный, исполнительный адрес тоже содержит 16 двоичных разрядов. Однако адресная шина, соединяющая процессор и память имеет 20 линий связи.
- Чтобы получить 20-битовый адрес, дополнительные 4 бита адресной информации извлекаются из сегментных регистров. Сами сегментные регистры имеют размер в 16 разрядов, а содержащиеся в этих регистрах (CS, DS, SS или ES) 16-битовые значения называются базовым адресом сегмента. Микропроцессор объединяет 16-битовый исполнительный адрес и 16-битовый базовый адрес следующим образом: он расширяет содержимое сегментного регистра (базовый адрес) 4 нулевыми битами (в младших разрядах), делая его 20-битовым (полный адрес сегмента) и прибавляет смещение (исполнительный адрес). При этом 20-битовый результат является физическим или абсолютным адресом ячейки памяти.

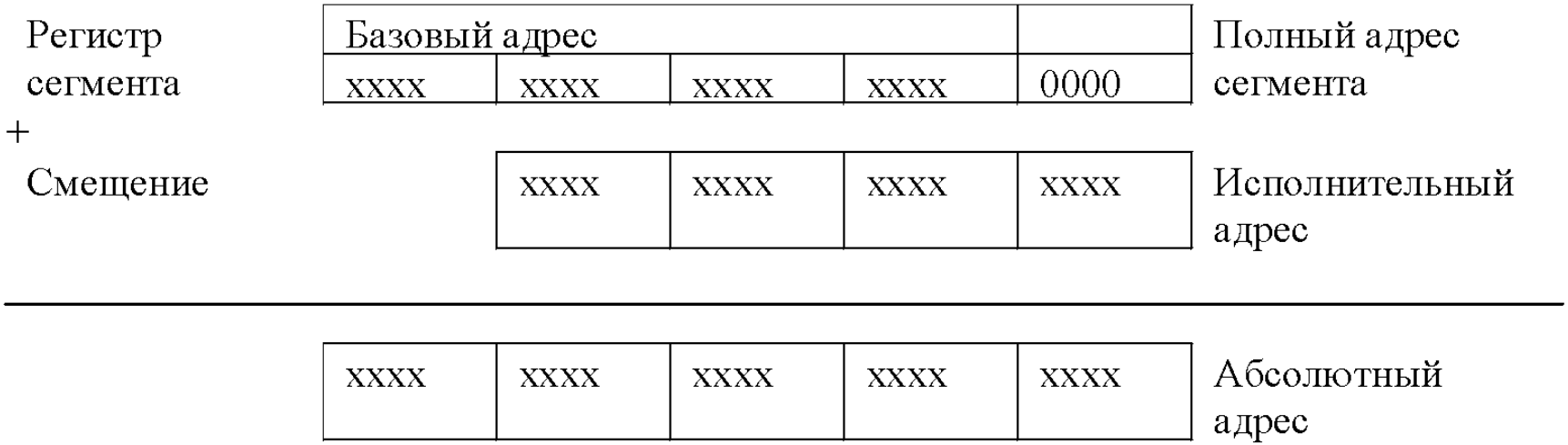


Рис. 1.2. Принцип получения абсолютного адреса.

Существуют три основных типа сегментов:

- сегмент кода – содержит машинные команды, Адресуется регистром CS;
- сегмент данных – содержит данные, то есть константы и рабочие области, необходимые программе. Адресуется регистром DS;
- сегмент стека – содержит адреса возврата в точку вызова подпрограмм. Адресуется регистром SS.

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

<адрес сегмента>:<смещение>

или

<сегментный регистр>:<адресное выражение>

Стек

Во многих случаях программе требуется временно запомнить некоторую информацию. Эта проблема в персональном компьютере решена посредством реализации стека LIFO ("последний пришел - первый ушел"), называемого также стеком включения/извлечения (stack). Стек – это область памяти для временного хранения данных, в которую по специальным командам можно записывать отдельные слова (но не байты); при этом для запоминания данных достаточно выполнить лишь одну команду и не нужно беспокоиться о выборе соответствующего адреса: процессор автоматически выделяет для них свободное место в области временного хранения. Наиболее важное использование стека связано с подпрограммами, в этом случае стек содержит адрес возврата из подпрограммы, а иногда и передаваемые в/из подпрограмму данные. Стек обычно рассчитан на косвенную адресацию через регистр указатель стека. При включении элементов в стек производится автоматический декремент указателя стека, а при извлечении – инкремент, то есть стек всегда «растет» в сторону меньших адресов памяти. Адрес последнего включенного в стек элемента называется вершиной стека (TOS), а адрес сегмента стека – базой стека.

Адресация данных

- Для четкого понимания того, как осуществляется адресация данных, проанализируем способ образования адреса операнда. Адрес операнда формируется по схеме сегментх:смещение. Селектор сегмента можно указать явным или неявным образом. Обычно селектор сегмента загружается в сегментный регистр, а сам регистр выбирается в зависимости от типа выполняемой операции, как показано ниже.
- Процессор автоматически выбирает сегмент в соответствии с условиями, описанными в таблице. При сохранении операнда в памяти или загрузке из памяти в качестве сегментного регистра по умолчанию используется DS, но можно и явным образом указать сегментный регистр, применяемый в операции.
- Пусть, например, требуется сохранить содержимое регистра EAX в памяти, адресуемой сегментным регистром ES и смещением, находящимся в регистре EBX. В этом случае можно использовать команду
- mov ES:[EBX],EAX
- Обратите внимание на то, что после имени сегмента указывается символ двоеточия.

Критерии выбора сегментного регистра

Тип операции	Регистр сегмента	Сегмент	Описание
Команды процессора	CS	Программный сегмент	Используется при вызовах команд процессора
Обращение к стеку	SS	Сегмент стека	Используется во всех операциях со стеком, а также в операциях с памятью, в которых базовыми являются регистры ESP и EBP
Локальные данные	DS	Сегмент данных	Используется во всех операциях с данными, исключая те, в которых применяется стек или строка-приемник при выполнении строковых операций
Строки-приемники	ES	Сегмент данных, адресуемый регистром ES	Операнд-приемник в строковых операциях

Режимы адресации

В зависимости от местоположения источников образования полного (абсолютного) адреса в языке ассемблера различают следующие способы адресации операндов:

- **регистровая;**
- **прямая;**
- **непосредственная;**
- **косвенная;**
- **базовая;**
- **индексная;**
- **базово-индексная**.

Регистровая адресация

Регистровая адресация подразумевает использование в качестве операнда регистра процессора, например:

- PUSH DS
- MOV BP,SP

Прямая адресация

При прямой адресации один операнд представляет собой адрес памяти, второй – регистр:

- MOV Data,AX

Непосредственная адресация

Непосредственная адресация применяется, когда операнд, длинной в байт или слово находится в ассемблерной команде:

- MOV AH,4CH

Косвенная адресация

При использовании косвенной адресации абсолютный адрес формируется исходя из сегментного адреса в одном из сегментных регистров и смещения в регистрах BX, BP, SI или DI:

- MOV AL,[BX] ;База – в DS, смещение – в BX
- MOV AX,[BP] ;База – в SS, смещение – в BP
- MOV AX,ES:[SI] ;База – в ES, смещение – в SI

Базовая адресация

В случае применения базовой адресации исполнительный адрес является суммой значения смещения и содержимого регистра BP или BX, например:

- MOV AX,[BP+6] ;База – SS, смещение – BP+6
- MOV DX,8[BX] ;База – DS, смещение – BX+8

Индексная адресация

При индексной адресации исполнительный адрес определяется как сумма значения указанного смещения и содержимого регистра SI или DI так же, как и при базовой адресации, например:

- MOV DX,[SI+5] ;База – DS, смещение – SI+5

Базово-индексная адресация

Базово-индексная адресация подразумевает использование для вычисления исполнительного адреса суммы содержимого базового регистра и индексного регистра, а также смещения, находящегося в операторе, например:

- MOV BX,[BP][SI] ;База – SS, смещение – BP+SI
- MOV ES:[BX+DI],AX ;База – ES, смещение – BX+DI
- MOV AX,[BP+6+DI] ;База – SS, смещение - BP+6+DI

```
//
#include<iostream.h>
 int dddS,cccS,aaaS;
extern "C" {void       Lab3S(void);}


void F_C(void)
{
 dddS=aaaS+cccS;
  cout<<"c:";
  cout<<"          ddS="<<dddS<<endl;
}


void F_ASM(void)
{
   Lab3S();
   cout<<«ASM:";
   cout<<"          dddS="<<dddS<<endl;

}


void main(void)
{
  cout<<"Input aaaS"; cin>>aaaS;
  cout<<"Input cccS"; cin>>cccS;
  F_C();
  F_ASM();
}
```

```
 .MODEL Large,C
        .data
        Extrn   aaaS:byte,cccS:byte,dddS:byte
        .code
        Public  Lab3S
Lab3S   proc    far
        mov al,aaaS
        add al,cccS
        mov dddS,al
        ret
Lab3S   endp
        end
```

Организация программы Сегменты

Программа состоит из одного или нескольких сегментов. Обычно область памяти, в которой находятся команды, называют сегментом кода, область памяти с данными — сегментом данных и область памяти, отведенную под стек, — сегментом стека. Ассемблер позволяет изменять устройство программы как угодно — помещать данные в сегмент кода, разносить код на множество сегментов, помещать стек в один сегмент с данными или вообще использовать один сегмент для всего.

Сегмент программы описывается директивами SEGMENT и ENDS.

имя_сегмента segment readonly выравн. тип разряд 'класс‘

……

...имя_сегмента ends

Имя сегмента — метка, которая будет использоваться для получения сегментного адреса, а также для комбинирования сегментов в группы.

Все пять операндов директивы SEGMENT необязательны.

- **READONLY**. Если этот операнд присутствует, MASM выдаст сообщение об ошибке на все команды, выполняющие запись в данный сегмент. Другие ассемблеры этот операнд игнорируют.
- **Выравнивание**. Указывает ассемблеру и компоновщику, с какого адреса может начинаться сегмент. Значения этого операнда:
- BYTE — с любого адреса;
- WORD — с четного адреса;
- DWORD — с адреса, кратного 4;
- PARA — с адреса, кратного 16 (граница параграфа);
- PAGE — с адреса, кратного 256.
- По умолчанию используется выравнивание по границе параграфа.

- **Тип**. Выбирает один из возможных типов комбинирования сегментов:
- тип PUBLIC (иногда используется синоним MEMORY) означает, что все такие сегменты с одинаковым именем, но разными классами будут объединены в один;
- тип STACK — то же самое, что и PUBLIC, но должен использоваться для сегментов стека, потому что при загрузке программы сегмент, полученный объединением всех сегментов типа STACK, будет использоваться как стек;
- сегменты типа COMMON с одинаковым именем также объединяются в один, но не последовательно, а по одному и тому же адресу, следовательно, длина суммарного сегмента будет равна не сумме длин объединяемых сегментов, как в случае PUBLIC и STACK, а длине максимального. Таким способом иногда можно формировать оверлейные программы;
- тип AT — выражение указывает, что сегмент должен располагаться по фиксированному абсолютному адресу в памяти. Результат выражения, использующегося в качестве операнда для AT, равен этому адресу, деленному на 16. Например: segment at 40h — сегмент, начинающийся по абсолютному адресу 0400h. Такие сегменты обычно содержат только метки, указывающие на области памяти, которые могут потребоваться программе;
- PRIVATE (значение по умолчанию) — сегмент такого типа не объединяется с другими сегментами.

- **Разрядность**. Этот операнд может принимать значения USE16 и USE32. Размер сегмента, описанного как USE16, не может превышать 64 Кб, и все команды и адреса в этом сегменте считаются 16-битными. В этих сегментах все равно можно применять команды, использующие 32-битные регистры или ссылающиеся на данные в 32-битных сегментах, но они будут использовать префикс изменения разрядности операнда или адреса и окажутся длиннее и медленнее. Сегменты USE32 могут занимать до 4 Гб, и все команды и адреса в них по умолчанию 32-битные. Если разрядность сегмента не указана, по умолчанию используется USE16 при условии, что перед директивой .MODEL не применялась директива задания допустимого набора команд .386 или старше.
- **Класс сегмента** — это любая метка, взятая в одинарные кавычки. Все сегменты с одинаковым классом, даже сегменты типа PRIVATE, будут расположены в исполняемом файле непосредственно друг за другом.

- Для обращения к любому сегменту следует сначала загрузить его сегментный адрес (или селектор в защищенном режиме) в какой-нибудь сегментный регистр. Если в программе определено много сегментов, удобно объединить несколько сегментов в группу, адресуемую с помощью одного сегментного регистра:
- имя_группы group имя_сегмента...
- Операнды этой директивы — список имен сегментов (или выражений, использующих оператор SEG), которые объединяются в группу. Имя группы теперь можно применять вместо имен сегментов для получения сегментного адреса и для директивы ASSUME.
- assume регистр:связь...
- Директива ASSUME указывает ассемблеру, с каким сегментом или группой сегментов связан тот или иной сегментный регистр. В качестве операнда «связь» могут использоваться имена сегментов, имена групп, выражения с оператором SEG или слово «NOTHING», означающее отмену действия предыдущей ASSUME для данного регистра. Эта директива не изменяет значений сегментных регистров, а только позволяет ассемблеру проверять допустимость ссылок и самостоятельно вставлять при необходимости префиксы переопределения сегментов, если они необходимы.

Перечисленные директивы удобны для создания больших программ на ассемблере, состоящих из разнообразных модулей и содержащих множество сегментов. В повседневном программировании обычно используется ограниченный набор простых вариантов организации программы, известных как модели памяти.

Модели памяти и упрощенные директивы определения сегментов

- Модели памяти задаются директивой .MODEL
- **.model модель,язык,модификатор**
- где модель — одно из следующих слов:
- TINY — код, данные и стек размещаются в одном и том же сегменте размером до 64 Кб. Эта модель памяти чаще всего используется при написании на ассемблере небольших программ;
- SMALL — код размещается в одном сегменте, а данные и стек — в другом (для их описания могут применяться разные сегменты, но объединенные в одну группу). Эту модель памяти также удобно использовать для создания программ на ассемблере;
- COMPACT — код размещается в одном сегменте, а для хранения данных могут использоваться несколько сегментов, так что для обращения к данным требуется указывать сегмент и смещение (данные дальнего типа);
- MEDIUM — код размещается в нескольких сегментах, а все данные — в одном, поэтому для доступа к данным используется только смещение, а вызовы подпрограмм применяют команды дальнего вызова процедуры;
- LARGE и HUGE — и код, и данные могут занимать несколько сегментов;
- FLAT — то же, что и TINY, но используются 32-битные сегменты, так что максимальный размер сегмента, содержащего и данные, и код, и стек, — 4 Мб.

- **Язык** — необязательный операнд, принимающий значения C, PASCAL, BASIC, FORTRAN, SYSCALL и STDCALL. Если он указан, подразумевается, что процедуры рассчитаны на вызов из программ на соответствующем языке высокого уровня, следовательно, если указан язык C, все имена ассемблерных процедур, объявленных как PUBLIC, будут изменены так, чтобы начинаться с символа подчеркивания, как это принято в C.
- **Модификатор** — необязательный операнд, принимающий значения NEARSTACK (по умолчанию) или FARSTACK. Во втором случае сегмент стека не будет объединяться в одну группу с сегментами данных.

После того как модель памяти установлена, вступают в силу упрощенные директивы определения сегментов, объединяющие действия директив SEGMENT и ASSUME.

Кроме того, сегменты, объявленные упрощенными директивами, не требуется закрывать директивой ENDS — они закрываются автоматически, как только ассемблер обнаруживает новую директиву определения сегмента или конец программы.

- Директива **.CODE** описывает основной сегмент кода

.code имя_сегмента

эквивалентно

_TEXT segment word public 'CODE'

для моделей TINY, SMALL и COMPACT и

name_TEXT segment word public 'CODE'

для моделей MEDIUM, HUGE и LARGE (name — имя модуля, в котором описан данный сегмент).

Директива **.STACK**

.stack размер

Директива .STACK описывает сегмент стека и эквивалентна директиве

STACK segment para public 'stack'.

- Директива **.data** описывает обычный сегмент данных и соответствует директиве

_DATA segment word public 'DATA'

.data?

Описывает сегмент неинициализированных данных:

- _BSS segment word public 'BSS'

- .const Описывает сегмент неизменяемых данных:
- CONST segment word public ’CONST’
- В некоторых операционных системах этот сегмент будет загружен так, что попытка записи в него может привести к ошибке.
- .fardata имя_сегмента
- Сегмент дальних данных:
- имя_сегмента segment para private ’FAR_DATA’
- Доступ к данным, описанным в этом сегменте, потребует загрузки сегментного регистра. Если не указан операнд, в качестве имени сегмента используется FAR_DATA.
- .fardata? имя_сегмента
- Сегмент дальних неинициализированных данных:
- имя_сегмента segment para private ’FAR_BSS’
- Как и в случае с FARDATA, доступ к данным из этого сегмента потребует загрузки сегментного регистра. Если имя сегмента не указано, используется FAR_BSS.

Директивы задания набора допустимых команд

- По умолчанию ассемблеры используют набор команд процессора 8086 и выдают сообщения об ошибках, если выбирается команда, которую этот процессор не поддерживал. Для того чтобы ассемблер разрешил использование команд, появившихся в более новых процессорах, и команд расширений, предлагаются следующие директивы:
- .8086 — используется по умолчанию. Разрешены только команды 8086;
- .186 — разрешены команды 80186;
- .286 и .286c — разрешены непривилегированные команды 80286;
- .286p — разрешены все команды 80286;
- .386 и .386c — разрешены непривилегированные команды 80386;
- .386p — разрешены все команды 80386;
- .486 и .486c — разрешены непривилегированные команды 80486;
- .486p — разрешены все команды 80486;
- .586 и .586c — разрешены непривилегированные команды P5 (Pentium);
- .586p — разрешены все команды P5 (Pentium);
- .686 — разрешены непривилегированные команды P6 (Pentium Pro, Pentium II);
- .686p — разрешены все команды P6 (Pentium Pro, Pentium II);
- .8087 — разрешены команды NPX 8087;
- .287 — разрешены команды NPX 80287;
- .387 — разрешены команды NPX 80387;
- .487 — разрешены команды FPU 80486;
- .587 — разрешены команды FPU 80586;
- .MMX — разрешены команды IA MMX;
- .K3D — разрешены команды AMD 3D.
- Не все ассемблеры поддерживают каждую директиву, например MASM и WASM не

Структура программы на языке Ассемблера

Исходный программный модуль – это последовательность предложений. Различают два типа предложений:

- инструкции процессора
- директивы ассемблера.

