# Лекция 1

## Вступление

Раньше мы рассматривали языки программирования высокого уровня. Преимущества этих языков - приближение к широкому пользователю, удобство программирования сложных логических задач.

Недостатки - сравнительно невысокая эффективность машинных программ. Все возможности ЭВМ полностью не используются. Поэтому для программирования системных задач, управления работой оборудования такие языки не достаточно эффективны.

Здесь преимущество имеют машинный ориентированные языки, одними из которых есть ассемблеры. Что такое ассемблер?

После трансляции программы из языка высокого уровня она превращается в последовательность машинных команд. Одна первичная команда реализуется несколькими машинными. Все имена пользователя (переменные, метки) заменяются адресами. Например, какая-то двухадресная команда имеет вид:

```
256 4725 0648
```

где первые три цифры - это код команды, дальше адрес первого операнда и адрес второго операнда.

В языке ассемблер коды команд и адреса заменяются их символическими именами. Например:

```
MOV X,Y
```

Это облегчает работу программиста - не нужно помнить машинные коды команд, самому заниматься делением памяти. Кроме того, ассемблер имеет средства модульного программирования - процедуры, макрокоманды.

Одна ассемблерная команда транслируется в одну машинную. Потому программы на ассемблере имеют намного больший объем, чем на языках высокого уровня. Но с помощью ассемблера можно использовать все возможности ЭВМ и эффективность выполнения таких программ будет очень высокой. Ясно, что программировать на ассемблере сложные логические программы будет очень трудоемким и потому нецелесообразным занятием. Назначение ассемблера для современных машин - системное программирование, для согласования отдельных программных систем, управления работой оборудования, систем передачи данных, и т.п.

| ЭВМ | Розрядность | Максим. память, Мб |
|---|---|---|
| XT 8088/86 | 16 | 1 |
| AT 80286 | 16 | 16 |
| A 80386 DX | 32 | 1024 |
| 80386 SX | 16 | 16 |
| 80386 SL | | |

Не случайно, что программная оболочка Norton Commander состоит из 20 тыс. строк языком С и 10 тыс. - на ассемблере..

Если универсальные языки программирования незначительной мерой зависят от типа ЭВМ, то ассемблер полностью связан с конкретным типом. Необходимо знать особенности архитектуры определенного класса ЭВМ, то есть, высшей квалификации программиста.

С другой стороны, изучение ассемблера позволяет познакомиться со всеми возможностями ЭВМ, глубже понять, как в действительности реализуется программа на машинном уровне.

# Раздел 1

# Особенности архитектуры ПЭВМ на базе микропроцессоров 8088/86

## 1.1. Структура ПЭВМ

ПЭВМ состоит из нескольких функциональных устройств, которые реализуют арифметические и логические операции, управления, запоминания, ввода/вывода данных.

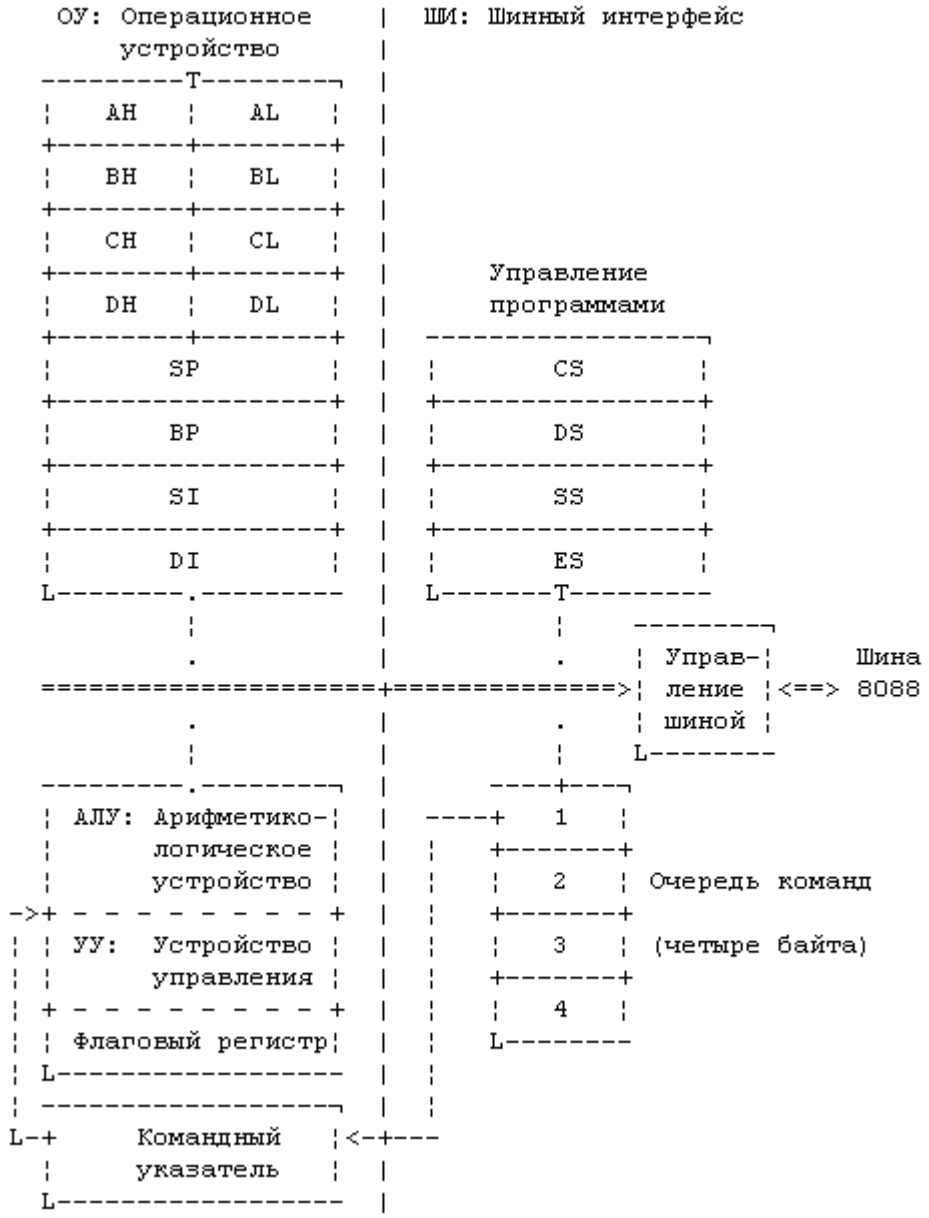



Рис.1.1. Операционное устройство и шинный интерфейс

Как показано на рис.1.1 процессор разделен на две части: операционное устройство (ОУ) и шинный интерфейс (ШИ). Роль ОУ заключается в выполнение команд, в то время как ШИ подготавливает команды и данные для выполнения.

Операционное устройство содержит арифметико-логическое устройство (АЛУ), устройство управления (УУ) и 14 регистров. Эти устройства обеспечивают выполнение команд, арифметические вычисления и логические операции (сравнение на больше, меньше или равно). Программы операционной системы и программы, которые выполняются, находятся в оперативной памяти (ОЗП).

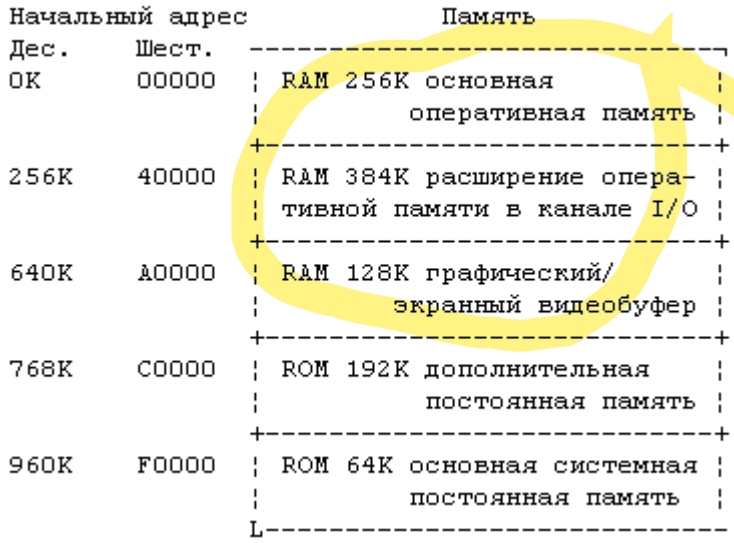



Рис.1.2. Карта физической памяти

Память RAM включает *первые три четверти* памяти, а ROM -- последнюю четверть. Первые 256К памяти находятся на системной плате. Для программиста доступные первые 640 К памяти.

### *1.1.1. Центральный процессор*

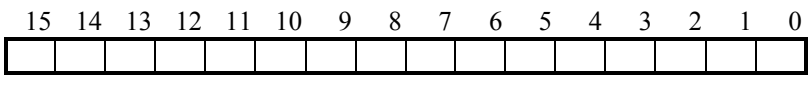
Как отмечалось, АЛУ и УУ объединяются в одном кремниевом кристалле и составляют центральный процессор (ЦП).

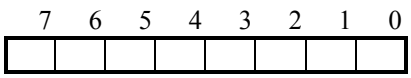
АЛУ выполняет основные арифметические операции (добавление, вычитание) и другие, логические операции. Эти действия реализуются специальными устройствами, например, сумматорами. Если бы между сумматором и ОЗУ не было никаких вспомогательных устройств, то нужно было бы очень много пересылок. Для уменьшения их количества В ЭВМ применяются специальные промежуточные ЗУ - *регистры*.