Инструкции управляют работой процессора, а директивы указывают ассемблеру и редактору связей, каким образом следует объединять, инструкции для создания модуля, который и станет работающей программой.

Инструкция процессора на языке ассемблера состоит не более чем из четырех полей и имеет следующий формат:

[[метка:]] мнемоника [[операнды]] [[;комментарии]]

Единственное обязательное поле – поле кода операции (мнемоника), определяющее инструкцию, которую должен выполнить микропроцессор.

Поле операндов определяется кодом операции и содержит дополнительную информацию о команде. Каждому коду операции соответствует определенное число операндов.

Метка служит для обозначения какого-то определенного места в памяти, т. е. содержит в символическом виде адрес, по которому храниться инструкция. Преобразование символических имен в действительные адреса осуществляется программой ассемблера.

Часть строки исходного текста после символа «;» (если он не является элементом знаковой константы или строки знаков) считается комментарием и ассемблером игнорируется.

Пример:

Метка Код операции Операнды ;Комментарий

MET: MOVE AX, BX ;Пересылка

Структура директивы аналогична структуре инструкции.

Программа типа COM

Программа, выводящая на экран текст «Hello world!».

```
; hello-l.asm
; Выводит на экран сообщение "Hello World!" и завершается
        .model    tiny              ; модель памяти, используемая для COM
        .code                       ; начало сегмента кода
         org      100h              ; начальное значение счетчика - 100h
start:  mov       ah,9                      ; номер функции DOS - в AH
         mov      dx,offset message         ; адрес строки - в DX
         int      21h                       ; вызов системной функции DOS
         ret                                ; завершение COM-программы
message db        "Hello World!",0Dh,0Ah,'$'        ; строка для вывода
        end       start                     ; конец программы
```

Компиляция

Для TASM:

tasm hello-1.asm

Для MASM:

ml /c hello-1.asm

Компоновка

Для TLINK:

tasm /t /x hello-1.obj

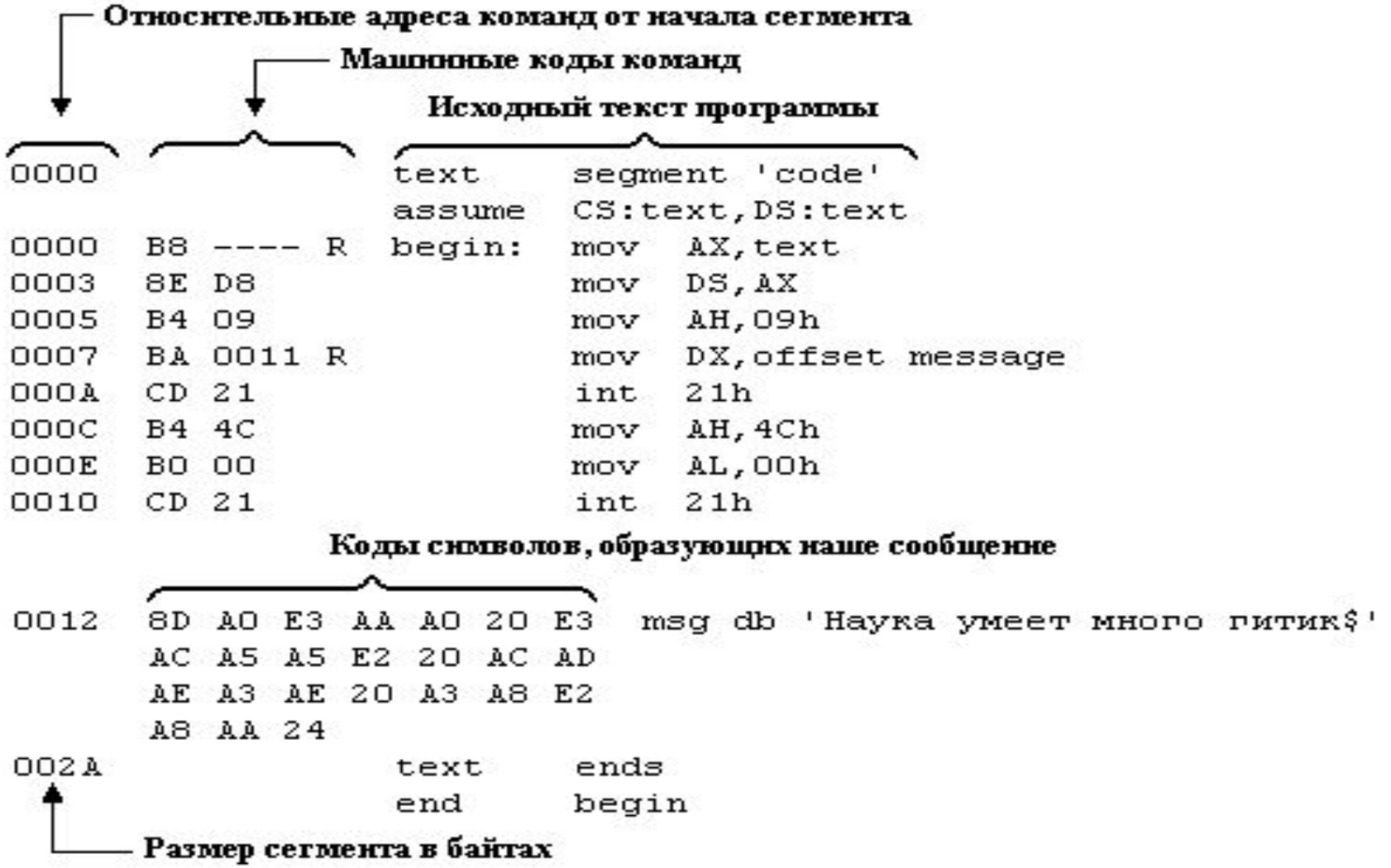
Для MASM:

Link hello-1.obj,,NUL,,,

exe2bin hello-1.exe hello-1.com

Теперь получился файл HELLO-1.COM . Если его выполнить, на экране появится строка «Hello World!» и программа завершится.

Листинг трансляции



- Команды программы имеют различную длину и располагаются в памяти вплотную друг к другу. Так, первая команда mov AX,text, начинающаяся с байта 0000 сегмента, занимает 3 байта. Соответственно, вторая команда начинается с байта 0003. Вторая команда имеет длину 2 байта, поэтому третья команда начинается с байта 0005 и т.д.

- Предложения программы с операторами segment, assume, end не транслируются в какие-либо машинные коды и не находят отражения в памяти. Они нужны лишь для передачи транслятору служебной информации.

- Транслятор не мог полностью определить код команды mov AX,text. В этой команде в регистр AX засылается адрес сегмента text. Однако этот адрес станет известен лишь в процессе загрузки выполнимого файла программы в память. По этому в листинге на месте этого адреса стоит прочерк.

- Текст, введенный в программу, также оттранслировался: вместо символов текста в загрузочный файл попадут коды ASCII этих символов.

Директивы инициализации и описания данных

- Данные могут размещаться в участках памяти, которые называются *сегменты.* Обычно это или *сегмент данных*, или *сегмент кода.* Сегменты описываются с помощью директивы A **SEGMENT** или с помощью упрощенных директив **.Model, .Code** или **.Data.**
- Для инициализации **простых** типов данных в Ассемблере используются специальные директивы **Dx,** являющиеся указаниями компилятору на выделение определенных объемов памяти. Для языка Ассемблера имеет значение только *длина ячейки*, в которой размещено данное, а какое это данное — зависит всецело от человека, пишущего программу его обработки.

Директивы для задания простых типов данных.

Длина (бит)	Директива	Описание
8	DB	BYTE
16	DW	WORD
32	DD	DWORD
64	DQ	QWORD
80	DT	TBYTE

- Мнемокоды директив инициализации данных **Dx** означают следующее:
- **DB (Define Byte)** — определить байт.
- **DW (Define Word)** — определить слово.
- **DD (Define Double Word)** — определить двойное слово.
- **DQ (Define Quarter Word)** — определить учетверенное слово.
- **DT (Define Ten Bytes)** — определить 10 байтов.
- Для директив инициализации данных **имя** может быть, а может и отсутствовать. Если имя есть, с ним связывается *адрес памяти,* и в дальнейшем в командах мы можем использовать это имя по своему усмотрению. Например, заносить по указанному именному адресу какое-то значение или извлекать из него хранимое значение.
- Регистр букв для имен и директив в Ассемблере безразличен
- **Но при стыковке программ на Ассемблере с программами на языке** C/C++ **регистр для имен переменных имеет ОЧЕНЬ большое значение**.

Если переменная не инициализируется, то в поле операнда директивы Dx нужно поставить знак вопроса. А если нужно выделить непрерывный участок памяти из нескольких ячеек, пишется параметр коэффициента повторений dup.

Pascal	C/C++	Assembler

N: Integer;	IntN; //Win16
	// Win32
		N dw ? N dd ?

A: single;	float A;	A dd ?
B: double; ... B :=-898.6897;	double B = -898.6897;	B DQ -898.6897
Arr:array [1..100J of extended;	long double Arr[100];	Arr DT 100dup(?)
Mas:array [1 ..10] of byte; Mas [1]:=1; Mas [2]:=2;	unsigned char Mas Q = {1,2,3,4,5,6,7,8,9,10};	Mas DB 1.2,3,4 DB 5,6,7,8,9,10
Mas [9]:=9; Mas [10]:=10;		
MasW:array [1 ..10] of word; Mas [1]:=0; Mas [2]:=0;	unsigned short int MasW Q = {0,0,0,0,0,0,0,0,0,0};	MasW DW 10 dup(0)
Mas [9]:=0; Mas [10]:=0;		

Арифметические команды

- Все арифметические команды устанавливают флаги CF, AF, SF, ZF, OF и PF в зависимости от результат операции.
- Двоичные числа могут иметь длину 8, 16 и 32 бит. Значение старшего (самого левого бита) задает знак числа: 0 – положительное, 1 – отрицательное. Отрицательные числа представляются в так называемом дополнительном коде, в котором для получения отрицательного числа необходимо инвертировать все биты положительного числа и прибавить к нему 1.

Пример

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

Команды сложения **ADD, ADC, INC**

- Командам **БЕЗРАЗЛИЧНО какие числа складываются (знаковые или нет).**
- Если в результате сложения результат НЕ поместился в отведенное место, устанавливается флаг переноса CF=1. Команда ADC как раз и реагирует на этот флаг. Вырабатываются еще 4 флага: PF, SF, ZF, OF

Состояние флагов после выполнения команд ADD, ADC, INC

Флаг	Пояснение
CF=1	Результат сложения НЕ поместился в операнде-приемнике
PF=1	Результат сложения имеет четное число бит со значением 1
SF=J	Копируется СТАРШИЙ (ЗНАКОВЫЙ) бит результата сложения
ZF=1	Результат сложения равен НУЛЮ
OF=I	Если при сложении двух чисел ОДНОГО знака (ОБА положительные или ОБА отрицательные) результат сложения получился БОЛЬШЕ допустимого значения. В этом случае приемник МЕНЯЕТ ЗНАК.

Команда ADD

ADD (ADDition — сложение).

Синтаксис:

ADD *Приемник, Источник*

Логика работы:

<Приемник> = < Приемник> + <Источник>

Команда ADC - сложение с переносом

Эта команда от команды ADD отличается использованием бита переноса CF при сложении (ADdition with Carry). Поэтому она может использоваться при сложении 32-разрядных чисел.

Синтаксис:

ADC Приемник, Источник

Логика работы:

<Приемник> = <Приемник> + <Источник> + <CF>

Обычно эта команда работает в паре с командой ADD(складывание младших частей числа).

```
          .MODEL Large, Pascal
          ;x=a+b    x,a,b:LongInt
          .data
          Extrn    x:Dword,a:Dword,b:Dword
          .code
          Public   addaL
addaL     proc     far
          mov       ax,WORD PTR a       ; ax <=== мл. часть a
          mov       bx,WORD PTR a+2     ; bx <=== ст. часть a
          mov       cx,WORD PTR b       ; cx <=== мл. часть b
          mov       dx,WORD PTR b+2     ; dx <=== ст. часть b
          add       ax,cx   ; <ax>:=<ax>+<cx>          мл.часть
          adc       bx,dx   ; <bx>:=<bx>+<dx>+<CF>   ст.часть
          mov       WORD PTR x,ax   ; мл. часть x <=== <ax>
          mov       WORD PTR x+2,bx; ст. часть x <=== <bx>
          ret
addaL     endp
```

Команда INC

- Мнемокод этой команды получен в результате сокращения такого предложения: INCrement operand by 1 — Увеличение значения операнда на 1. Команда содержит один **операнд** и имеет следующий

синтаксис:

- **INC *Операнд***

Логика работы команды:

- **< Операнд > = < Операнд > + 1**
- В качестве операнда допустимы регистры и память: R8, R16, Мет8, Мет16.

Команды вычитания SUB, SBB, DEC и NEG

- Команды вычитания SUB, SBB, DEC ОБРАТНЫ соответствующим командам сложения ADD, ADC и INC. Они имеют те же самые операнды.
- SUB (SUBtract — Вычитание).
- SBB (SuBtract with Borrow CF — Вычитание с заемом флага переноса CF).
- DEC (DECrement operand by 1 — Уменьшение значения операнда на 1).

```
title subaL
;x=a-b
datasegment para public
Extrn x:Dword,a:Dword,b:Dword
dataEnds
code      segment para public
assume cs:code,ds:data
Public suba
suba      procfar
movax,WORD PTR a
movbx,WORD PTR a+2
movcx,WORD PTR b
movdx, WORD PTR b+2
sub ax,cx
sbb bx,dx
movWORD PTR x,ax
movWORD PTR x+2,bx
ret
suba      endp
code      ends
end
```

```
bigval_1        dd      0,0,0                   ; 96-битное число
bigval_2        dd      0,0,0
bigval_3        dd      0,0,0

; сложение 96-битных чисел bigval_1 и bigval_2
        mov     eax,dword ptr bigval_1
        add     eax,dword ptr bigval_2      ; сложить младшие двойные слова
        mov     dword ptr bigval_3,eax
        mov     eax,dword ptr bigval_1[4]
        adc     eax,dword ptr bigval_2[4] ; сложить средние двойные слова
        mov     dword ptr bigval_3[4],eax
        mov     eax,dword ptr bigval_1[8]
        adc     eax,dword ptr bigval_2[8] ; сложить старшие двойные слова
        mov     dword ptr bigval_3[8],eax

; вычитание 96-битных чисел bigval_1 и bigval_2
        mov     eax,dword ptr bigval_1
        sub     eax,dword ptr bigval_2      ; вычесть младшие двойные слова
        mov     dword ptr bigval_3,eax
        mov     eax,dword ptr bigval_1[4]
        sbb     eax,dword ptr bigval_2[4] ; вычесть средние двойные слова
        mov     dword ptr bigval_3[4],eax
        mov     eax,dword ptr bigval_1[8]
        sbb     eax,dword ptr bigval_2[8] ; вычесть старшие двойные слова
        mov     dword ptr bigval_3[8],eax
```

Команда NEG

- Команда NEG (NEGate operang — изменение знака операнда).
- Синтаксис:
- NEG *Операнд*
- Логика работы команды:
- < *Операнд* > = — < *Операнд* >
- В качестве операнда допустимы регистры и память: R8, R16, Мем8, Мем16

Команды умножения MUL и IMUL

- Это одни из САМЫХ НЕПРИЯТНЫХ команд целочисленной арифметики из-за наличия ***неявных*** **операндов**.
- **MUL** (MULtiply - БЕЗЗНАКОВОЕ умножение)
- **IMUL** (Integer MULtiply - ЗНАКОВОЕ целочисленное умножение).
- **Команды учитывают наличие ЗНАКА.**
- Синтаксис:
- **MUL** Источник
- **IMUL** Источник
- Логика работы команд:
- <Произведение> = <Множимоё> * <**Множитель>**
- В качестве *Множителя* допустимы регистры и память: R8, R16, Мем8, Мем1б. **Константы НЕ допускаются!!!**
- *Множимое* и *Произведение* находятся в **СТРОГО ОПРЕДЕЛЕННОМ МЕСТЕ** в зависимости от **длины *Множителя.***

Неявные операнды команд MUL, IMUL

Длина источника (Множителя)	Множимое	Произведение
Byte	AL	AX (<AH:AL>)
Word	AX	<DX:AX>

ВЫВОДЫ

- Команда IMUL реагирует на ЗНАК перемножаемых чисел.
- Расположение *Множимого* и *Произведения* **— строго определенное** и зависит од ДЛИНЫ *Множителя* — это нужно просто ЗАПОМНИТЬ.
- Длина *Произведения* всегда в ДВА раза больше, чем у *Множителя.* Причем старшая часть *Произведения* находится либо в регистре AH, либо в DX. **Именно об** ЭТОМ **обстоятельстве** ЗАБЫВАЮТ **как люди, так и компиляторы** (их ведь тоже делали люди!)
- Эти команды тоже вырабатывают флаги. Устанавливаются флаги CF=1 и OF=1, если результат слишком велик для отведенных ему регистров назначения.

```
            title MULword
            ;z=5*w
data        segment para public
            Extrn    w:word,z:Dword
data        Ends
code        segment para public
   assume    cs:code,ds:data
   Public    MULword
MULword proc      far
            mov       cx,5       ; Множитель 5 ===> CX
            mov       AX,w       ; Множимое w ====> AX
            MUL       cx          ; <DX:AX> = <AX>*<CX>
            mov       WORD PTR z,ax     ; Младшая часть
            mov       WORD PTR z+2,DX ; Старшая часть
            ret
MULword endp
code           ends
            end
```

Умножение больших чисел

- Чтобы умножить большие числа, придется вспомнить правила умножения десятичных чисел в столбик: множимое умножают на каждую цифру множителя, сдвигают влево на соответствующее число разрядов и затем складывают полученные результаты. В нашем случае роль цифр будут играть байты, слова или двойные слова, а сложение должно выполняться по правилам сложения чисел повышенной точности. Алгоритм умножения оказывается заметно сложнее, поэтому умножим для примера только 64-битные числа:

```
; беззнаковое умножение двух 64-битных чисел (X и Y) и сохранение
; результата в 128-битное число Z
        mov         eax,dword ptr X
        mov         ebx,eax
        mul         dword ptr Y          ; перемножить младшие двойные слова
        mov         dword ptr Z,eax      ; сохранить младшее слово произведения
        mov         ecx,edx              ; сохранить старшее двойное слово
        mov         eax,ebx              ; младшее слово "X" в eax
        mul         dword ptr Y[4]       ; умножить младшее слово на старшее
        add         eax,ecx
        adc         edx,0                ; добавить перенос
        mov         ebx,eax              ; сохранить частичное произведение
        mov         ecx,edx
        mov         eax,dword ptr X[4]
        mul         dword ptr Y          ; умножить старшее слово на младшее
        add         eax,ebx              ; сложить с частичным произведением
        mov         dword ptr Z[4],eax
        adc         ecx,edx
        mov         eax,dword ptr X[4]
        mul         dword ptr Y[4]       ; умножить старшие слова
        add         eax,ecx              ; сложить с частичным произведением
        adc         edx,0                ; и добавить перенос
        mov         word ptr Z[8],eax
        mov         word ptr Z[12],edx
```

Чтобы выполнить умножение со знаком, потребуется сначала определить знаки множителей, изменить знаки отрицательных множителей, выполнить обычное умножение и изменить знак результата, если знаки множителей были разными.