### *1.1.2. Память и особенности ее использования*

Память состоит из отдельных ячеек. Внутри ЭВМ данные подаются в двоичной системе счисления. Поэтому ячейка состоит из ряда двоичных разрядов (*битов*), в каждом из которых записано 1 или 0.

8 бит составляют *байт*, который является наименьшей адресованной единицей данных. Ячейка состоит из одного *слова*, или двух байтов. Все разряды слова или регистра нумеруются справа налево от 0 до 15. В байте разряды нумеруются от 0 до 7.



Операции можно выполнять как над словами, так и над байтами. Каждая такая ячейка памяти определяется местом в памяти - *адресом* и данными, которые содержатся в ней, то есть, *содержанием*.

Поскольку байт является наименьшей адресованной единицей данных, то, чтобы к нему добраться, нужно иметь его адрес или номер. Все байты нумеруются от 0. В слове правый байт имеет меньший адрес, а левый - больший. Поэтому и называются соответственно -- *младший и старший*. Адрес слова совпадает с адресом его младшего байта. Следовательно, байты нумеруются специально от 000, а слова - лишь парной нумерацией. То есть, слово не может иметь непарного адреса.

Наибольший номер, который можно записать до 16 битов, это будет $2^{16}$-1, а учитывая нулевой номер - $2^{16}$. Следовательно, с использованием 16-ти битов можно пронумеровать $2^{16} = 2^6 * 2^{10}$ = 64 Кбайт. В 16-ой системе это будет наибольший адрес 0FFFFH.

Вся память разделяется на *сегменты* не больше 64 Кбайт. Одновременно можно использовать не больше 4 сегментов. Каждая ячейка будет определяться своим адресом относительно начала сегмента - смещением или *исполнительным адресом*. Адреса начала сегмента сохраняются в *регистрах сегментов*. Адрес слова в памяти будет равняться сумме исполнительного адреса и адреса сегмента (IP + CS) и называется *абсолютным адресом*. Адреса сегмента и исполнительная занимают 16 бит, а абсолютный адрес - 20 бит. Это достигается специальным превращением.

### *1.1.3. Регистры ЭВМ*

Рассмотрим особенности использования отдельных регистров.

*Регистр* - это промежуточная память, вместимостью одно машинное слово (16 бит). В ЦП существуют много регистров, большинство из которых недосягаемые для программиста. Но есть несколько регистров, которые используются в программах на языке ассемблер.

За ними закреплены названия и имена, через которые к ним можно обращаться:

1. 4 регистра общего назначения (**AX, BX, CX, DX**).
2. 4 регистра указатели -**SP, BP, SI, DI**.
3. 4 сегментных регистра - **CS, DS, SS, ES**.
4. 2 управляющих регистра - указателя команд **IP** и регистр флагов **F**.

Назначение отдельных регистров и особенности использования рассмотрим дальше.

Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде.

Адреса дополнительных сегментов данных должны содержаться в регистрах **ES, GS, FS** (extension data segment registers).

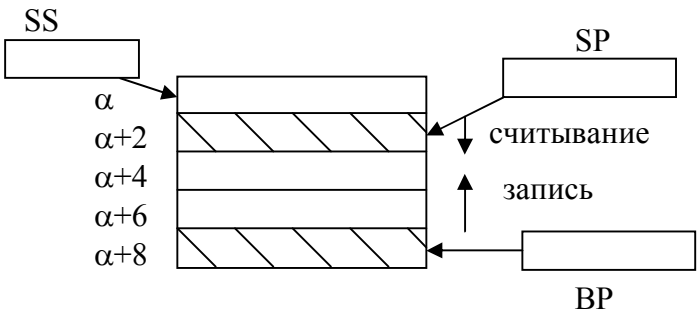



Рис. 1.3. Модель работы стека

В современных ЭВМ широко используются *стеки*, например, при работе с подпрограммами, при обработке прерываний, а также для запоминания промежуточных результатов. *Стек* - это участок памяти, которая заполняется в сторону меньших адресов, а освобождается в сторону увеличения адресов. Здесь реализуется дисциплина “последний пришел - первый обслуживайся”. Пример стека - пачка листов бумаги на столе.

Стек характеризуется своим базовым адресом, который записывается в базовый регистр SS, и адресом вершины, которая записывается в регистр указателя стека SP (stack pointer). В стек записываются слова (2 байта).

# Лекция 2

### *1.1.4. Регистры состояния и управления*

В микропроцессор включены несколько регистров (см. рис. 1.1.4), которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. К этим регистрам относятся:

1. регистр флагов eflags/flags;
2. регистр указателя команды eip/ip.

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора. Рассмотрим подробнее назначение и содержимое этих регистров:

**eflags/flags** (flag register) — регистр флагов. Разрядность eflags/flags — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру flags для i8086.

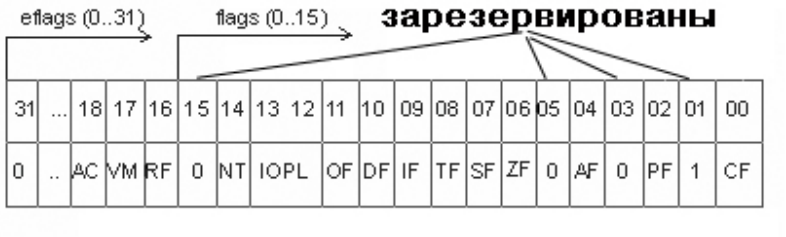



Рис. 1.1.4. Регистр флагов

Таблица

Назначение бит регистра флагов

| Мнемоника флага | Флаг | № бита в eflags | Содержание и назначение |
|---|---|---|---|
| **CF** | Флаг переноса (Carry Flag); Переполнение | 0 | 1 — указывает на переполнение старшего бита при арифметических командах. Старшим является 7, 15, 31-й бит в зависимости от размерности операнда |
| **PF** | Флаг чётности (Parity Flag) | 2 | 1 — 8 младших разрядов (только для 8 младших разрядов) результата содержат четное число единиц |
| **AF** | Флаг коррекции (Adjust Flag) | 4 | 1- если арифметическая операция производит перенос или заём в/из 3-й бит результата, иначе - сбрасывается. Этот флаг используется для двоично-кодированной десятичной (BCD - Binary-Coded Decimal) арифметики |
| **ZF** | Флаг нуля (Zero Flag) | 6 | 1 — результат нулевой;<br>0 — результат ненулевой |
| **SF** | Флаг знака (Sign Flag) | 7 | Отражает состояние старшего бита результата (биты 7, 15, 31 для 8, 16, 32-разрядных операндов соот-но):<br>1 — старший бит результата равен 1;<br>0 — старший бит результата равен 0 |
| **TF** | Флаг ловушки (Trap Flag) | 8 | 1 - то процессор использует покомандную отладку текущей программы;<br>0 - программа выполняется обычным образом |
| **IF** | Флаг | 9 | 1 - то в ответ на IRQ процессор генерирует |

| | | | |
|---|---|---|---|
| | разрешения прерываний (Interrupt enable Flag) | | прерывания;<br>0 - процессор не отвечает на них (но не игнорирует) |
| **DF** | Флаг направления (Direction Flag) | 10 | 1 - строковые команды обрабатывают строки данных, переходя от младших адресов к старшим;<br>0 - то в обратном направлении |
| **OF** | Флаг переполнения (Overflow Flag) | 11 | 1 — в результате операции происходит перенос (заем) в(из) старшего, знакового бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно);<br>Флаги состояния используются командами целочисленной арифметики трёх типов. При переполнении, индикатором является:<br>1. для знаковой арифметики - флаг **OF**,<br>2. для беззнаковой арифметики - флаг **CF**,<br>3. для BCD-арифметики - флаг **AF** |
| **IOPL** | Уровень Привилегий ввода-вывода (Input/Output Privilege Level) | 12, 13 | Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода в зависимости от привилегированности задачи |
| **NT** | Флаг вложенной задачи (Nested Task flag) | 14 | 1 - текущая задача является вызванной из предыдущей;<br>0 - текущая задача либо ***НЕ*** является вызванной из предыдущей |
| **RF** | Флаг возобновления (Resume Flag) | 16 | Управляет ответом процессора на исключение отладки. |
| **VM** | Флаг режима виртуального 8086 (Virtual-8086 Mode flag) | 17 | 1 - процессор переходит в режим виртуального 8086;<br>0 - возвращается в защищённый режим |
| **AC** | Флаг проверки выравнивания (Alignment Check flag) | 18 | 1- заставляет процессор проверять выравнивание при доступе к памяти и в случае невыравненного доступа генерировать исключение |
| **VIF** | Флаг виртуальных прерываний (Virtual Interrupt) | 19 | Это виртуальный образ флага IF, используется совместно с флагом VIP при включённом расширении режима виртуального 8086 |
| **VIP** | Флаг ожидания виртуального прерывания (Virtual Interrupt Pending flag) | 20 | Устанавливается, когда возникает прерывание. Процессором только считывается и используется совместно с флагом VIF; изменяется только программно |
| **ID** | Флаг идентификации (IDentification flag) | 21 | Если программа смогла установить и сбросить этот флаг, то это значит, что процессор может выполнить команду CPUID |