Команды деления DIV и IDIV

- **DIV** (DIVide - БЕЗЗНАКОВОЕ деление),
- **IDIV** (Integer Divide - ЗНАКОВОЕ деление целых чисел).

Синтаксис:

- **DIV *Источник* IDIV *Источник***

Логика работы команд:

- *< Частное: Остаток>* = *<Делимоё>* / **<Делитель>**
- В качестве *Делителя* допустимы регистры и память: **R8, R16,** Мем8, Мем**16. Константы НЕ допускаются!!!**
- Где же находятся *Делимое* и *<Частное:Остаток>*
- Опять источник ошибок — НЕЯВНЫЕ операнды. Они находятся тоже в СТРОГО ОПРЕДЕЛЕННОМ МЕСТЕ в зависимости от **длины Делителя.**

Неявные операнды команд DIV, IDIV

Длина источника (Делителя)	Делимое	Результат	
		Частное	Остаток
Byte	AX (<AH:AL>)	AL	AH
Word	<DX: AX>	AX	DX

Выводы

- Команда IDIV реагирует на ЗНАК обрабатываемых чисел.
- Расположение *Делимого* и *Результата* — **строго определенное** и зависсит от ДЛИНЫ Делителя.
- *Результат* состоит из *Частного* и *Остатка*, которым при обычных целочисленных вычислениях пренебрегают.
- • Длина *Делимого* всегда в ДВА раза больше, чем у *Делителя.* Причем старшая часть *Делимого* находится либо в регистре AH, либо в DX.
- Эти команды тоже вырабатывают флаги. Но устанавливаются флаги CF=1 и OF**=I,** если частное НЕ помещается в регистры AL или AX.
- Здесь, в отличие от команд умножения, **может генерироваться** ПРЕРЫВАНИЕ **"Деление на ноль”.**

```
            title divword
            ;z=5*w/4
data        segment para public
            Extrn    w:word,z:word
data        Ends
code        segment para public
   assume    cs:code,ds:data
   Public    divword
divword proc      far
            mov       cx,5       ; Множитель 5 ===> CX
            mov       AX,w       ; Множимое w ====> AX
            MUL       cx          ; <DX:AX> = <AX>*<CX>
            mov       cx,4
            div       cx                  ; 5*w/4
            mov       z,ax
            ret
divword endp
code           ends
            end
```

Преобразование байта в слово и слова в двойное слово

Данное преобразование для знаковых и беззнаковых данных осуществляется по разному.

БЕЗЗНАКОВЫЕ числа занимают всю ячейку памяти, понятие знак для них НЕ существует - они считаются ПОЛОЖИТЕЛЬНЫМИ. Поэтому при преобразовании БЕЗЗНАКОВЫХ чисел в СТАРШУЮ часть результата надо занести НОЛЬ. Это можно сделать уже известными нам командами: MOV AH,0 или MOV DX,0. Однако это НЕ эффективно, используем родную для компьютера команду сложения по модулю 2:

XOR AH,AH или XOR DX,DX.

Для ЗНАКОВЫХ данных существуют две команды распространения знака.

CBW (Convert Byte toWord — преобразовать байт, находящийся в регистре AL, в слово — регистр AX) и

CWD (Convert Word to Double word — преобразовать слово, находящееся в регистре AX, в двойное слово — регистры <DX:AX>).

Операнды им НЕ нужны.

Синтаксис:

CBW

CWD

Логика работы команды CBW .

Получаем старшую часть (AH)	Известна младшая часть (AL)	
Биты 15-8	Знак - бит 7	Биты 6 - 0
1111 1111	1	Информационная часть числа
0000 0000	0	Информационная часть числа
Результат AX		

Логика работы команды CWD .

Получаем старшую часть (DX)	Известна младшая часть (AX)	
Биты 31-16	Знак - бит 15	Биты 14-0
1111 1111 1111 1111	1	Информационная часть числа
0000 0000 0000 0000	0	Информационная часть числа
Результат DX:AX		

```
            title divA
; CopyRight by Голубь Н.Г., 2001
            ;z=w/10
data        segment para public
             Extrn    wW:word,zW:word ;  Word
             Extrn    wI:word,zI:word ;  Integer
             Extrn    wB:BYTE,zB:BYTE ;  Byte
             Extrn    wS:BYTE,zS:BYTE ;  ShortInt
data        Ends
code        segment para public
   assume    cs:code,ds:data
   Public    divA
divA        proc      far
```

Деление больших чисел

Общий алгоритм деления числа любого размера на число любого размера нельзя построить с использованием команды DIV — такие операции выполняются при помощи сдвигов и вычитаний и оказываются весьма сложными. Рассмотрим сначала менее общую операцию (деление любого числа на слово или двойное слово), которую можно легко выполнить с помощью команд DIV

```
; деление 64-битного числа divident на 16-битное число divisor.
; Частное помещается в 64-битную переменную quotent,
; а остаток - в 16-битную переменную modulo
        mov         ax,word ptr divident[6]
        xor         dx,dx
        div         divisor
        mov         word ptr quotent[6],ax
        mov         ax,word ptr divident[4]
        div         divisor
        mov         word ptr quotent[4],ax
        mov         ax,word ptr divident[2]
        div         divisor
        mov         word ptr quotent[2],ax
        mov         ax,word ptr divident
        div         divisor
        mov         word ptr quotent,ax
        mov         modulo,dx
```

Деление любого другого числа полностью аналогично — достаточно только добавить нужное число троек команд mov/div/mov в начало алгоритма.

Наиболее очевидный алгоритм для деления чисел любого размера на числа любого размера — деление в столбик с помощью последовательных вычитаний делителя (сдвинутого влево на соответствующее количество разрядов) из делимого, увеличивая соответствующий разряд частного на 1 при каждом вычитании, пока не останется число, меньшее делителя (остаток):

```
; деление 64-битного числа в EDX:EAX на 64-битное число в ECX:EBX.
; Частное помещается в EDX:EAX, и остаток - в ESI:EDI
        mov     ebp,64          ; счетчик бит
        xor     esi,esi
        xor     edi,edi         ; остаток = 0
bitloop:
        shl     eax,1
        rcl     edx,1
        rcl     edi,1           ; сдвиг на 1 бит влево 128-битного числа
        rcl     esi,1           ; ESI:EDI:EDX:EAX
        cmp     esi,ecx         ; сравнить старшие двойные слова
        ja      divide
        jb      next
        cmp     edi,ebx         ; сравнить младшие двойные слова
        jb      next
divide:
        sub     edi,ebx
        sbb     esi,ecx         ; ESI:EDI = EBX:ECX
        inc     eax             ; установить младший бит в EAX
next:
        dec     ebp             ; повторить цикл 64 раза
        jne     bitloop
```

Команды передачи управления

Команды передачи управления позволяют нарушить естественную последовательность выполнения команд.

1. Команды безусловной передачи управления
2. Команды условной передачи управления

Команды передачи управления НЕ **меняют значения флагов.**

Команды безусловной передачи управления

1. **JMP**
2. **CALL**
3. **RET**

Команда безусловного перехода JMP

Команда JMP LABEL осуществляет переход на указанную метку. Если заранее известно, что переход вперед делается на место, лежащее в диапазоне 128 байт от текущего места, можно использовать команду JMP SHORT LABEL. Атрибут SHORT заставляет Ассемблер сформировать короткую форму команды перехода, даже если он еще не встретил метку LABEL.

NEAR переход в пределах сегмента

FAR межсегментный переход

Логика работы

NEAR

IP=IP+смещение

FAR

CS=CODE2

IP=IP+смещение

Команды условной передачи управления

- Команды условного перехода реализуют короткий переход, т.е. смещение в пределах [-128...127]. Если требуется переход дальше, нужно воспользоваться двумя командами Jcc и JMP.

- Базовых команд условного перехода всего 17, но они могут иметь различную мнемонику (это команды-синонимы — для удобства чтения и понимания программы), поэтому получается 31 команда. Читать эти команды достаточно просто, если знаешь как формируются их имена.

Первая буква команды J от уже известного нам слова (Jump — прыжок). Остальные буквы (cc) в сокращенном виде описывают условие перехода.

- E — Equal (равно).
- N — Not (не, отрицание).
- G — Greater (больше) — применяется для чисел со ЗНАКОМ.
- L — Less (меньше) — применяется для чисел со ЗНАКОМ.
- A — Above (выше, больше) — применяется для чисел БЕЗ ЗНАКА.
- B — Below (ниже, меньше) — применяется для чисел БЕЗ ЗНАКА.

Например, команда JL — переход, если меньше. Ей эквивалентна команда-синоним JNGE — переход, если НЕ больше и НЕ равно.

Условный переход обычно реализуется в два шага:

1. **Сравнение (CMP),** в результате чего формируются флаги
2. **Условная передача управления** (Jcc *Короткая_метка)* на помеченную команду в зависимости от значения флагов.

Таким образом, в ассемблере реализуется условный оператор **if.**

Знаковые и беззнаковые данные

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: имена, адреса и натуральные числа. В знаковых данных самый левый бит представляет собой знак, причем если его значение равно нулю, то число положительное, и если единице, то отрицательное. Многие числовые значения могут быть как положительными так и отрицательными.

В качестве примера предположим, что регистр AX содержит 11000110, а BX - 00010110. Команда

```
CMP  AX,BX
```

сравнивает содержимое регистров AX и BX. Если данные беззнаковые, то значение в AX больше, а если знаковые - то меньше.

Разница в командах перехода для знаковых и беззнаковых данных объясняется тем, что они реагируют на РАЗНЫЕ флаги (для знаковых данных существенен флаг SF, а для беззнаковых — CF)

Переходы 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.

Переходы 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 беззнаковых и знаковых 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/JP	Переход, если паритет нечетный	PF

Еще одна команда условного перехода проверяет равно ли содержимое регистра CX нулю. Одним из мест для команды JCXZ может быть начало цикла, где она проверяет содержит ли регистр CX ненулевое значение.

- Промоделируем на Ассемблере простейшую задачу для 16-разрядных знаковых и беззнаковых данных:
  - unsigned int c, d;
  - int a, b;
- if (a=b) then Fsign = 0;
- if (a<b) then Fsign = -1;
- if (a>b) then Fsign = 1;

- if (c=d) then Fusign = 0;
- if (c<d) then Fusign = -1;
- if (c>d) then Fusign = 1;

Команды циклов LOOPx

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

Синтаксис команды: LOOP *короткая метка*

Логика работы команды:

CX = Counter
 short_label:
 Выполнение тела цикла
CX = CX - 1
 if (CX != 0) goto short_label

Аналог реализации команды LOOP на Ассемблере:

```
        MOV CX, Counter
short_label:
      ; Выполнение тела цикла ;
      ; Проверка условия ПРОДОЛЖЕНИЯ цикла
       DEC CX
       CMP CX, 0
       JNE short_label
```

Команда **LOOP** уменьшает содержимое регистра CX на 1, затем передает управление метке shorMabel, если содержимое CX не равно 0. Передача управления на метку shortjabel для базовых процессоров — только КОРОТКАЯ [-128,0]. Поскольку условие выхода из цикла проверяется в КОНЦЕ, при значении Counter=0 цикл все равно выполнится. Этого мало, **мы еще и зациклимся**. Чтобы этого избежать, обычно ДО НАЧАЛА ЦИКЛА проверяют содержимое регистра CX на ноль. Таким образом, стандартная последовательность команд для организации цикла СО СЧЕТЧИКОМ имеет следующий вид:

```
MOV CX, Counter
JCXZ ExitCicle ; если CX = 0, цикл ОБОЙТИ
short_label:
; Выполнение тела цикла
LOOP short_label
ExitCicle:
```

ПРИМЕР

Вычислить значение факториала p = n! = 1*2*3*...*n

Известно, что 0! = 1. Отрицательным значение n быть НЕ может

```
.Model Large,C
 ; определение префикса для локальных меток
 locals @@
.code
Extrn C n: Word
Extrn C p: Word
Public proizv1
Proizv1 Proc far ; Вариант 1
mov cx,n ; количество повторений
mov si,1
mov ax,si
jcxz @@Exit ;if cx=0 then Exit
@@begin: ;    = НАЧАЛО цикла                =
mul si      ; <dx:ax> = <ax>*si
inc si
; ==== Выход из цикла ==================
loop @@beg±n
 @@Exit:
mov p,ax
ret
proizv1 endp
```

```
Public proizv2
Proizv2 Proc far ; Вариант 2
mov cx ,n    ; количество повторений
mov ax,1
jcxz @@Exit ;if cx=0 then Exit
@@begin: ;— = НАЧАЛО цикла =====
mul cx ; <dx:ax> = <ax>*cx
; ==== Выход из цикла ==========
loop @@begin
@@Exit:
mov p,ax ret
proizv2 endp end
```

Директива locals

Директива locals позволяет нам не думать о дублировании имен меток в разных подпрограммах. Метки с префиксом @@ считаются локальными. Если компилятор встретит метку с таким же именем, он просто при компиляции присвоит ей другое имя (обычно эти метки получают в конце имени номер, который увеличивается на единицу — все очень просто!).

Команда LOOPE (LOOPZ)

- **Команда LOOPE (LOOPZ) переход по счетчику и если равно**

  Данная команда имеет два равнозначных мнемонических имени (if Equal — если Равно или if Zero — если Ноль).

  Синтаксис команды:

  LOOPE короткая_метка

  LOOPZ короткая_метка

  Логика работы команды:

  CX = Counter

  Short_Label:

  Выполнение тела цикла

  CX = CX - 1

  if (CX != 0 && ZF = 1) goto Short_Label

  Все то, что говорилось для команды LOOP, справедливо и для команды LOOPE (LOOPZ), добавляется еще проверка флага ZF. Применяется данная команда в случае, если нужно досрочно выйти из цикла, как только находится ПЕРВЫЙ элемент, ОТЛИЧНЫЙ от заданной величины.

Команда LOOPNE (LOOPNZ)

- **Команда LOOPNE (LOOPNZ) переход по счетчику и если НЕ равно**

  Данная команда тоже имеет два равнозначных мнемонических имени (if Not Equal — если НЕ равно или if Not Zero — если НЕ ноль). В отличие от предыдущей команды проверяется, *сброшен* ли флаг нуля ZF=0.

Синтаксис команды: LOOPNE короткая метка
LOOPNZ короткая_метка

Логика работы команды:

CX = Counter

shortlabel: Выполнение тела цикла

CX = CX - 1

if (CX != 0 && ZF == 0) goto shortlabel

Все то, что говорилось для предыдущей команды, справедливо и для команды **LOOPNE (LOOPNZ).** Применяется данная команда в случае, если нужно досрочно выйти из цикла, как только находится ПЕРВЫЙ элемент, РАВНЫЙ заданной величине.

Вычислить значение суммы чисел натурального ряда: s = **1**+2+3+...+n. Вычисления закончить, как только значение суммы станет равным некоторому числу k или не будут перебраны все n чисел.

```
        Extrn       C n:Word
        Extrn       C s:Word
        Extrn       C k:Word
        Public      sum
        sum Proc far
        mov cx,n  ; количество повторений
        Xor  ax, ax
        xor  si, si
        jcxz  @@Exit ; if cx=0 then Exit
@@begin:            ;========Начало цикла ==============
        inc si
        add  ax, si
        cmp ax,k
        ;=Выход из цикла, если ax=k или cx=0
        loopNE @@begin
              @@Exit:
      mov s,ax
      ret
sum endp
```

Работа с подпрограммами

Команды

CALL proc – Вызов подпрограммы proc

RET - Возврат из подпрограммы

Команда вызова процедур CALL

Команда CALL (вызов) аналогична команде JMP. Поскольку команда CALL предназначена для вызова процедур, дополнительно она запоминает в стеке еще и адрес точки возврата.

Синтаксис: CALL *Имя_процедуры*

Логика работы команды в случае Near-вызова процедуры:

PUSH IP

IP= IP + смещение_к_нужной_процедуре

Логика работы команды в случае Far-вызова процедуры:

PUSH CS

PUSH IP

CS=Code2

IP= IP + смещение_к_нужной_процедуре

Если у вызываемой процедуры есть параметры, они передаются через стек ДО вызова процедуры.

Паскаль и C++ **по-разному работают с параметрами** (рассмотрим далее).

Для облегчения передачи параметров можно использовать команду **CALL** в **расширенном синтаксисе:**

CALL *Имя_процедуры [язык[, арг1]...],*

где *язык* - это C, CPP, PASCAL, BASIC, FORTRAN, PROLOG, NOLANGUAGE,

арг — аргумент, помещаемый в стек в соответствии с принятыми соглашениями.

Команда возврата в точку вызова RET

Команда RET (RETurn from procedure — возврат из процедуры) по действию является обратной команде CALL. Она обеспечивает возврат управления вызывающей программе и, если нужно, очистку стека на величину 16-разрядной константы *[im16]* байт.

Синтаксис: RET [Iml6]

Логика работы команды в случае Near-вызова процедуры:

POP IP

Логика работы команды в случае Far-вызова процедуры:

POP IP

POP CS

Соглашения о вызовах в стиле Pascal

Параметры в стек передаются по порядку слева направо, т.е. первым в стек помещается первый параметр, последним — последний. По окончании работы **вызываемая процедура** должна очистить стек.

Если процедура является функцией (в том понимании, как это принято в языке Паскаль), она должна возвратить значение. Возвращаемые значения должны находиться в соответствующих регистрах в зависимости от их типа:

Порядковый тип (integer, word, byte, shortint, char, longInt, boolean, перечисления):

Байт AL

Слово AX

Двойное слово <DX:AX>

Тип Real

<DX:BX:AX>

DX — старшая часть числа,

AX — младшая часть числа.

Single, double, extended

ST(0) — вершина стека математического сопроцессора

Pointer

<DX:AX> (DX — сегмент, AX — смещение).

String — указатель на временную область памяти.