Десятичное значение 42936

Преобразование десятичного формата в шестнадцатеричный

---

| Частное | Остаток | Шест. | | |
|---|---|---|---|---|
| 42936 / 16 | 2683 | 8 | 8 | (младшая цифра) |
| 2683 / 16 | 167 | 11 | B | |
| 167 / 16 | 10 | 7 | 7 | |
| 10 / 16 | 0 | 10 | A | (старшая цифра) |

Преобразование шестнадцатеричного формата в десятичный

---

| | |
|---|---|
| Первая цифра: A (10) | 10 |
| Умножить на 16 | *16 |
| | 160 |
| Прибавить следующую цифру, 7 | +7 |
| | 167 |
| Умножить на 16 | *16 |
| | 2672 |
| Прибавить следующую цифру, B (11) | +11 |
| | 2683 |
| Умножить на 16 | *16 |
| | 42928 |
| Прибавить следующую цифру, 8 | +8 |
| Десятичное значение | 42936 |

## 1.2 Особенности выполнения команд

Шина МП 8086 состоит из трех шин: информационной (16 бит), адресной (20 бит) и управляющей, по которой передаются сигналы управления. Выполнение программы заключается в повторении 5-ти этапов. Эти этапы выполняются последовательно.

1. выбор служебной машинной команды из памяти;
2. расшифровка команды;
3. чтение операндов из памяти (если необходимо);
4. выполнение команды;
5. запись операндов в память (если необходимо).

Для ускорения выполнения команд они реализуются двумя устройствами: интерфейсом шины (Bus Interface Unit) и операционным блоком (Execution Unit) (рис. 1.2.1.). Первое устройство считывает команду и осуществляет передачу данных. Второй - лишь выполняет команды. Интерфейс шины может выбирать следующую команду в то время, как операционное устройство выполняет раньше выбранную.

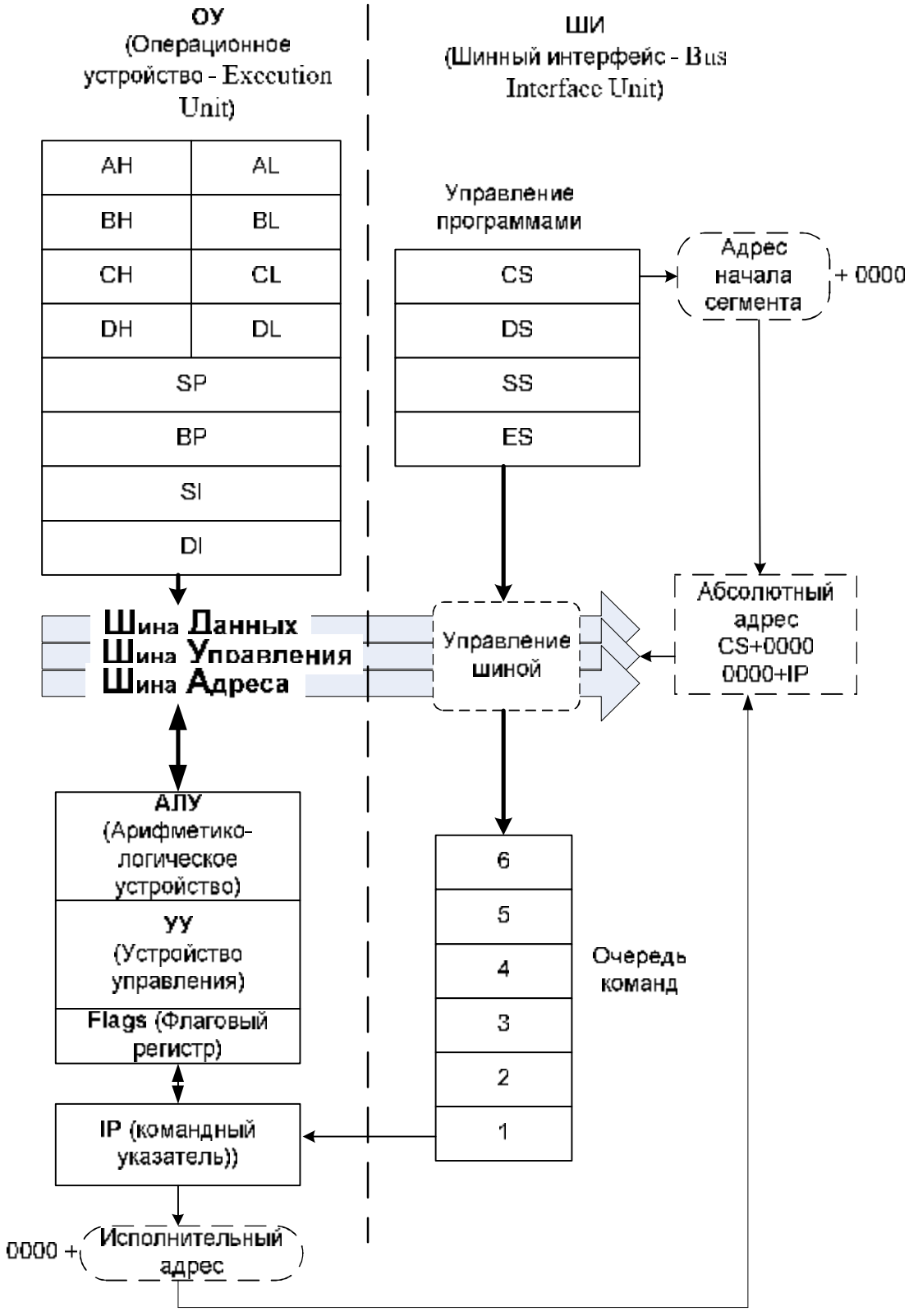



Рис. 1.2.1. Операционное устройство и шинный интерфейс.

Процессор имеет внутреннюю память, которая называется *очередью команд*. Здесь сохраняется до 4 (в 8086 - до 6) предварительно выбранных из потока команд-байтов. То есть, реализуется своеобразный конвеер команд. Как же формируется абсолютный адрес команды?

В регистре указателя команд IP записан исполнительный адрес, а в регистре сегмента кода CS - адрес начала сегмента. Абсолютный адрес равняется сумме CS+IP и записывается в 20 бит, в то время как и CS и IP имеют по 16.

Принято, что сегмент должен начинаться не с любого адреса, а кратного 16 бит. Область памяти 16 бит называется *параграфом*. Иначе говоря, сегмент выравнивается по границе параграфа. Следовательно, сегмент может начинаться лишь с адресов 16, 32, 48 . В двоичной системе это будет:

```
00000
1000000
1100000
```

Как видим, при этом последние 4 биты будут нулевыми. Для хранения они лишние и их отбрасывают. То есть, в 16-ти разрядных регистрах сегментов фактически сохраняется 20-ти разрядный адрес, но без 4 нулей справа.

При вычислении абсолютного адреса к содержанию регистра CS дописывают 4 нуля справа, а к содержанию IP - 4 нуля налево и полученные коды добавляют.

### *1.2.1. Реализация прерываний*

Считывание символа и запись к памяти длится несколько микросекунд. Если ЦП будет лишь принимать эти символы, то большинство времени он будет простаивать. Потому, закончив обработку символа, ЦП переходит к выполнению другой программы. Каждый раз, как нажимается клавиша, устройство подает запрос на прерывание. ЦП перерывает выполнение программы и переходит к выполнению процедуры обработки для прерываний для клавиатуры. Каждая такая процедура является определенной программой, записанной в памяти.

Для того, чтобы к ней перейти, нужно знать ее начальный адрес. Этот начальный адрес и записан в так называемом *векторе прерывания*. Каждое из возможных 256 прерываний имеет свой вектор прерывания в памяти. Вектор прерываний состоит из двух слов: **CS:IP**. Записанные в начальных адресах от 0 к 03FFH (1024 байта).

**Особенности 32-разрядных процессоров**

В процессоре i80286 адресная шина состояла из 24 битов, что дало возможность адресовать ло 16М памяти. Но это возможно сделать в т. зв. *защищенном режиме*.

В процессоре i486 32-разрядное слово, которое дает возможность адресовать $2^{32} = 4*^{230} = 4$ Гбайт. При сегментной организации памяти размер сегмента и будет таким. Кроме того, есть еще и страничная организация памяти. Размер страницы - 4 Кб. Такой способ позволяет использовать виртуальную память, объем которой больше физической, около 4 Тбайт.

Эти процессоры должны возможность реализовать многозначительные вычисления. Потому их структура сложнее.

Для программиста процессор состоит из 32 регистров, 16 из которых является системными, а остальные - пользователя.