Этот же порядок хранения возвращаемых значений действует и в языках *C/C++.*

Написать функцию с параметрами для вычисления арифметического выражения **i+j-k** для 16-разрядных целых знаковых или беззнаковых чисел (integer или word).

```
;Called as: TESTA(i, j, k);
;leftmost parameter — Крайний левый параметр
i equ 8     ; смещение относительно
j equ 6     ;       вершины стека
;rightmost parameter — Самый правый параметр
k equ 4
            ; Вариант 1 — параметры готовит программист
            .MODEL small,pascal
            .CODE
PUBLIC TESTA
TESTA PROC
            push bp
            mov bp, sp
      mov ax, [bp+i]    ;get i
      add ax,[bp+j]     ;add j to i
      sub ax,[bp+k]     ;subtract к from the sum
            pop bp
ret 6                   ;return, discarding 6 parameter bytes
TESTA ENDP
 END
```

Процедуру TESTA вызывается в Паскале: ta:= TESTA(i,j,k);

До вызова процедуры в стек должны быть переданы параметры - в данном случае в естественном для европейца порядке (слева направо: i, j, k):

```
PUSH I
PUSH j
PUSH k
```

О передаче параметров в данном случае заботится Паскаль.

Затем следует вызов функции: CALL TESTA. Это влечет за собой опять загрузку стека, на этот раз адресом возврата в точку вызова:

```
PUSH IP
```

В примере есть еще две команды :

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

Работать напрямую с вершиной стека НЕ рекомендуется, а работать надо. Поэтому и был предложен вариант использовать для этого регистр BP, который и является вершиной стека на момент начала выполнения процедуры.

Содержимое стека после вызова процедуры TESTA(i, j, k).

Положение вершины стека	Содержимое стека {длина ячейки 16 бит)	Значение смещении относительно регистра BP
SP	????	
	BP	
	IP	+2
	K	+4
	J	+6
	I	+8

В конце процедуры происходит восстановление значения регистра BP и возврат в точку вызова с очисткой стека. В данном случае было занято под параметры 6 байт.

Директива EQU (EQUivalent — Эквивалент) применяется для удобства сопоставления смещения в стеке с реальными параметрами. Вообще обратите внимание на эту директиву — она может существенно повысить читабельность программы и упростить программирование на Ассемблере.

Директива ARG

Существует более простой способ реализации подпрограмм с параметрами. В директиве **ARG** перечисляют по порядку все формальные параметры, далее указывается имя переменной (в нашем случае argLen), в которой подсчитывается количество байт, занимаемых параметрами, и, если нужно, указывается возвращаемое значение (параметр RETURNS).

Синтаксис директивы:

ARG ***список формалъных параметров [=их_длина] [RETURNS возв_значение]***

При этом с параметрами и возвращаемым значением можно работать так, как мы привыкли, не заботясь о принятых соглашениях. В конце процедуры (перед возвратом в точку вызова) нужно восстановить из стека содержимое регистра BP и саму команду возврата в точку вызова записать следующим образом: **ret** ***их_длина ;argLen***

```
;     i+j-k : integer or word
; Called as: TESTA(i, j, k) ;
; Вариант 2 - директива ARG
.MODEL small,pascal
.CODE
PUBLIC TESTA
TESTA PROC
;Function TESTA( i, j, k:integer):INTEGER;
ARG i:WORD,j:WORD,k:WORD=argLen RETURNS x:WORD
mov ax, i
add ax, j
sub ax, к
mov x,ax
pop bp
ret argLen
TESTA ENDP
END
```

Соглашения о вызовах в стиле C/C++

Параметры в стек передаются **справа налево**.

Первым в стек помещается ПОСЛЕДНИЙ параметр, последним — ПЕРВЫЙ.

По окончании работы **очистить стек должна** ВЫЗЫВАЮЩАЯ **процедура.**

```
;x=(2*a + b*c)/(d-a),

        ;int x,a,b,c,d;
        CODESEG
        Extrn       C X1:Dword
        Extrn       C X2:word ; ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
        Public      C prim
; int prim (int a, int b, int c, int d);
```

```
prim    proc        far
        push        bp
        mov         bp,sp ; указатель bp - на вершину стека
;!!!!!!!!!! параметры в стеке хранятся в ОБРАТНОМ
;       порядке
;       CS          EQU [bp+2] ; FAR!!!
;       IP          EQU [bp+4]
a       EQU         [bp+6]
b       EQU         [bp+8]
c       EQU         [bp+10]
d       EQU         [bp+12]
```

```
        mov     ax,2
        Imul    a               ; <dx>:<ax>=2*a
        mov     bx,dx           ; bx <=== ст.часть (dx)
        mov     cx,ax           ; cx <=== мл.часть (ax)
        mov     ax,b
        Imul    c               ; <dx>:<ax>=b*c
        add     ax,cx           ; <ax>=<ax>+<cx> (мл.часть)
        adc     dx,bx           ; <dx>=<dx>+<bx> (ст.часть)
        mov     word PTR X1, ax
        mov     word PTR X1+2,dx ; числитель
        mov     cx,d
        sub     cx,a            ; <cx>=<cx>-a
        mov     X2,cx           ; знаменатель
        Idiv    cx              ; <ax>=<dx>:<ax>/<cx>

        pop     bp
        ret
prim    endp
        end
```

- Модель large означает, что все процедуры имеют атрибут FAR.
- Функция int prim (int a, lilt b, int c, int d) вызывает из C++.
- ДО вызова функции в стек должны быть переданы параметры в обратном порядке (справа налево: d,c,b,a):
    - PUSH d
    - PUSH c
    - PUSH b
    - PUSH a
- О передаче параметров заботится программа на C++.
- Затем следует ДАЛЬНИЙ вызов функции:
  CALL prim.
- При этом в стек загружается адрес возврата в точку вызова:
    - PUSH IP
    - PUSH CS
- Затем начинает выполняться функцию, реализованную на Ассемблере. Вначале устанавливаем указатель на вершину стека. Для этого используется регистр BP:
    - push bp
    - mov bp,sp

Содержимое стека после вызова подпрограммы prim

Положение вершины стека	Содержимое стека (длина ячейки 16 бит)	Значение смещения относительно регистра BP
SP	????	
	BP	
	CS	+2
	IP	+4
	a	+6
	b	+8
	c	+10
	d	+12

В конце подпрограммы происходит восстановление значения регистра BP и возврат в точку вызова. Очистка стека здесь НЕ делается, этим должна заниматься программа на C++. Директива EQU применяется для удобства сопоставления смещения в стеке с реальными параметрами.

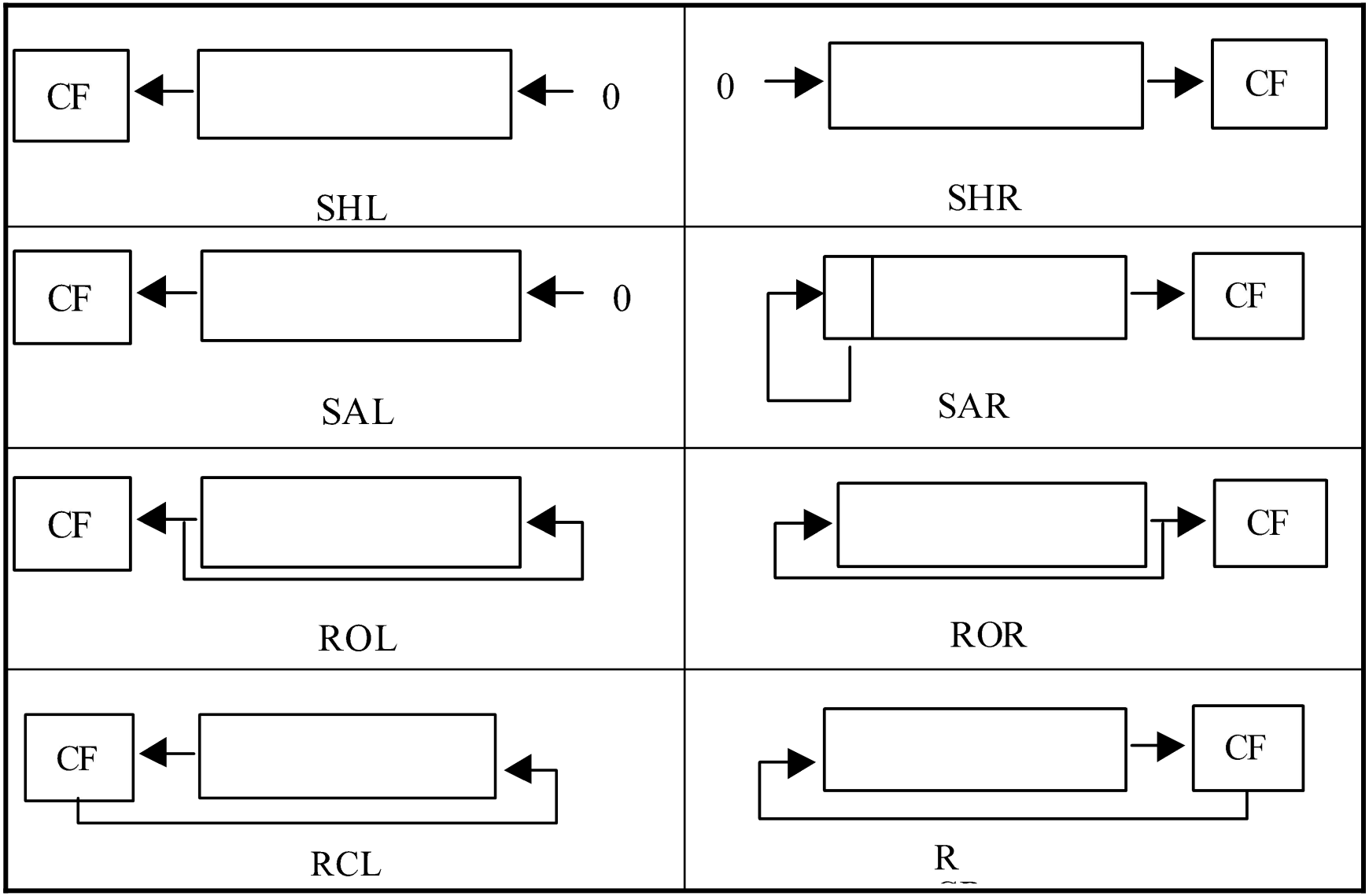
Команды сдвигов

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

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

Команды сдвигов

- команды логического сдвига вправо SHR и влево SHL;
- команды арифметического сдвига вправо SAR и влево SAL;
- команды циклического сдвига вправо ROR и влево ROL;
- команды циклического сдвига вправо RCR и влево RCL с переносом;


CF
0
SHL
0
CF
SHR
CF
0
SAL
CF
SAR
CF
ROL
CF
ROR
CF
RCL
CF
R

Пример использования команды SHR

```
MOV  CL,03              ;    AX:
MOV  AX,10110111B   ; 10110111
SHR  AX,1               ; 01011011   ;Сдвиг вправо на 1
SHR  AX,CL              ; 00001011   ;Сдвиг вправо на 3
```

- Первая команда SHR сдвигает содержимое регистра AX вправо на 1 бит.
- Выдвинутый в результате один бит попадает в флаг CF, а самый левый бит регистра AX заполняется нулем.
- Вторая команда сдвигает содержимое регистра
- AX еще на три бита. При этом флаг CF последовательно принимает значения 1,1, 0, а в три левых бита в регистре AX заносятся нули.

Команда арифметического сдвига вправо SAR

```
MOV  CL,03              ;    AX:
MOV  AX,10110111B   ; 10110111
SAR  AX,1               ; 11011011   ;Сдвиг вправо на 1
SAR  AX,CL              ; 11111011   ;Сдвиг вправо на 3
```

Команда SAR имеет важное отличие от команды SHR:

Для заполнения левого бита используется знаковый бит. Таким образом, положительные и отрицательные величины сохраняют свой знак.

В приведенном примере знаковый бит содержит единицу.

- При сдвигах влево правые биты заполняются нулями. Таким образом, результат команд сдвига SHL и SAL идентичен.
- Сдвиг влево часто используется для удваивания чисел, а сдвиг вправо - для деления на 2. Эти операции осуществляются значительно быстрее, чем команды умножения или деления. Деление пополам нечетных чисел (например, 5 или 7) образует меньшие значения (2 или 3, соответственно) и устанавливает флаг CF в 1. Кроме того, если необходимо выполнить сдвиг на 2 бита, то использование двух команд сдвига более эффективно, чем использование одной команды с загрузкой регистра CL значением 2.

Команды циклического сдвига

- Циклический сдвиг представляет собой операцию сдвига, при которой выдвинутый бит занимает освободившийся разряд.
- Команды циклического сдвига:

- ROR ;Циклический сдвиг вправо
- ROL ;Циклический сдвиг влево
- RCR ;Циклический сдвиг вправо с переносом
- RCL ;Циклический сдвиг влево с переносом

Команда циклического сдвига ROR:

- MOV CL,03 ; BX:
- MOV BX,10110111B ; 10110111
- ROR BX,1 ; 11011011 ;Сдвиг вправо на 1
- ROR BX,CL ; 01111011 ;Сдвиг вправо на 3

Первая команда ROR при выполнении циклического сдвига переносит правый единичный бит регистра BX в освободившуюся левую позицию. Вторая команда ROR переносит таким образом три правых бита.

КОМАНДЫ ЛОГИЧЕСКИХ ОПЕРАЦИЙ: AND, OR, XOR, TEST, NOT

Логические операции являются важным элементом в проектировании микросхем и имеют много общего в логике программирования. Команды AND, OR, XOR и TEST - являются командами логических операций. Эти команды используются для сброса и установки отдельных бит. Все эти команды обрабатывают один байт или одно слово в регистре или в памяти, и устанавливают флаги CF, OF, PF, SF, ZF.

- AND: Если оба из сравниваемых битов равны 1, то результат равен 1; во всех остальных случаях результат - 0.

- OR: Если хотя бы один из сравниваемых битов равен 1, то результат равен 1; если сравниваемые биты равны 0, то результат - 0.

- XOR: Если один из сравниваемых битов равен 0, а другой равен 1, то результат равен 1; если сравниваемые биты одинаковы (оба - 0 или оба - 1) то результат - 0.

- TEST: действует как AND-устанавливает флаги, но не изменяет биты.

Первый операнд в логических командах указывает на один байт или слово в регистре или в памяти и является единственным значением, которое может изменятся после выполнения команд. Пример:

	AND	OR	XOR
	0101	0101	0101
	0011	0011	0011
	----	----	----
Результат:	0001	0111	0110

Пример: вывести на экран шестнадцатеричное представление кода символа «Q».

```
;Сегмент стека
SSEG        SEGMENT  STACK
  DB  256 DUP (?)
SSEG        ENDS
;Сегмент данных
DSEG        SEGMENT
SMP DB ‘Q’              ;Символ
TBL   DB ‘0123456789ABCDEF’   ;Таблица 16-ричных цифр
DSEG        ENDS
;Сегмент кода
CSEG        SEGMENT
ASSUME      CS:CSEG, DS:DSEG, SS:SSEG
START:
  MOV     AX,DSEG         ;Инициализация DS
  MOV     DS,AX
```

```
        MOV     AH,2            ;В AH номер функции вывода
    MOV     BX,0
;Вывод на экран цифры соответствующей левой тетраде
    MOV     BL,Smp      ;В BL символ
    MOV     CL,4            ;В CL величина сдвига
    SHR     BL,CL           ;Сдвиг левой тетрады на место правой
    MOV     DL,Tbl[BX]  ;Загрузка цифры  из таблицы в DL
    INT 21H             ;Вывод на экран
;Вывод на экран цифры соответствующей правой  тетраде
    MOV     BL,Smp      ;В BL символ
    AND     BL,00001111B   ;Обнуление левой тетрады
    MOV     DL,Tbl[BX]  ;Загрузка цифры  из таблицы в DL
    INT 21H             ;Вывод на экран
;Вывод на экран символа «h»
    MOV     DL,'h'
    INT 21H
CSEG        ENDS
END     START
```

Операции ввода с клавиатуры и вывода на экран в DOS приложениях

1. Непосредственное чтение из клавиатурного буфера или запись в видеопамять;
2. Использование средств BIOS (INT 10H);
3. Использование функций MS DOS (INT 21H);

ВЫВОД НА ЭКРАН СРЕДСТВАМИ DOS

- Вывод на экран средствами DOS осуществляет 09 функция INT 21H DOS. Номер функции указывается в регистре AH. Адрес выводимой строки в DS:DX. В процессе выполнения операции конец сообщения определяется по ограничителю ($).

```
PRMP    DB   ‘Строка','$'
        .
        .
        MOV  AH,09        ;Запрос вывода на экран
        LEA  DX,PRMP      ;Загрузка адреса сообщ.
        INT  21H          ;Вызов DOS
```

- Знак ограничителя "$" можно кодировать непосредственно после символьной строки (как показано в примере), внутри строки: 'Имя покупателя?$', или в следующем операторе DB '$'. Используя данную операцию, нельзя вывести на экран символ доллара "$". Кроме того, если знак доллара будет отсутствовать в конце строки, то на экран будут выводиться все последующие символы, пока знак "$" не встретиться в памяти.
- Команда LEA загружает адрес области PRMP в регистр DX для передачи в DOS адреса выводимой информации. Адрес поля PRMP, загружаемый в DX по команде LEA, является относительным, поэтому для вычисления абсолютного адреса данных DOS складывает значения регистров DS и DX (DS:DX).

ВВОД ДАННЫХ С КЛАВИАТУРЫ

Для ввода с клавиатуры используется функция 0AH INT 21H. Она требует наличия списка параметров, содержащего поля, которые необходимы при выполнении команды INT.

- должна быть определена максимальная длина вводимого текста;
- должно быть определенное поле, куда команда возвращает действительную длину введенного текста в байтах.
- должно быть зарезервировано в памяти место для вводимой строки

Ниже приведен пример, в котором определен список параметров для области ввода. LABEL представляет собой директиву с атрибутом BYTE. Первый байт содержит максимальную длину вводимых данных. Второй байт необходим DOS для занесения в него действительного числа введенных символов. Третьим байтом начинается поле, которое будет содержать введенные символы.

```
NAMEPAR   LABEL   BYTE          ;Список параметров:
MAXLEN    DB      20            ; Максимальная длина
ACTLEN    DB      ?             ; Реальная длина
NAMEFLD   DB      20 DUP (' ')  ; Введенные символы
```

Так как в списке параметров директива LABEL не занимает места, то NAMEPAR и MAXLEN указывают на один и тот же адрес памяти. Для запроса на ввод необходимо поместить в регистр AH номер функции -10 (шест. 0AH), загрузить адрес списка параметров (NAMEPAR в нашем примере) в регистр DX и выполнить INT 21H:

```
MOV  AH,0AH          ;Запрос функции ввода
LEA  DX,NAMEPAR      ;адреса списка параметров
INT  21H             ;Вызвать DOS
```

Команда INT ожидает пока пользователь не введет с клавиатуры текст, проверяя при этом, чтобы число введенных символов не превышало максимального значения, указанного в списке параметров (20 в нашем случае). Для указания конца ввода пользователь нажимает клавишу Return.

Код этой клавиши (шест. 0D) также заносится в поле ввода (NAMEFLD в нашем примере). Если, например, пользователь ввел имя BROWN (Return), то список параметров будет содержать информацию:

- дес.: |20| 5| B| R| O| W| N| # | | | | | ...
- шест.: |14|05|42|52|4F|57 |4E|0D|20|20|20|20| ...

Во второй байт списка параметров (ACTLEN в нашем примере) команда заносит длину введенного имени - 05. Код Return находится по адресу NAMEFLD +5. Символ # использован здесь для индикации конца данных, так как шест. 0D не имеет отображаемого символа. Поскольку максимальная длина в 20 символов включает шест.0D, то действительная длина вводимого текста может быть только 19 символов.

ПРИМЕР: ВВОД И ВЫВОД ИМЕН

EXE-программа запрашивает ввод имени, затем отображает в середине экрана введенное имя и включает звуковой сигнал.

Программа продолжает запрашивать и отображать имена, пока пользователь не

нажмет Return в ответ на очередной запрос.

Рассмотрим ситуацию, когда пользователь ввел имя TED SMITH:

- 1. Разделим длину имени 09 на 2 получим 4
- 2. Вычтем это значение из 40, получим 36

Команда SHR в процедуре E10CENT сдвигает длину 09 на один бит вправо, выполняя таким образом деление на 2. Значение бит 00001001 переходит в 00000100. Команда NEG меняет знак +4 На -4. Команда ADD прибавляет значение 40, получая в регистре DL номер начального столбца - 36. При установке курсора на строку 12 и столбец 36 имя будет выведено на экран в следующем виде:

```
Строка 12:     TED SMITH
                |      |
Столбец:       36    40
```

В процедуре E10CODE имеется команда, которая устанавливает символ звукового сигнала (07) в области ввода непосредственно после имени:

```
MOV  NAMEFLD[BX],07
```

Предшествующая команда устанавливает в регистре BX значение длины, и команда MOV затем, комбинируя длину в регистре BX и адрес поля NAMEFLD пересылает код 07. Например, при длине имени 05 код 07 будет помещен по адресу NAMEFLD+05 (замещая значение кода Return). Последняя команда в процедуре E10CODE устанавливает ограничитель "$" после кода 07. Таким образом, когда процедура F10CENT выводит на экран имя, то генерируется также звуковой сигнал.

```
;-----------------------------------------------
STSCKSG SEGMENT PARA STACK 'Stack'
        DW      32 DUP(?)
STACKSG ENDS
;-----------------------------------------------
DATASG  SEGMENT PARA 'Data'
NAMEPAR LABEL   BYTE            ;Имя списка параметров:
MAXNLEN DB      20              ; макс. длина имени
NAMELEN DB      ?               ; число введенных
                                ;символов
NAMEFLD DB      20 DUP(' '),'$' ;имя и ограничитель
                                ;для вывода на экран
PROMPT  DB      'Name? ', '$'
DATASG  ENDS
```

```
CODESG  SEGMENT PARA 'Code"
BEGIN   PROC    FAR
        ASSUME  CS:CODESG,DS:DATASG,SS:STACKSG,ES:DATASG
        PUSH    DS
        SUB     AX,AX
        PASH    AX
        MOV     AX,DATASC
        MOV     DS,AX
        MOV     ES,AX
        CALL    Q10CLR          ;Очистить экран
A20LOOP:
        MOV     DX,0000         ;Установить курсор в 00,00
        CALL    Q20CURS
        CALL    B10PRMP         ;Выдать текст запроса
        CALL    D10INPT         ;Ввести имя
        CALL    Q10CLR          ;Очистить экран
        CMP     NAMELEN,00      ;Имя введено?
        JE      A30             ; нет - выйти
        CALL    E10CODE         ;Установить звуковой сигнал
                                ; и ограничитель '$'
        CALL    F10CENT         ;Центрирование и вывод
        JMP     A20LOOP
        A30:
        RET             ;Вернуться в DOS
BEGIN   ENDP
```

Вывод текста запроса:

- B10PRMP PROC NEAR
- MOV AH,09 ;Функция вывода на экран
- LEA DX,PROMPT
- INT 21H
- RET
- B10PRMP ENDP

Ввод имени с клавиатуры:

```
D10INPT PROC    NEAR
        MOV     AH,0AH      ;Функция ввода
        LEA     DX,NAMEPAR
        INT     21H
        RET
D10INPT ENDP
```

Установка звукового сигнала и ограничителя '$':

- E10CODE PROC NEAR
- MOV BH,00 ;Замена символа Return (0D)
- MOV BL,NAMELEN ; на зв. сигнал (07)
- MOV NAMEFLD[BX],07
- MOV NAMEFLD[BX+1],'$' ;Установить ограничитель
- RET
- E10CODE ENDP

Центрирование и вывод имени на экран:

```
F10CENT PROC   NEAR
        MOV    DL,NAMELEN  ;Определение столбца:
        SHR    DL,1        ; разделить длину на 2,
        NEG    DL          ; поменять знак,
        ADD    DL,40       ; прибавить 40
        MOV    DH,12       ;Центральная строка
        CALL   Q20CURS     ;Установить курсор
        MOV    AH,09
        LEA    DX,NAMEFLD  ;Вывести имя на экран
        INT    21H
        RET
F10CENT ENDP
```

Очистить экран:

```
Q10CLR  PROC   NEAR
MOV    AX,0600H          ;Функция прокрутки экрана
MOV    BH,30             ;Цвет (07 для ч/б)
MOV    CX,0000           ;От 00,00
MOV    DX,184FH          ;До 24,79
INT    10H               ;Вызов BIOS
RET
Q10CLR
```

Установка курсора (строка/столбец):

```
Q20CURS PROC   NEAR          ;DX уже установлен
        MOV    AH,02         ;Функция установки курсора
        MOV    BH,00         ;Страница #0
        INT    10H           ;Вызов BIOS
        RET
Q20CURS ENDP

CODESG  ENDS
END    BEGIN
```

Преобразование строки в число и числа в строку

Данные, вводимые с клавиатуры, имеют ASCII-формат, например, буквы SAM имеют в памяти шестнадцатеричное представление 53414D, цифры 1234 - шест.31323334. Во многих случаях формат алфавитных данных, например, имя человека или описание статьи, не меняется в программе. Но для выполнения арифметических операций над числовыми значениями, такими как шест.31323334, требуется специальная обработка.

ПРЕОБРАЗОВАНИЕ ASCII-ФОРМАТА (строки) В ДВОИЧНЫЙ ФОРМАТ (число)

Для выполнения арифметических операций часто требуется преобразование их в двоичный формат.

Процедура преобразования заключается в следующем:

- 1. Начинают с самого правого байта числа в ASCII-формате и обрабатывают справа налево.
- 2. Удаляют тройки из левых шестнадцатеричных цифр каждого ASCII-байта.
- 3. Умножают ASCII-цифры на 1, 10, 100 (шест.1, A, 64) и т.д.
- складывают результаты.

Для примера рассмотрим преобразование числа 1234 из ASCII-формата в двоичный формат:

	Десятичное	Шестнадцатеричное
• 4 x 1 =	4	4
• 3 x 10 =	30	1E
• 2 x 100 =	200	C8
• 1 x 1000 =	1000	3E8
• Результат:		04D2

Действительно шест.04D2 соответствует десятичному 1234.

В процедуре B10ASBI выполняется преобразование ASCII-числа 1234 в двоичный формат. В примере предполагается, что длина ASCII-числа равна 4 и она записана в поле ASCLEN. Для инициализации адрес ASCII-поля ASCVAL-1 заносится в регистр SI, а длина - в регистр BX. Команда по метке B20 пересылает ASCII-байт в регистр AL:

```
MOV  AL,[SI+BX]
```

Здесь используется адрес ASCVAL-1 плюс содержимое регистра BX (4), т.е. получается адрес ASCVAL+3 (самый правый байт поля ASCVAL). В каждом цикле содержимое регистра BX уменьшается на 1, что приводит к обращению к следующему слева байту. Для данной адресации можно использовать регистр BX, но не CX, и, следовательно, нельзя применять команду LOOP. В каждом цикле происходит также умножение поля MULT10 на 10, что дает в результате множители 1,10,100 и т.д. Такой прием применен для большей ясности, однако, для большей производительности множитель можно хранить в регистре SI или DI.

```
CODESG  SEGMENT
        ASSUME  CS:CODESG,DS:CODESG,SS:CODESG
        ORG     100H
BEGIN:  JMP     SHORT MAIN
; ----------------------------------------
ASCVAL  DB      '1234'          ;Элементы данных
BINVAL  DB      0
ASCLEN  DB      4
MULT10  DB      1
; ----------------------------------------
MAIN    PROC    NEAR            ;Основная процедура:
        CALL    B10ASBI         ;Вызвать преобразование ASCII
        CALL    C10BIAS         ;Вызвать преобразование двоичное
        RET
MAIN    ENDP
;           ------------------------------
;           Преобразование ASCII в двоичное:
;           ------------------------------
B10ASBI PROC
        MOV     CX,10           ;Фактор умножения
        LEA     SI,ASCVAL-1     ;Адрес ASCVAL
        MOV     BX,ASCLEN       ;Длина ASCVAL
B20:
        MOV     AL,[SI+BX]      ;Выбрать ASCII-символ
        AND     AX,000FH        ;Очистить зону тройки
        MUL     MULT10          ;Умножить на фактор 10
        ADD     BINVAL,AX       ;Прибавить к двоичному
        MOV     AX,MULT10       ;Вычислить следующий
        MUL     CX              ;фактор умножения
        MOV     MULT10,AX
        DEC     BX              ;Последн. ASCII-символ?
        JNZ     B20             ; Нет - продолжить
        RET
B10ASBI ENDP
```

ПРЕОБРАЗОВАНИЕ ДВОИЧНОГО ФОРМАТА В ASCII-ФОРМАТ

Для того, чтобы напечатать или отобразить на экране результат выполнения арифметических операций, необходимо преобразовать его в ASCII-формат. Данная операция включает в себя процесс обратный предыдущему. Вместо умножения используется деление двоичного числа на 10 (шест.0A) пока результат не будет меньше 10. Остатки, которые лежат в границах от 0 до 9, образуют число в ASCII формате. В качестве примера рассмотрим преобразование шест.4D2 обратно в десятичный формат:

	Частное	Остаток
4D2 : A	7B	4
7B : A	C	3
C : A	1	2

Так как последнее частное 1 меньше, чем шест.A, то операция завершена. Остатки вместе с последним частным образуют результат в ASCII-формате, записываемый справа налево 1234. Все остатки и последнее частное должны записываться в память с тройками, т.е. 31323334.

Процедура C10BIAS преобразует шест.4D2 (результат вычисления в процедуре B10ASBI) в ASCII-число 1234.

Преобр. дв. в ASCII:

```
C10BIAS PROC
        MOV     CX,0010          ;Фактор деления
        LEA     SI,ASCVAL+3      ;Адрес ASCVAL
        MOV     AX,BINVAL        ;Загрузить дв. число
C20:
        CMP     AX,0010          ;Значение меньше  10?
        JB      C30               ; Да - выйти
        XOR     DX,DX            ;Очистить часть частного
        DIV     CX                ;Разделить на 10
        OR      DL,30H
        MOV     [SI],DL          ;Записать ASCII-символ
        DEC     SI
        JMP     C20
C30:
        OR      AL,30H            ;Записать посл. частное
        MOV     [SI],AL           ; как ASCII-символ
        RET
C10BIAS ENDP

CODESG  ENDS
END    BEGIN
```

Программирование в LINUX (первый пример)

```
#include <stdlib.h>
#include <stdio.h>

extern void asm_calc(int,int,int*,int*);
int main(void){
    int i,j,iplusj=0,imulj=0;
    scanf("%d %d",&i,&j);
    printf("C  :\t  i+j=%d\n",i+j);
    printf("C  :\t  i*j=%d\n",i*j);
    asm_calc(i,j,&iplusj,&imulj);
    printf("ASM:\t i+j=%d\n",iplusj);
    printf("ASM:\t i*j=%d\n",imulj);
    return 0;
};
```

```
section .text
global asm_calc
asm_calc:
        ;параметры: int i,int j, int* iplusj, int* imulj
        ;то есть - число 1,число 2, указатель на результат 1, указатель на результат 2
          ;по правилам вызова C - операнды в стеке.
          ;[esp] - адрес возврата
          ;[esp+4] - первый аргумент
          ;[esp+8] - второй и т.д.
    push eax
    push ecx
    push ebx
    push edx

    mov eax, [esp+4+4*4] ;i    4*4 - сохранённые в стек регистры
    mov ebx, [esp+8+4*4] ;j
    mov ecx, [esp+16+4*4] ;imulj
    mul ebx         ;Результат в edx:eax
    mov [ecx],eax   ;результат перемножения заброшен по адресу в ecx

    mov ecx, [esp+12+4*4] ;iplusj
    mov eax, [esp+4+4*4]  ;i
    mov ebx, [esp+8+4*4]  ;j
    add eax,ebx
    mov [ecx],eax   ;результат сложения заброшен туда же - по адресу в ecx

    pop edx
    pop ebx
    pop ecx
    pop eax
    ret
```

makefile

- LDFLAGS=-g
- CFLAGS=-g
- all: main

- main: main.o asm_file.o

- main.o: main.c

- asm_file.o: asm_file.asm
- nasm -g -o $@ -f elf $^

Утилита make

Студенты привыкают к использованию графических интерфейсов для разработки программного обеспечения и часто не понимают, как это делается без кнопочки "билд". В целях облегчения жизни нужно снижать трудозатраты на рутинную работу, перекладывать её на плечи умных утилит, но как бы мы не обрастали графическими оболочками и средствами разработки проектов не стоит забывать на чем основывается процесс сборки проектов.

Допустим, вы разрабатываете некую программу под названием foo, состоящую из пяти заголовочных файлов

1.h, 2.h, 3.h, 4.h и -- 5.h,

и шести файлов с исходным текстом программы на языке C

1.cpp, 2.cpp, 3.cpp, 4.cpp, 5.cpp и main.cpp.

Теперь представим себе, что вы обнаружили ошибку в файле 2.cpp и исправили ее. Чтобы получить исправленную версию программы вы компилируете все файлы, входящие в состав проекта, хотя изменения коснулись только одного файла. Это приводит к нерациональной потере времени, особенно если компьютер не слишком быстрый.

Утилита make

Задача утилиты make - автоматически определять, какие файлы проекта были изменены и требуют компиляции, и применять необходимые для этого команды. Хотя примеры применения относятся к использованию утилиты для описания процесса компиляции пограмм на языке C/C++, утилита может использоваться для описания сценариев обновления любых файлов.

Файл Makefile

Для использования утилиты необходимо описать сценарии в файле **Makefile**. Сценарии указывают взаимосвязь между файлами проекта и определяют необходимые действия для обновления каждого файла проекта. Например, для создания исполняемого кода программы необходимо использовать объектные файлы, которые получены путем компиляции исходных кодов. Для этого в сценарии должны содержаться правила создания и обновления объектных файлов из исходных и правило получения исполняемого файла из объектных файлов.

Правила компиляции составных частей проекта заносятся в **Makefile**. Затем все необходимые действия по компиляции и сборке проекта могут быть выполнены автоматически при запуске утилиты **make из рабочей директории проекта**. Необходимость выполнения команд для обновления объектных и исполняемых кодов программ определяется исходя из даты и времени обновления исходных файлов проекта.

Стандартизация процедуры сборки программ с использованием утилиты make позволяет собирать пакеты программ из исходных кодов не имея представления о структуре и составных частях исходных кодов. Для сборки проекта распространяемого с исходными кодами достаточно выполнить команду "make" в корневой директории проекта. Именно так собирается открытое программное обеспечение.

Структура Makefile

Makefile состоит из так называемых "правил", имеющих вид:

- **имя-результата: исходные-имена ...**
- **команды**
- ...
- ...

имя-результата - это обычно имя файла, генерируемого программой, например, исполняемый или объектный файл. "Результатом" может быть действие никак не связанное с процессом компиляции, например, clean - очистка.

исходное-имя - это имя файла, используемого на вводе, необходимое, чтобы создать файл с именем-результата.

команда - это действие, выполняемое утилитой make. Правило может включать более одной команды, В начале каждой команды надо вставлять **отступ (символ "Tab").** Команда выполняется, если один из файлов в списке исходные-имена изменился. Допускается написание правила содержащего команду без указания зависимостей. Например, можно создать правило "clean", удаляющее объектные файлы проекта, без указания имен.

Итак, правила объясняют как и в каком случае надо пересобирать определённые файлы проекта.

Стандартные правила

К числу стандартных правил относятся:

- all - основная задача, компиляция программы.
- install - копирует исполняемые коды программ, библиотеки настройки и всё что необходимо для последующего использования
- uninstall - удаляет компоненты программы из системы
- clean - удаляет из директории проекта все временные и вспомогательные файлы.

Пример Makefile

Ниже приводится простой пример (номера строк добавлены для ясности).

```
1 client: conn.o
2    g++ client.cpp conn.o -o client
3 conn.o: conn.cpp conn.h
4    g++ -c conn.cpp -o conn.o
```

В этом примере строка, содержащая текст client: conn.o, называется "строкой зависимостей", а строка g++ client.cpp conn.o -o client называется "правилом" и описывает действие, которое необходимо выполнить.

Более подробно о примере

1. Задается цель -- исполняемый файл client, который зависит от объектного файла conn.o;
2. Правило для сборки данной цели;
3. Задается цель conn.o и файлы, от которых она зависит -- conn.cpp и conn.h;
4. Описывается действие по сборке цели conn.o.

Комментарии

Строки, начинающиеся с символа "#", являются комментариями. Пример makefile с комментариями:

```
# Создать исполняемый файл "client"
client: conn.o
    g++ client.cpp conn.o -o client

# Создать объектный файл "conn.o"
conn.o: conn.cpp conn.h
    g++ -c conn.cpp -o conn.o
```

"Ложная" цель

Обычно "ложные" [phony] цели, представляющие "мнимое" имя целевого файла, используются в случае возникновения конфликтов между именами целей и именами файлов при явном задании имени цели в командной строке. Допустим в makefile имеется правило, которое не создает ничего, например:

- clean:
- rm *.o temp

Поскольку команда rm не создает файл с именем clean, то такого файла никогда не будет создано и поэтому команда make clean всегда будет срабатывать.