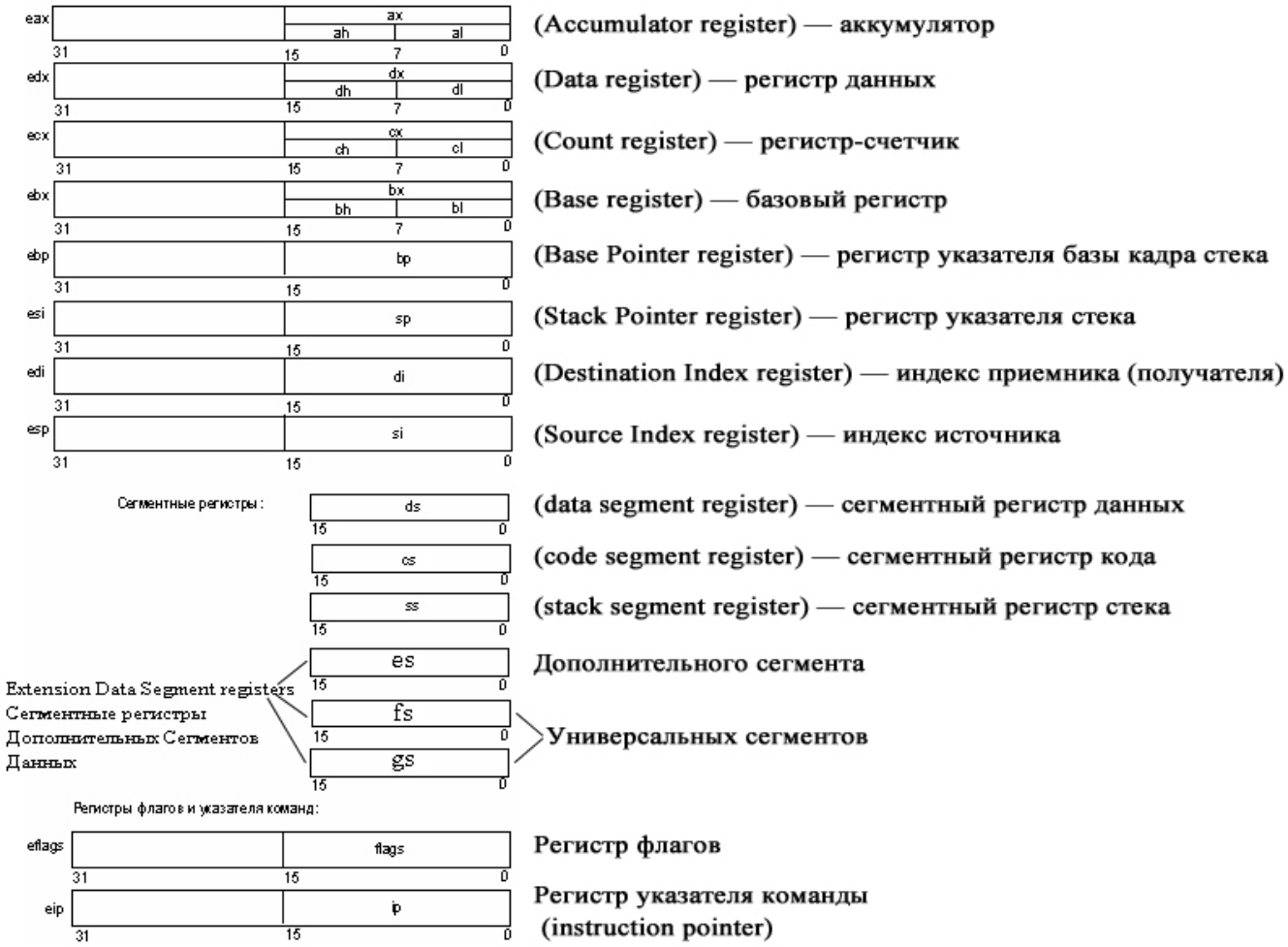
Рассмотрим особенности регистров пользователя.

Регистры общего назначения: 32-разрядные. Имя 32-разрядного начинается буквой E (extended): **EAX, EBX, ECX, EDX**. Младшая половина просто **AX** . Младшая половина доступна и может делиться пополам. Старшая половина отдельно недоступна.

Регистры-указатели - такие же имена: **ESP, EBP, ESI, EDI**. Младшая половина - SP.

Сегментные регистры - 16-разрядны: **CS, SS, DS, ES** и есть еще 2 дополнительных -- **GS, FS**. То есть, 6 регистров.

Регистры управления: 32-разрядные -- **EFLAGS** и **EIP**, а младшая часть **- FLAGS** и **IP**.



# Лекция 3

## *1.2.2. Контроллер прерываний*

### *1.2.2.1. Механизм прерываний*

Для обработки событий, происходящих асинхронно по отношению к выполнению программы, лучше всего подходит механизм прерываний. Прерывание можно рассматривать как некоторое особое событие в системе, требующее моментальной реакции. Например, хорошо спроектированные системы повышенной надежности используют прерывание по аварии в питающей сети для выполнения процедур записи содержимого регистров и оперативной памяти на магнитный носитель, с тем чтобы после восстановления питания можно было продолжить работу с того же места.

Кажется очевидным, что возможны самые разнообразные прерывания по самым различным причинам. Поэтому прерывание рассматривается не просто как таковое, с ним связывают число, называемое номером типа прерывания или просто номером прерывания. С каждым номером прерывания связывается то или иное событие. Система умеет распознавать, какое прерывание, с каким номером произошло и запускает соответствующую этому номеру процедуру.