Декларация .PHONY

Однако, данное правило не будет работать, если в текущем каталоге будет существовать файл с именем clean. Поскольку цель clean не имеет зависимостей, то она никогда не будет считаться устаревшей и, соответственно, команда 'rm *.o temp' никогда не будет выполнена. (при запуске make проверяет даты модификации целевого файла и тех файлов, от которых он зависит. И если цель оказывается "старше", то make выполняет соответствующие команды-правила) Для устранения подобных проблем предназначена специальная декларация .PHONY, объявляющая "ложную" цель. Например:

```
.PHONY : clean
```

Таким образом мы указываем необходимость исполнения цели, при явном ее указании, в виде make clean вне зависимости от того существует файл с таким именем или нет.

Переменные

Определить переменную в makefile вы можете следующим образом:

```
$VAR_NAME=value
```

В соответствии с соглашениями имена переменных задаются в верхнем регистре:

```
$OBJECTS=main.o test.o
```

Чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '$', например:

```
$(VAR_NAME)
```

В makefile-ах существует два типа переменных: "упрощенно вычисляемые" и "рекурсивно вычисляемые". В рекурсивно вычисляемых переменных все ссылки на другие переменные будут замещены их значениями, например:

```
TOPDIR=/home/tedi/project
SRCDIR=$(TOPDIR)/src
```

При обращении к переменной SRCDIR вы получите значение /home/tedi/project/src.

Однако рекурсивные переменные могут быть вычислены не всегда, например следующие определения:

```
CC = gcc -o
CC = $(CC) -O2
```

выльются в бесконечный цикл. Для разрешения этой проблемы следует использовать "упрощенно вычисляемые" переменные:

```
CC := gcc -o
CC += $(CC) -O2
```

Где символ ':=' создает переменную CC и присваивает ей значение "gcc -o". А символ '+=' добавляет "-O2" к значению переменной CC.

Вывод на экран через INT 80H

```
section .text
    global _start

_start:

 mov   eax,15
 mov   ebx,20
 mul    ebx
 cmp   eax,300
 jnzerror
```

```
all_ok:
mov     edx,len1 ;third argument:
    message length
mov     ecx,msg1 ;second argument:
                 pointer to message to write
mov     ebx,1    ;first argument:
              file handle   (stdout)
mov     eax,4    ;system call number
        (sys_write)
int     0x80     ;call kernel
jmp exit1
```

- error: mov edx,len2 ;third argument: message length
- mov ecx,msg2 ;second argument: pointer to message to write
- mov ebx,1 ;first argument: file handle (stdout)
- mov eax,4 ;system call number (sys_write)
- int 0x80 ;call kernel
- jmp exit1

- exit1:
- mov ebx,0 ;first syscall argument: exit code
- mov eax,1 ;system call number (sys_exit)
- int 0x80 ;call kernel

- section .data ;section declaration

- msg1 db "All OK!",0xa
- len1 equ $ - msg1 ;length of our string
- msg2 db "Error!",0xa
- len2 equ $ - msg2

makefile

- all:main
-
- main: main.o
- ld -s -o main main.o

- main.o:main.asm
- nasm -f elf $^

- clean:
- rm main *.o

Вывод с помощью printf

```
extern printf
section .text                    ;section declaration

    global main
main:

;write our string to stdout
        mov eax,15
        push    eax
        push    dword msg
        call    printf
        add     esp,8
;and exit

        mov ebx,0       ;first syscall argument: exit code
        mov     eax,1       ;system call number (sys_exit)
        int     0x80        ;call kernel

section .data           ;section declaration

msg     db      "And the number in eax=%d",0xa,0x0
```

makefile

- LDFLAGS=-g
- all:main
-
- main: main.o

- main.o:main.asm
- 	nasm -g -f elf $^

- clean:
- 	rm main *.o

Вызов функций scanf и printf из Nasm

- **Функции scanf и printf определены в библиотеке glibc. Эти функции можно указать в ассемблерной программе как внешние с помощью директивы extern. Объектный файл получается стандартным образом. А вот при компоновке (линковке) необходимо указать библиотеку libc.so либо использовать для компоновки gcc, который, в отличие от ld по умолчанию компонует все объектные файлы с библиотекой libc.so**
- **global _start**

- **;Объявляем используемые внешние функции из libc**
- **extern exit**
- **extern puts**
- **extern scanf**
- **extern printf**

- **;Сегмент кода:**
- **section .text**
-
- **;Функция main:**
- **_start:**
-
- **;Параметры передаются в стеке:**
- **push dword msg**
- **call puts**
-
- **;По конвенции Си вызывающая процедура должна**
- **;очищать стек от параметров самостоятельно:**
- **sub esp, 4**
-
- **push dword a**
- **push dword b**
- **push dword msg1**
- **call scanf**
- **sub esp, 12**
-

```
mov eax, dword [a]
add eax, dword [b]

push eax
push dword msg2

call printf
add esp, 8

;Завершение программы с кодом выхода 0:
push dword 0
call exit

ret

;Сегмент инициализированных данных
section .data
msg  : db "An example of interfacing with GLIBC.",0xA,0
msg1 : db "%d%d",0
msg2 : db "%d", 0xA, 0

; Сегмент неинициализированных данных
section .bss
a resd 1
b resd 1
```

Арифметические операции в формате ASCII

Данные, вводимые с клавиатуры, имеют ASCII-формат, например, цифры 1234 - шест.31323334. Для выполнения арифметических операций над числовыми значениями, такими как шест.31323334, требуется специальная обработка.

С помощью следующих ассемблерных команд можно выполнять арифметические операции непосредственно над числами в ASCII-формате:

AAA (ASCII Adjust for Addition - коррекция для сложения ASCII-кода)
AAD (ASCII Adjust for Division - коррекция для деления ASCII-кода)
AAM (ASCII Adjust for Multiplication - коррекция для умножения ASCII-кода)
AAS (ASCII Adjust for Subtraction - коррекция для вычитания ASCII-кода)

Эти команды кодируются без операндов и выполняют автоматическую коррекцию в регистре AX. Коррекция необходима, так как ASCII-код представляет так называемый распакованный десятичный формат, в то время, как компьютер выполняет арифметические операции в двоичном формате.

Сложение в ASCII-формате

- Рассмотрим процесс сложения чисел 8 и 4 в ASCII-формате:

- Шест. 38
- 34
- --
- Шест. 6C

- Полученная сумма неправильна ни для ASCII-формата, ни для двоичного
- формата. Однако, игнорируя левую 6 и прибавив 6 к правой шест. C:
- шест.C + 6 = шест.12 - получим правильный результат в десятичном формате.
- Правильный пример слегка упрощен, но он хорошо демонстрирует процесс,
- который выполняет команда AAA при коррекции.

В качестве примера, предположим, что регистр AX содержит шест.0038, а регистр BX - шест.0034. Числа 38 и 34 представляют два байта в ASCII-формате, которые необходимо сложить. Сложение и коррекция кодируется следующими командами:

```
ADD  AL,BL     ;Сложить 34 и 38
AAA            ;Коррекция для сложения ASCII-кодов
```

Команда AAA проверяет правую шест. цифру (4 бита) в регистре AL. Если эта цифра находится между A и F или флаг AF равен 1, то к регистру AL прибавляется 6, а к регистру AH прибавляется 1, флаги AF и CF устанавливаются в 1. Во всех случаях команда AAA устанавливает в 0 левую шест. цифру в регистре AL. Результат - в регистре AX:

После команды ADD: 006C
После команды AAA: 0102

Для того, чтобы выработать окончательное ASCII-представление, достаточно просто поставить тройки на место левых шест. цифр:

```
OR   AX,3030H  ;Результат 3132
```

Все показанное выше представляет сложение однобайтовых чисел. Сложение многобайтных ASCII-чисел требует организации цикла, который выполняет обработку справа налево с учетом переноса. В примере складываются два трехбайтовых ASCII-числа в четырехбайтовую сумму. Обратите внимание на следующее:

Сложение в ASCII-формате

```
CODESG  SEGMENT
        ASSUME  CS:CODESG,DS:CODESG,SS:CODESG
        ORG     100H
BEGIN:  JMP     SHORT MAIN
; ----------------------------------------------
ASC1    DB      '578'       ;Элементы данных
ASC2    DB      '694'
ASC3    DB      '0000'
; ----------------------------------------------
MAIN    PROC    NEAR
        CLC
        LEA     SI,AASC1+2   ;Адреса ASCII-чисел
        LEA     DI,AASC2+2
        LEA     BX,AASC1+3
        MOV     CX,03        ;Выполнить 3 цикла
A20:
        MOV     AH,00        ;Очистить регистр AH
        MOV     AL,[SI]      ;Загрузить ASCII-байт
        ADC     AL,[DI]      ;Сложение (с переносом)
        AAA                  ;Коррекция для ASCII
        MOV     [BX],AL      ;Сохранение суммы
        DEC     SI
        DEC     DI
        DEC     BX
        LOOP    A20          ;Циклиться 3 раза
        MOV     [BX],AH      ;Сохранить перенос
        RET
MAIN    ENDP
CODESG  ENDS
        END     BEGIN
```

В программе используется команда ADC, так как любое сложение может вызвать перенос, который должен быть прибавлен к следующему (слева) байту. Команда CLC устанавливает флаг CF в нулевое состояние.

Команда MOV очищает регистр AH в каждом цикле, так как команда AAA может прибавить к нему единицу. Команда ADC учитывает переносы. Заметьте, что использование команд XOR или SUB для очистки регистра AH изменяет флаг CF.

Когда завершается каждый цикл, происходит пересылка содержимого регистра AH (00 или 01) в левый байт суммы.

В результате получается сумма в виде 01020702. Программа не использует команду OR после команды AAA для занесения левой тройки, так как при этом устанавливается флаг CF, что изменит результат

команды ADC. Одним из решений в данном случае является сохранение флагового регистра с помощью команды PUSHF, выполнение команды OR, и, затем, восстановление флагового регистра командой POPF:

```
ADC  AL,[DI]    ;Сложение с переносом
AAA             ;Коррекция для ASCII
PUSHF           ;Сохранение флагов
OR   AL,30H     ;Запись левой тройки
POPF            ;Восстановление флагов
MOV  [BX],AL    ;Сохранение суммы
```

Вместо команд PUSHF и POPF можно использовать команды LAHF (Load AH with Flags загрузка флагов в регистр AH) и SAHF (Store AH in Flagregister - запись флагов из регистра AH во флаговый регистр). Команда LAHF загружает в регистр AH флаги SF, ZF, AF, PF и CF; а команда SAHF записывает содержимое регистра AH в указанные флаги. В приведенном примере, однако, регистр AH уже используется для арифметических переполнений. Другой способ вставки троек для получения ASCII-кодов цифр - организовать обработку суммы командой OR в цикле.

Вычитание в ASCII-формате

Команда AAS (ASCII Adjust for Subtraction - коррекция для вычитания ASCII-кодов) выполняется аналогично команде AAA. Команда AAS проверяет правую шест. цифру (четыре бита) в регистре AL. Если эта цифра лежит между A и F или флаг AF равен 1, то из регистра AL вычитается 6, а из регистра

AH вычитается 1, флаги AF и CF устанавливаются в 1. Во всех случаях команда AAS устанавливает в 0 левую шест.цифру в регистре AL.

В следующих двух примерах предполагается, что поле ASC1 содержит шест.38, а поле ASC2 - шест.34:

Пример 1:

```
                    AX      AF

MOV  AL,ASC1        ;0038
SUB  AL,ASC2        ;0034    0
AAS                 ;0004    0
```

Пример 2:

```
                    AX      AF

MOV  AL,ASC2        ;0034
SUB  AL,ASC1        ;00FC    1
AAS                 ;FF06    1
```

В примере 1 команде AAS не требуется выполнять коррекцию. В примере 2, так как правая цифра в регистре AL равна шест.C, команда AAS вычитает 6 из регистра AL и 1 из регистра AH и устанавливает в 1 флаги AF и CF. Результат (который должен быть равен -4) имеет шест. представление FF06, т.е. десятичное дополнение числа -4.

Умножение в ASCII-формате

Команда AAM (ASCII Adjust for Multiplication - коррекция для умножения ASCII-кодов) выполняет корректировку результата умножения ASCII-кодов в регистре AX. Однако, шест. Цифры должны быть очищены от троек и полученные данные уже не будут являться действительными ASCII-кодами. (В руководствах фирмы IBM для таких данных используется термин распакованный десятичный формат). Например, число в ASCII-формате 31323334 имеет распакованное десятичное представление 01020304. Кроме этого, надо помнить, что коррекция осуществляется только для одного байта за одно выполнение, поэтому можно умножать только однобайтные поля. Для более длинных полей необходима организация цикла.

Команда AAM делит содержимое регистра AL на 10 (шест.0A) и записывает частное в регистр AH, а остаток в AL. Предположим, что в регистре AL содержится шест.35, а в регистре CL - шест.39. Следующие команды умножают содержимое регистра AL на содержимое CL и преобразуют результат в ASCII-формат:

```
                                                    AX:
AND  CL,0FH    ;Преобразовать CL в 09
AND  AL,0FH     ;Преобразовать AL в 05          0005
MUL  CL         ;Умножить AL на CL              002D
AAM              ;Преобразовать в распак.дес.   0405
OR   AX,3030H  ;Преобразовать в ASCII-ф-т      3435
```

Команда MUL генерирует 45 (шест.002D) в регистре AX, после чего команда AAM делит это значение на 10, записывая частное 04 в регистр AH и остаток 05 в регистр AL. Команда OR преобразует затем распакованное десятичное число в ASCII-формат.

Следующий пример демонстрирует умножение четырехбайтового множимого на однобайтовый множитель. Так как команда AAM может иметь дело только с однобайтовыми числами, то в программе организован цикл, который обрабатывает байты справа налево. Окончательный результат умножения в данном примере - 0108090105.

Если множитель больше одного байта, то необходимо обеспечить еще один цикл, который обрабатывает множитель. В этом случае проще будет преобразовать число из ASCII-формата в двоичный формат.

```
CODESG  SEGMENT
        ASSUME  CS:CODESG,DS:CODESG,SS:CODESG
        ORG     100H
BEGIN:  JMP     MAIN
; -----------------------------------------
MULTCND DB      '3783'          ;Элементы данных
MULTPLR DB      '5'
PRODUCT DB      5 DUP(0)
; -----------------------------------------
MAIN    PROC    NEAR
        MOV     CX,04           ;4 цикла
        LEA     SI,MULTCND+3
        LEA     DI,PRODUCT+4
        AND     MULTPLR,0FH     ;Удалить ASCII-тройку
A20:
        MOV     AL,[SI]         ;Загрузить ASCII-символ
        AND     AL,OFH          ;Удалить ASCII-тройку
        MUL     MULTPLR         ;Умножить
        AAM                     ;Коррекция для ASCII
        ADD     AL,[DI]         ;Сложить с
        AAA                     ; записанным
        MOV     [DI],AL         ; произведением
        DEC     DI
        MOV     [DI],AH         ;Записать перенос
        DEC     SI
        LOOP    A20             ;Циклиться 4 раза
        RET
MAIN    ENDP
CODESG  ENDS
        END     BEGIN
```

Деление в ASCII-формате

Команда AAD (ASCII Adjust for Division - коррекция для деления ASCII-кодов) выполняет корректировку ASCII-кода делимого до непосредственного деления. Однако, прежде необходимо очистить левые тройк ASCII-кодов для получения распакованного десятичного формата. Команда AAD может оперировать с двухбайтовыми делимыми в регистре AX. Предположим, что регистр AX содержит делимое 3238 в ASCII-формате и регистр CL содержит делитель 37 также в ASCII-формате. Следующие команды выполняют коррекцию для последующего деления:

```
AND  CL,0FH       ;Преобразовать CL в распак.дес.
AND  AX,0F0FH  ;Преобразовать AX в распак.дес.  0208
AAD                  ;Преобразовать в двоичный          001C
DIV  CL              ;Разделить на 7                          0004
```

Команда AAD умножает содержимое AH на 10 (шест.0A), прибавляет результат 20 (шест.14) к регистру AL и очищает регистр AH. Значение 001C есть шест. представление десятичного числа 28. Делитель может быть только однобайтовый от 01 до 09.

```
CODESG  SEGMENT
        ASSUME  CS:CODESG,DS:CODESG,SS:CODESG
        ORG     100H
BEGIN:  JMP     SHORT MAIN
; ----------------------------------------------
DIVDND  DB      '3698'          ;Элементы данных
DIVSOR  DB      '4'
QUOTNT  DB      4 DUP(0)
; ----------------------------------------------
MAIN    PROC    NEAR
        MOV     CX,04           ;4 цикла
        SUB     AH,AH           ;Стереть левый байт делимого
        AND     DIVSOR,0FH      ;Стереть ASCII 3 в делителе
        LEA     SI,DIVDND
        LEA     DI,QUOTNT
A20:
        MOV     AL,[SI]         ;Загрузить ASCII байт
                                ; (можно LODSB)
        AND     AL,0FH          ;Стереть ASCII тройку
        AAD                     ;Коррекция для деления
        DIV     DIVSOR          ;Деление
        MOV     [DI],AL         ;Сохранить частное
        INC     SI
        INC     DI
        LOOP    A20             ;Циклиться 4 раза
        RET
MAIN    ENDP
CODEGS  ENDS
```

ДВОИЧНО-ДЕСЯТИЧНЫЙ ФОРМАТ (BCD)

В предыдущем примере деления в ASCII-формате было получено частное 00090204. Если сжать это значение, сохраняя только правые цифры каждого байта, то получим 0924. Такой формат называется двоично-десятичным (BCD - Binary Coded Decimal) (или упакованным). Он содержит только десятичные цифры от 0 до 9. Длина двоично-десятичного представления в два раза меньше ASCII-представления.

Заметим, однако, что десятичное число 0924 имеет основание 10 и, будучи преобразованным в основание 16 (т.е. в шест. представление), даст шест.039C.

Можно выполнять сложение и вычитание чисел в двоично-десятичном представлении (BCD-формате). Для этих целей имеются две корректирующих команды:

DAA (Decimal Adjustment for Addition - десятичная коррекция для сложения)
DAS (Decimal Adjustment for Subtraction - десятичн. коррекция для вычит.)

Обработка полей также осуществляется по одному байту за одно выполнение. В примере программы выполняется преобразование чисел из ASCII-формата в BCD-формат и сложение их.

Процедура B10CONV преобразует ASCII в BCD. Обработка чисел может выполняться как справа налево, так и слева направо. Кроме того, обработка слов проще, чем обработка байтов, так как для генерации одного байта BCD-кода требуется два байта ASCII-кода. Ориентация на обработку слов требует четного количества байтов в ASCII-поле.

Процедура C10ADD выполняет сложение чисел в BCD-формате. Окончательный результат - 127263.

```
CODESG  SEGMENT PARA "Code"
        ASSUME  CS:CODESG,DS:CODESG,SS:CODESG
        ORG     100H
BEGIN:  JMP     SHORT MAIN
; ---------------------------------------
ASC1    DB      '057836'
ASC2    DB      '069427'
BCD1    DB      '000'
BCD2    DB      '000'
BCD3    DB      4 DUP(0)
; ---------------------------------------
MAIN    PROC    NEAR
        LEA     SI,ASC1+4       ;Инициализировать для ASC1
        LEA     DI,BCD1+2
        CALL    B10CONV          ;Вызвать преобразование
        LEA     SI,ASC2+4       ;Инициализировать для ASC2
        LEA     DI,BCD2+2
        CALL    B10CONV          ;Вызвать преобразование
        CALL    C10ADD          ;Вызвать сложение
        RET
MAIN    ENDP
```

```
;           Преобразование ASCII в BCD:
;           -----------------------
B10CONV PROC
        MOV     CL,04           ;Фактор сдвига
        MOV     DX,03           ;Число слов
B20:
        MOV     AX,[SI]         ;Получить ASCII-пару
       XCHG     AH,AL
        SHL     AL,CL           ;Удалить тройки
        SHL     AX,CL           ; ASCII-кода
        MOV     [DI],AH         ;Записать BCD-цифру
        DEC     SI
        DEC     SI
        DEC     DI
        DEC     DX
        JNZ     B20
        RET
B10CONV ENDP
```

```
;           Сложение BCD-чисел:
;           -----------------
C10ADD PROC
      XOR     AH,AH             ;0чистить AH
      LEA     SI,BCD1+2         ;Инициализация
      LEA     DI,BCD2+2         ; BCD
      LEA     BX,BCD3+3          ; адресов
      MOV      CX,03            ;Трехбайтные поля
      CLC
C20:
      MOV      AL,[SI]          ;Получить BCD1 (или LODSB)
      ADC      AL,[DI]          ;Прибавить BCD2
      DAA                       ;Десятичная коррекция
      MOV      [BX],AL           ;Записать в BCD3
      DEC      SI
      DEC      DI
      DEC      BX
      LOOP     C20              ;Цикл 3 раза
      RET
C10ADD ENDP

CODESG ENDS
      END      BEGIN
```

Обработка строк

- 1. В мнемонике команд обработки строк всегда содержится буква S (String — строка). Она является последней или предпоследней буквой.
- 2. Содержимое строки для микропроцессора НЕ имеет никакого значения. Это могут быть символы, числа и все, что угодно. Основное, что имеет значение, — это длина операнда.
- 3. Строка в **базовом Ассемблере** может обрабатываться побайтно **(BYTE** —последняя буква в команде будет B) или пословно **(WORD** — последняя буква в команде будет W).
- 4. Строка может обрабатываться группой (цепочкой), тогда перед командой появляется префикс **REPx** (REPeat — повторить). Количество повторений должно находиться в регистре CX. Этот префикс алгоритмически подобен команде **LOOPx**

- 5. *Строка-приемник* должна находиться **обязательно** в дополнительном сегменте памяти ES со смещением DI (адресация <ES:DI>).
- 6. *Строка-источник* должна находиться в сегменте данных DS со смещением SI (адресация <DS:SI>). Допускается замена регистра сегмента DS с помощью префикса замены сегмента.

7. В процессе *циклического* выполнения команд *указатели* SI и DI автоматически модифицируются в зависимости от длины элемента строки и значения флага направления DF:

Если <DF>=0, значения SI и DI увеличиваются **(строка обрабатывается слева направо** — в сторону **больших** адресов).

Если <DF>=1, значения SI и DI уменьшаются **(строка обрабатывается справа налево** — в сторону **меньших** адресов).

8. Флаг направления DF очищается или устанавливается, соответственно, командами **CLD или STD**

9. Длина строки в **базовом Ассемблере** <=64К байт.

Команда	Назначение	Алгоритм работы
Команды пересылки MOVSx		
MOVS *приемник, источник*	Копирование строки	*Приемник* <== *источник*
MOVSB	Копирование строки байтов	[DI] <== [SI]
MOVSW	Копирование строки слов	
Команды сравнения CMPSx		
CMPS *приемник, источник*	Сравнение строк	*Приемник* ~ *источник*
CMPSB	Сравнение строк байтов	[DI] ~ [SI]
CMPSW	Сравнение строк слов	
Команды сканирования SCASx		
SCAS *приемник*	Сканирование строки	*Приемник* ~ **AX** (или **AL**)
SCASB	Сканирование строки байтов	[DI] ~ AL
SCASW	Сканирование строки слов	[DI] ~ AX

Команды загрузки LODSx		
LODS *источник*	Чтение из строки	*Источник* ==> **AX (AL)**
LODSB	Чтение байта из строки	[SI] ==> AL
LODSW	Чтение слова из строки	[SI] ==> AX
Команды сохранения STOSx		
STOS *приемник*	Запись в строку	**AX** (или **AL**) ==> *приемник*
STOSB	Запись байта в строку	AL ==> [DI]
STOSW	Запись слова в строку	AX ==> [DI]
Префиксы повторения REPx		
REP	Повторять команду	
REPE/REPZ	Повторять команду, пока равно (флаг ZF=1)	
REPNE/REPNZ	Повторять команду, пока НЕ равно (флаг ZF=0)	

- MOVS - переслать один байт или одно слово из одной области памяти в другую;
- LODS - загрузить из памяти один байт в регистр AL или одно слово в регистр AX;
- STOS - записать содержимое регистра AL или AX в память;
- CMPS - сравнить содержимое двух областей памяти, размером в один байт или в одно слово;
- SCAS - сравнить содержимое регистра AL или AX с содержимым памяти.

В примере выполняется пересылка 20 байт из STRING1 в STRING2. Предположим, что оба регистра DS и ES инициализированы адресом сегмента данных:

```
STRING1   DB   20 DUP('*')
STRING2   DB   20 DUP(' ')
                  ...
    CLD                    ;Сброс флага DF
   MOV  CX,20              ;Счетчик на 20 байт
    LEA  DI,STRING2        ;Адрес области "куда"
    LEA  SI,STRING1        ;Адрес области "откуда"
    REP MOVSB              ;Переслать данные
```

Обработка одномерных массивов

- **Для того чтобы обрабатывать массив, нужно знать, где он хранится (его начальный аарес), длину его элементов и их число. Как и в языке C/C++, имя массива в Ассемблере является также и его начальным адресом.**
- ***Режим адресации с индексацией* вида *имя_массива[регистр_индекс]***

  позволяет обрабатывать каждый элемент массива. В качестве *регистра__индекса* можно брать любой допустимый для косвенной адресации регистр, например, регистр BX.

```
EXTRN C ArrI:WORD, ArrF:DWORD
```

Байты в ОЗУ	1	2	3	4	5	6	7	8	…	17	18	19	20
Индекс-переменная (i)	0				1				…	4			
Элементы массива (значения)	ArrF[0]				ArrF[1]				…	ArrF[4]			
Адреса (смещения)	ArrF	ArrF +1	ArrF +2	ArrF +3	…				…	ArrF +16	ArrF +17	ArrF +18	ArrF +19
Индекс-регистр (смещение)	BX_0				BX_0+4				…	BX_0+4*i			

РИС. 9.3. *Основные характеристики вещественного массива float ArrF [5] в Ассемблере.*

В общем случае, если взять в качестве индекс-регистра регистр **BX**, доступ к любому элементу одномерного массива **Array [i]** длины **Larray** подчиняется в Ассемблере следующей закономерности:

Array[i] → Array+BX_i = Array[BX_i],

где

BX_i = BX_0 + Larray*i = BX_{i-1} + Larray;

BX_0 =0; i=0,...,n-1; n — длина массива Array.

Двумерные массивы

Для двухмерных массивов (матриц) идея будет та же самая, только нужно определиться, как такой массив будет располагаться в оперативной памяти: по строкам или по столбцам (память-то линейная!). Соответственно, и индексных регистров тоже будет два. А также два цикла: внешний и внутренний. Значит, и вычислений прибавится.

Например, пусть имеется некая матрица Matr[M] [N] и в памяти она располагается по строкам: сначала N элементов первой строки, потом N второй строки и т.д. Длину элемента обозначим Larr

Тогда адрес элемента Matr[I,j] будет равен Matr+N*i* Larray+j, где i=0,...,M-1; j=0….N-1. Выделим в Ассемблере для хранения величины N*i* Larray регистр BX, а для j регистр SI (или DI). Тогда Matr[BX] будет означать начальный адрес строки i, а Matr[BX][SI] (Matr[BX+SI] или Matr+[BX+SI] — эти три записи равнозначны) — адрес элемента j в этой строке, т.е.

Matr[i j] - Matr[BX] [SI]

Байты	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...	97	98	99	100
Столбцы (j)	0		1		2		3		4		0		1		...	3		4	
Индекс-регистр SI	0		+2		+4		+6		+8		0		+2		...	+6		+8	
Строки (i)	0										1				I	9			
Индекс-регистр BX	0										+10				+(10*I)	+(10*9)=+90			

РИС. 9.4. *Основные характеристики целочисленной матрицы A [10][5] в Ассемблере.*

```
; переход на следующую строку
        add   bx,N
; восстановить счетчик ВНЕШНЕГО цикла из стека
        pop   cx
        LOOP  @@1 ;=====================================
@@4:
        ret
SummA2  EndP
        End
```

Математический сопроцессор

- **Для обработки данных с плавающей точкой служит специальное устройство — *математический сопроцессор* (FPU — Floating Point Unit). С момента своего возникновения сопроцессор расширял вычислительные возможности основного процессора i8086 (i80286, i80386, i80486) и сначала был выполнен в виде отдельной микросхемы i8087 (i80287, i80387, i80487). Его присутствие в первых моделях процессора было не обязательным. Если сопроцессора не было, то его команды можно было *эмулировать* программным путем, что немного ухудшало производительность основного процессора. Начиная с семейства процессоров i486DX, сопроцессор стал составной частью основного процессора .**
- **Современный сопроцессор обеспечивает полную поддержку стандартов IEEE-754 и IEEE-854 по представлению и обработке чисел с плавающей точкой. Он может выполнять *трансцендентные операции* (вычисление тригонометрических функций, логарифмов и проч.) с большой точностью.**

Особые числа

- Кроме обычных чисел, спецификация стандарта IEEE предусматривает несколько специальных форматов, которые могут получиться в результате выполнения математических операций сопроцессора.
- **Положительный ноль** — все биты числа сброшены в ноль:
- **Отрицательный ноль** — знаковый бит равен 1, остальные биты числа сброшены в ноль.
- **Положительная бесконечность** — знаковый бит равен 0, все биты экспоненты установлены в 1, а биты мантиссы сброшены в 0.
- **Отрицательная бесконечность** — знаковый бит равен 1, все биты экспоненты установлены в 1, а биты мантиссы сброшены в 0.
- **Денормализованные числа** — все биты экспоненты сброшены в 0. Эти числа позволяют представлять очень маленькие числа при вычислениях с расширенной точностью.
- **Неопределенность** — знаковый бит равен 1, первый бит мантиссы равен 1 (для 80-разрядных чисел первые два бита равны 11), остальные биты мантиссы сброшены в ноль, все биты экспоненты установлены в **1.**
- **Не-число типа SNAN (сигнальное)**
- **Не-число типа QNAN (тихое)**
- **Неподдерживаемое число** — все остальные ситуации

Регистры: 8-дынные, 5-вспомогательные

R6 - (TOP) вершина стека ST или ST(0) (80 бит)	
	Расширенное вещественное или любое другое допустимое данное сопроцессора
R7 - ST(1) (80 бит)	
	Расширенное вещественное или любое другое допустимое данное сопроцессора

Регистры данных

Регистры данных сопроцессора **R0-R7** имеют длину 80 бит (т.е. пять 16-разрядных слов) и рассматриваются как **круговой стек,** вершина которого **(TOP)** называется ST или **ST(0)** и является плавающей. Принцип работы с круговым стеком сопроцессора аналогичен обычному калькулятору. Любая команда загрузки данных сопроцессора автоматически перемещает вершину стека сопроцессора: **TOP=TOP+1.** На рис. 13.1 показана гипотетическая ситуация, когда в результате выполнения какой-то команды вершиной стека стал регистр R6. Остальные регистры распределяются подряд по кругу: **R7-ST(1), R0-ST(2),...,R5-ST(7).** Это и есть их текущие имена ST(i), i=I,...,7 на момент выполнения данной команды сопроцессора. Если в этих регистрах есть данные, то они могут служить операндами в командах сопроцессора. Обращаться напрямую к регистрам **R0-R7** нельзя.

Система команд

Система команд сопроцессора достаточно простая, если знать ключ и немного понимать английский язык. Для их подключения нужно сделать следующее.

Воспользоваться одной из директив Ассемблера: .8087, .287 (.286p), .387(.386 .487 (.486p). Необходимо иметь в виду, что не все команды сопроцессора, к сожалению, совместимы сверху вниз. Кроме того, директива использования процессора **.x86p** предполагает компиляцию и работу программы в 32-разрядном режиме. Поэтому результаты расчета, выполненные с использованием команд младших моделей сопроцессора, могут отличаться от результатов, полученных на старших моделях.

Сделать инициализацию сопроцессора с помощью команды FINIT

При компиляции использовать ДОПОЛНИТЕЛЬНЫЙ ключ /r или /e (Emulated or Real floating-point instructions). Таким образом, теперь вызов компилятора Ассемблера для стыковки с C++ может иметь следующий вид:

tasm.exe Name.asm /l /r /ml

13.4.3. Команды загрузки констант

Команды этой группы помещают в вершину стека **ST(0)** часто используемые константы. Начиная с сопроцессора **i80387**, *эти константы хранятся в более точном формате.* Команды операндов не имеют.

Таблица 13.6. Команды загрузки констант.

Команда	Назначение
FLD1	Поместить в ST(0) число 1.0
FLDZ	Поместить в ST(0) число +0.0
FLDPI	Поместить в ST(0) число π
FLDL2E	Поместить в ST(0) число, равное $\log_2 e$
FLDL2T	Поместить в ST(0) число, равное $\log_2 10$
FLDLN2	Поместить в ST(0) число, равное $\ln 2$
FLDLG2	Поместить в ST(0) число, равное $\lg 2$

Арифметические команды

Таблица 13.7. Команды базовой арифметики.

Команда	Тип данных	Алгоритм выполнения
Команды сложения		
FADD приемник, источник	Вещественное	Приемник = приемник + источник
FADDP приемник, источник	Вещественное	
FIADD источник	Целое	
Команды обычного вычитания		
FSUB приемник, источник	Вещественное	Приемник = приемник - источник
FSUBP приемник, источник	Вещественное	
FISUB источник	Целое	
Команды реверсного (обратного) вычитания		
FSUBR приемник, источник	Вещественное	Приемник = источник – приемник
FSUBRP приемник, источник	Вещественное	
FISUBR источник	Целое	
Команды умножения		
FMUL приемник, источник	Вещественное	Приемник = приемник * источник
FMULP приемник, источник	Вещественное	
FIMUL источник	Целое	

Команды обычного деления		
FDIV приемник, источник	Вещественное	Приемник = приемник/источник
FDIVP приемник, источник	Вещественное	
FIDIV источник	Целое	
Команды реверсного (обратного) деления		
FDIVR приемник, источник	Вещественное	Приемник = источник/приемник
FDIVRP приемник, источник	Вещественное	
FIDIVR источник	Целое	

```
    ; Вычислить действительные корни квадратного
    уравнени¤:
    ;           a*x*x + b*x + c = 0
    .387
    .model large,C
    .data
EXTRN C
    a:Dword,b:Dword,c:Dword,x1:dword,x2:dword,d:dword
EXTRN C ac:Dword,bb:Dword
    .code
    public  quadr
four    dd  4.
two     dd  2.
```

```
quadr  proc C far
   finit                  ;иниц. 8087
                          ;-------ST(0)-------!------ST(1)------!
   fild   b                     ;b                    !?
   fmul    st(0),st(0)   ;b*b                  !?
   FST     bb      ;копирование вершины стека ==> bb
   fild  a               ;a                  !b*b
   fmul    four            ;4*a                !b*b
   fimul   c             ;4*a*c              !b*b
   FST     ac      ;копирование вершины стека ==> ac

   fsubP   st(1),st(0) ;d=b*b-4*a*c     !?
   fst   d           ;копирование вершины стека ==> d
```

```
    fsqrt               ;sqrt(d)        !?
    fld    st(0)              ;sqrt(d)              !sqrt(d)
    fchs                          ;-sqrt(d)             !sqrt(d)
   fiadd    b                     ;b-sqrt(d)           !sqrt(d)
    fchs                ;-b+sqrt(d)         !sqrt(d)
    fxch  st(1)              ;sqrt(d)              !-b+sqrt(d)
   fiadd    b                     ;b+sqrt(d)          !-b+sqrt(d)
    fchs                ;-b-sqrt(d)         !-b+sqrt(d)
   fidiv     a                    ;-b-sqrt(d)/a       !-b+sqrt(d)
   fdiv     two                   ;-b-sqrt(d)/a/2     !-b+sqrt(d)
    fstp   x2           ;-b+sqrt(d)          !?
   fidiv     a                    ;-b+sqrt(d)/a       !-b+sqrt(d)
   fdiv     two                   ;-b+sqrt(d)/a/2     !?
    fstp   x1           ;?                    !?
    ret
quadr    endp
    end
```

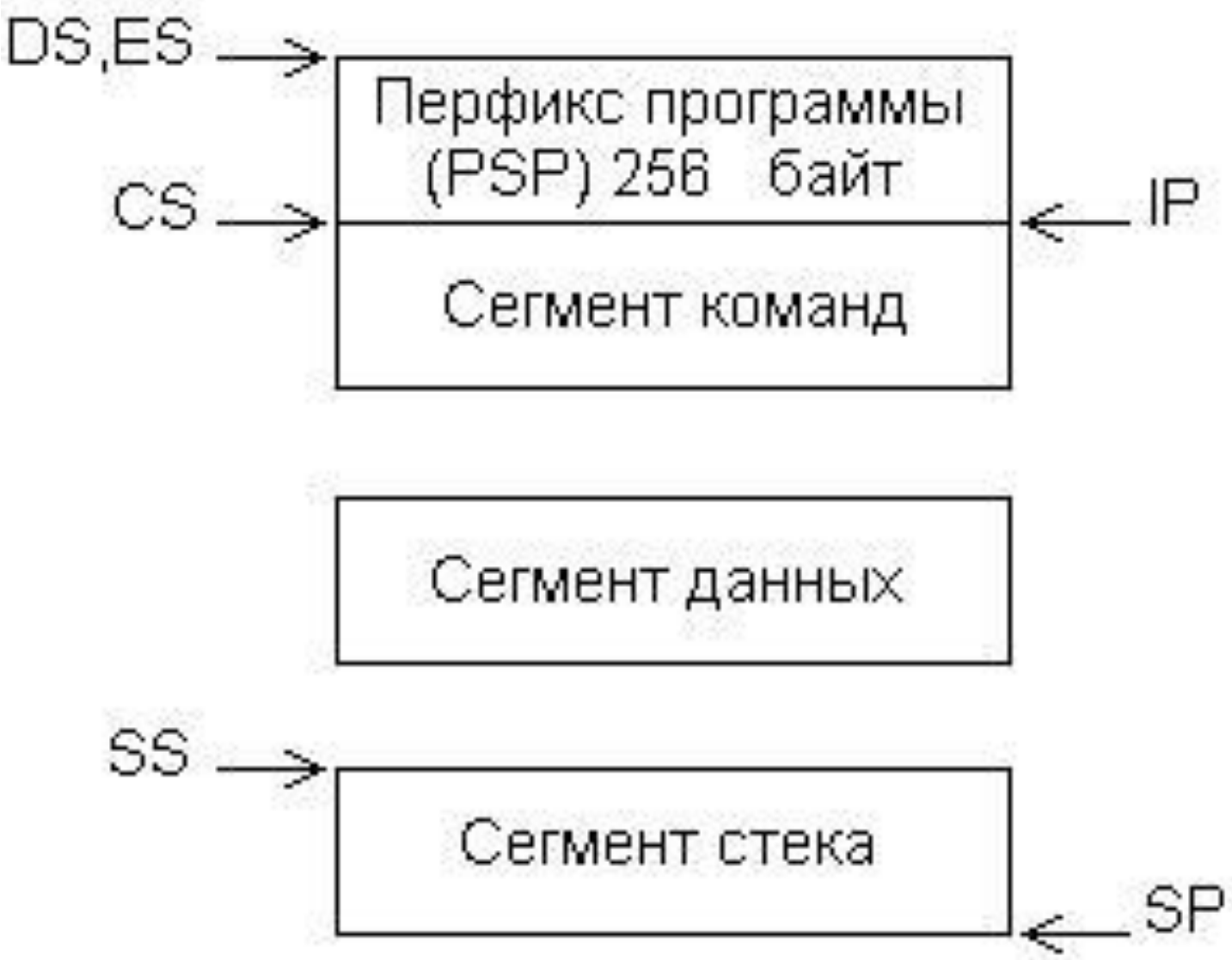
Распределение программы в памяти

- Рассмотрим распределение памяти на примере простейшей программы.

```
;Данные программы
DATA SEGMENT
MSG  DB   ‘Текст$’
DATA ENDS
STK SEGMENT STACK
DB 256 dup(?)
STK ENDS
;Код программы
CODE       SEGMENT
ASSUME CS:CODE,DS:DATA,SS:STK
START:
  MOV      AX,DATA
  MOV      DS,AX
  MOV      AH,09H          ;Вывод сообщения
  MOV      DX,OFFSET MSG
  INT  21H
  MOV      AH,4CH          ;Завершение работы
  INT  21H
CODE ENDS
END   START
```

В этой программе явно описаны три сегмента – кода с именем CODE, данных с именем DATA и стека с именем STK. Директива ASSUME связывает имена этих сегментов, которые в общем случае могут быть произвольными, с сегментными регистрами CS , DS и SS соответственно. Распределение памяти при загрузке программы на исполнение показано на рисунке

Распределение в памяти EXE программы



Распределение в памяти COM программы

После инициализации в регистре IP находится смещение первой команды программы относительно начала кодового сегмента, адрес которого помещен в регистр CS. Процессор, считывая эту команду, начинает выполнение программы, постоянно изменяя содержимое регистра IP и при необходимости CS для получения кодов очередных команд. DS после загрузки программы установлен на начало PSP, поэтому для его использования в первых двух командах программы выполняется загрузка DS значением сегмента данных.

EXE- и COM-программы

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

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

Файл **COM**-формата – это двоичный образ кода и данных программы. Такой файл может занимать менее 64K.

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

Заголовок EXE-файла

Заголовок EXE-файла состоит из форматированной зоны и таблицы расположения сегментов (Relocation Table). Форматированная зона выглядит следующим образом:

(0) 2	signature	два байта 'MZ' (4Dh, 5Ah), индентифицирующие файл в формате EXE
(+2) 2	part_pag	длина последней страницы программы в байтах (страница содержит 512 байт)
(+4) 2	file_size	размер программы в страницах по 512 байт
(+6) 2	rel_item	число элементов в таблице расположения сегментов

(+8) 2	hdr_size	размер заголовка файла в параграфах (длина параграфа - 16 байт)
(+10) 2	min_mem	минимальное количество памяти в параграфах, которое нужно зарезервировать в памяти за концом загруженной программы
(+12) 2	max_mem	максимальное количество памяти в параграфах, которое нужно зарезервировать в памяти за концом загруженной программы
(+14) 2	ss_reg	величина смещения от начала программы, которая используется для загрузки сегментного регистра стека SS
(+16) 2	sp_reg	величина смещения от начала программы, которая используется для загрузки регистра SP
(+18) 2	chk_summ	контрольная сумма всех слов в файле
(+20) 2	ip_reg	значение для регистра IP, которое будет использовано при начальном запуске программы
(+22) 2	cs_reg	смещение от начала программы для установки сегментного регистра кода CS
(+24) 2	relt_off	смещение от начала файла таблицы расположения сегментов программы
(+26) 2	overlay	номер оверлея, равен 0 для основного модуля

- Таблица расположения сегментов программы начинается сразу после форматированной области и состоит из четырехбайтовых значений в формате "смещение:сегмент".
- Область файла после таблицы расположения сегментов выравнивается на границу параграфа с помощью байта-заполнителя, и дальше начинается сама программа.

Описание структуры заголовка EXE файла и таблицы расположения сегментов

```
typedef struct _EXE_HDR_ {
        unsigned signature;
        unsigned part_pag;
        unsigned file_size;
        unsigned rel_item;
        unsigned hdr_size;
        unsigned min_mem;
        unsigned max_mem;
        unsigned ss_reg;
        unsigned sp_reg;
        unsigned chk_summ;
        unsigned ip_reg;
        unsigned cs_reg;
        unsigned relt_off;
        unsigned overlay;
} EXE_HDR;
typedef struct _RELOC_TAB_ {
        unsigned offset;
        unsigned segment;
} RELOC_TAB;
```

Процесс загрузки программ в память

- Загрузка COM- и EXE-программ происходит по-разному, однако есть некоторые действия, которые операционная система выполняет в обоих случаях одинаково.
- Определяется наименьший сегментный адрес свободного участка памяти для загрузки программы (обычно DOS загружает программу в младшие адреса памяти, если при редактировании не указана загрузка в старшие адреса).
- Создаются два блока памяти - блок памяти для переменных среды и блок памяти для PSP и программы.
- В блок памяти переменных среды помещается путь файла программы.
- Заполняются поля префикса сегмента программы PSP в соответствии с характеристиками программы (количество памяти, доступное программе, адрес сегмента блока памяти, содержащего переменные среды и т.д.)
- Устанавливается адрес области Disk Transfer Area (DTA) на вторую половину PSP (PSP:0080).
- Анализируются параметры запуска программы на предмет наличия в первых двух параметрах идентификаторов дисковых устройств. По результатам анализа устанавливается содержимое регистра AX при входе в программу. Если первый или второй параметры не содержат правильного идентификатора дискового устройства, то соответственно в регистры AL и AH записывается значение FF.
- А дальше действия системы по загрузке программ форматов COM и EXE будут различаться.

Для COM-программ

- Для COM-программ, которые представляют собой двоичный образ односегментной программы, выполняется чтение файла программы с диска и запись его в память по адресу PSP:0100. Вообще говоря, программы типа COM могут состоять из нескольких сегментов, но в этом случае они должны сами управлять содержимым сегментных регистров, используя в качестве базового адреса адрес PSP.
- После загрузки файла операционная система для COM-программ выполняет следующие действия:
- сегментные регистры CS, DS, ES, SS устанавливаются на начало PSP;
- регистр SP устанавливается на конец сегмента PSP;
- вся область памяти после PSP распределяется программе;
- в стек записывается слово 0000;
- указатель команд IP устанавливается на 100h (начало программы) с помощью команды JMP по адресу PSP:100.

Загрузка EXE-программ

- Загрузка EXE-программ происходит значительно сложнее, так как связана с настройкой сегментных адресов:
- Считывается во внутренний буфер DOS форматированная часть заголовка файла.
- Определяется размер загрузочного модуля по формуле:
- size=((file_size*512)-(hdr_size*16)-part_pag
- Определяется смещение начала загрузочного модуля в EXE-файле как hdr_size*16.
- Вычисляется сегментный адрес для загрузки START_SEG, обычно используется значение PSP+100h.
- Загрузочный модуль считывается в память по адресу START_SEG:0000.
- Сканируются элементы таблицы перемещений, располагающейся в EXE-файле со смещением relt_off.

- **Для каждого элемента таблицы:**
- **1. Считывается содержимое элемента таблицы как два двухбайтных слова (OFF,SEG).**
- **2. Вычисляется сегментный адрес ссылки перемещения**
- **REL_SEG = (START_SEG + SEG)**
- **3. Выбирается слово по адресу REL_SEG:OFF, к этому слову прибавляется значение START_SEG, затем сумма записывается обратно по тому же адресу.**
- **Заказывается память для программы, исходя из значений min_mem и max_mem.**
- **Инициализируются регистры, и программа запускается на выполнение.**
- **При инициализации регистры ES и DS устанавливаются на PSP, регистр AX устанавливается так же, как и для COM-программ, в сегментный регистр стека SS записывается значение START_SEG + ss_reg, а в SP записывается sp_reg.**
- **Для запуска программы в CS записывается START_SEG+cs_reg, а в IP - ip_reg. Такая запись невозможна напрямую, поэтому операционная система сначала записывает в свой стек значение для CS, затем значение для IP и после этого выполняет команду дальнего возврата RETF (команда возврата из дальней процедуры).**

Префикс программного сегмента

(0) 2	int 20h	двоичный код команды int 20h (программы могут использовать эту команду для завершения своей работы)
(+2) 2	mem_top	нижняя граница доступной памяти в системе в параграфах
(+4) 1	reserv1	зарезервировано
(+5) 5	call_dsp	команда вызова FAR CALL диспетчера MS-DOS
(+10) 4	term_adr	адрес завершения (Terminate Address)
(+14) 4	cbrk_adr	адрес обработчика Ctrl-Break
(+18) 4	crit_err	адрес обработчика критической ошибки
(+22) 2	parn_psp	сегмент PSP программы, запустившей данную программу (программы-родителя)
(+24) 20	file_tab	таблица открытых файлов, если здесь находятся байты 0FFH, то таблица не используется
(+44) 2	env_seg	сегмент блока памяти, содержащего переменные среды
(+46) 4	ss_sp	адрес стека SS:SP программы

(+50) 2	max_open	максимальное число открытых файлов
(+52) 4	file_tba	адрес таблицы открытых файлов
(+56) 24	reserv2	зарезервировано
(+80) 3	disp	диспетчер функций DOS
(+83) 9	reserv3	зарезервировано
(+92) 16	fcb1	форматируется как стандартный FCB, если первый аргумент командной строки содержит правильное имя файла
(+108) 20	fcb2	заполняется для второго аргумента командной строки аналогично fcb1
(+128) 1	p_size	число значащих символов в неформатированной области параметров, либо буфер обмена с диском DTA, назначенный по умолчанию
(+129) 127	parm	неформатированная область параметров, заполняется при запуске программы из командной строки

Структура PSP

```
typedef struct _PSP_ {
        unsigned char int20h[2];
        unsigned mem_top;
        unsigned char reserv1;
        unsigned char call_dsp[5];
        void far *term_adr;
        void far *cbrk_adr;
        void far *crit_err;
        unsigned parn_psp;
        unsigned char file_tab[20];
        unsigned env_seg;
        void far *ss_sp;
        unsigned max_open;
        void far *file_tba;
        unsigned char reserv2[24];
        unsigned char disp[3];
        unsigned char reserv3[9];
        unsigned char fcb1[16];
        unsigned char fcb2[20];
        unsigned char p_size;
        unsigned char parm[127];
} PSP;
```

- Программы могут получить из PSP такую информацию, как параметры командной строки при запуске, размер доступной памяти, найти сегмент области переменных среды и т.д.
- Как программе узнать адрес своего PSP? Очень просто сделать это для программ, написанных на языке ассемблера: при запуске программы этот адрес передается ей через регистры DS и ES. То есть этот адрес равен DS:0000 или ES:0000 (для COM-программ на PSP указывают также регистры CS и SS).

EXE-программы.

- EXE-программы содержат несколько программных сегментов, включая сегмент кода, данных и стека. В процессе загрузки считывается информация заголовка EXE в начале файла и выполняется перемещение адресов сегментов. Это означает, что ссылки типа
- mov ax,data_seg
- mov ds,ax
- и
- call my_far_proc
- должны быть приведены (пересчитаны), чтобы учесть тот факт, что программа была загружена в произвольно выбранный сегмент.
- После перемещения управление передается загрузочному модулю посредством инструкции далекого перехода (FAR JMP) к адресу CS:IP, извлеченному из заголовка EXE.
- В момент получения управления программой EXE -формата:
- DS и ES указывают на начало PSP
- CS, IP, SS и SP инициализированы значениями, указанными в заголовке EXE
- поле PSP MemTop (вершина доступной памяти системы в параграфах) содержит значение, указанное в заголовке EXE. Обычно вся доступная память распределена программе

COM-программы.

- COM-программы содержат единственный сегмент . Образ COM-файла считывается с диска и помещается в память, начиная с PSP:0100h. В общем случае, COM-программа может использовать множественные сегменты, но тогда она должна сама вычислять сегментные адреса, используя PSP как базу.
- COM-программы предпочтительнее EXE-программ, когда дело касается небольших ассемблерных утилит. Они быстрее загружаются, ибо не требуется перемещения сегментов, и занимают меньше места на диске, поскольку заголовок EXE и сегмент стека отсутствуют в загрузочном модуле.
- После загрузки двоичного образа COM-программы:
- CS, DS, ES и SS указывают на PSP;
- SP указывает на конец сегмента PSP (обычно 0FFFEH, но может быть и меньше, если полный 64K сегмент недоступен);
- слово по смещению 06H в PSP (доступные байты в программном сегменте) указывает, какая часть программного сегмента доступна;
- вся память системы за программным сегментом распределена программе;
- слово 00H помещено (PUSH) в стек.
- IP содержит 100H (первый байт модуля) в результате команды JMP PSP:100H.

Выход из программы

- Завершить программу можно следующими способами:
- через функцию 4CH (EXIT) прерывания 21H в любой момент, независимо от значений регистров;
- через функцию 00H прерывания 21H или прерывание INT 20H, когда CS указывает на PSP.
- Функция DOS 4CH позволяет возвращать родительскому процессу код выхода, который может быть проверен вызывающей программой или командой COMMAND.COM "IF ERRORLEVEL".
- Можно также завершить программу и оставить ее постоянно резидентной (TSR), используя либо INT 27H , либо функцию 31H (KEEP) прерывания 21H. Последний способ имеет те преимущества, что резидентный код может быть длиннее 64K, и что в этом случае можно сформировать код выхода для родительского процесса.

Машинные коды

Машинным кодом будем называть внутреннее представление команд Ассемблера в памяти компьютера, т.е. **собственно команды процессора Intel.** Эти команды имеют определенную структуру, состоящую в общем случае из восьми элементов — полей. Каждое поле имеет длину 1 байт. В машинном коде ключевым полем, является **байт кода операции** (поле 3), остальные элементы являются необязательными. Максимальное число полей реальной команды обычно не превышает шести.

Байт кода операции

Что же собой представляет байт кода операции? Как процессор знает, что ему надо делать? Байт кода операции может иметь несколько модификаций. Рассмотрим их и укажем основные операции, которые они поддерживают. Кодировка кода операции для различных команд Ассемблера приведена в табл. П7.1.

Биты	7	6	5	4	3	2	1	0
	КОП						d	w
							s	w
							v	w
								z

Вариант 1 — большинство двухадресных команд.

Биты 2-7 определяют информационную часть кода операции (КОП).

Бит 0 может интерпретироваться двояко:

- **длина операндов:**
  w=1 — обрабатывается слово; **w=0** — обрабатывается байт,
- **значение флага ZF при использовании префикса повторения REP:**
  z=1 — флаг **ZF=1** (команды **REPZ/REPE**);
  z=0 — флаг **ZF=0** (команды **REP/REPNE**).

Бит 1 — может иметь три интерпретации

- **задавать бит направления передачи информации** (в этом случае в команде ОБЯЗАТЕЛЬНО присутствует *байт способа адресации* — см. п. 8.1.2, рис. 8.7):
  d=1 — r/m ===> reg;
  d=0 — r/m <=== reg. Значение reg определяется по табл. 8.1.
- **бит размера непосредственного операнда:**
  s=1 — длина данного 8 бит;
  s=0 — длина данного 16 бит,

- **бит, определяющий значение счетчика в циклических командах:**
  v=1 — значение счетчика находится в регистре CL;
  v=0 — значение счетчика равно 1.

Таблица 8.1. Кодировка регистров.

reg	w=0	w=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Таблица 8.2. Кодировка сегментных регистров.

rs	Сегментные регистры
00	ES
01	CS
10	SS
11	DS

Биты	7	6	5	4	3	2	1	0
	КОП							w

РИС. 8.6. *Вариант 5 — команды загрузки аккумулятора AX (AL) в память и наоборот, работа с портами, арифметические операции с непосредственной адресацией и аккумулятором AX (AL), команды деления и умножения.*

- **Строка 19 0006 B8 0002 mov ax,2**

Адрес (смещение) данной команды равен 0006. Пока адрес нам НЕ нужен.

КОП = B8. Он соответствует команде **MOV AX,im16** (см. Табл. П7.1). Команда имеет *непосредственный режим адресации* (16-разрядная константа непосредственно встраивается в машинную команду). Распишем КОП в двоичной системе счисления: 1011 1000 и попробуем понять, какой из вариантов байта кода операции нам подходит (см. п. 8.1.1). Первый вариант нам НЕ подходит, поскольку **бит 0 (w)=0** (наша команда НЕ работает с 8-разрядными данными). Вариант 2 нам подходит: **w=1, reg=000=AX** (см. табл. 8.1):

Биты	7	6	5	4	3	2	1	0
	1	0	1	1	1	0	0	0
	КОП				w	reg		