Программы могут сами вызывать прерывания с заданным номером. Для этого они используют команду ***INT***. Это так называемые программные прерывания. Программные прерывания не являются асинхронными, так как вызываются из программы (а она-то знает, когда она вызывает прерывание!).

Программные прерывания удобно использовать для организации доступа к отдельным, общим для всех программ модулям. Например, программные модули операционной системы доступны прикладным программам именно через прерывания, и нет необходимости при вызове этих модулей знать их текущий адрес в памяти. Прикладные программы могут сами устанавливать свои обработчики прерываний для их последующего использования другими программами. Для этого встраиваемые обработчики прерываний должны быть резидентными в памяти.

Аппаратные прерывания вызываются физическими устройствами и приходят асинхронно. Эти прерывания информируют систему о событиях, связанных с работой устройств, например о том, что наконец-то завершилась печать символа на принтере и неплохо было бы выдать следующий символ, или о том, что требуемый сектор диска уже прочитан его содержимое доступно программе.

Использование прерываний при работе с медленными внешними устройствами позволяют совместить ввод/вывод с обработкой данных в центральном процессоре и в результате повышает общую производительность системы.

Некоторые прерывания (первые пять в порядке номеров) зарезервированы для использования самим центральным процессором на случай каких-либо особых событий вроде попытки деления на ноль, переполнения и т.п.

Иногда желательно сделать систему нечувствительной ко всем или отдельным прерываниям. Для этого используют так называемое маскирование прерываний, о котором мы еще будем подробно говорить. Но некоторые прерывания замаскировать нельзя, это немаскируемые прерывания.

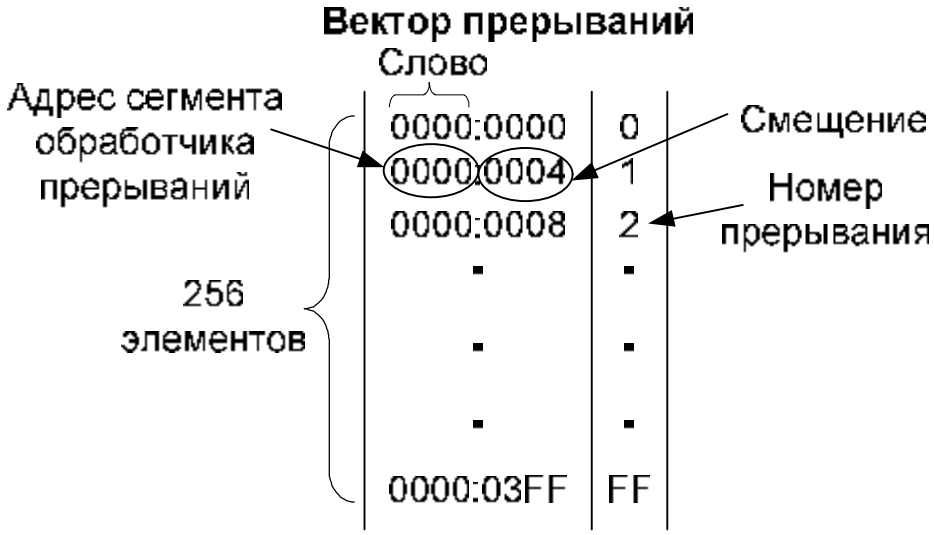
Заметим еще, что обработчики прерываний могут сами вызывать программные прерывания, например, для получения доступа к сервису BIOS или DOS (сервис BIOS также доступен через механизм программных прерываний).

### *1.2.2.2. Таблица векторов прерываний*

Для того чтобы связать адрес обработчика прерывания с номером прерывания, используется таблица векторов прерываний, занимающая первый килобайт оперативной памяти - адреса от 0000:0000 до 0000:03FF. Таблица состоит из 256 элементов - FAR-

адресов обработчиков прерываний. Эти элементы называются векторами прерываний. В первом слове элемента таблицы записано смещение, а во втором - адрес сегмента обработчика прерывания.

Прерыванию с номером 0 соответствует адрес 0000:0000, прерыванию с номером 1 - 0000:0004 и т.д.




Инициализация таблицы происходит частично BIOS после тестирования аппаратуры и перед началом загрузки операционной системой, частично при загрузке DOS. DOS может переключить на себя некоторые прерывания BIOS.

Приведем назначение некоторых наиболее важных векторов:

Таблица

Назначение некоторых наиболее важных векторов

| Номер | Описание |
|---|---|
| 0 | **Ошибка деления.** Вызывается автоматически после выполнения команд DIV или IDIV, если в результате деления происходит переполнение (например, при делении на 0). DOS обычно при обработке этого прерывания выводит сообщение об ошибке и останавливает выполнение программы. Для процессора 8086 при этом адрес возврата указывает на следующую после команды деления команду, а в процессоре 80286 - на первый байт команды, вызвавшей прерывание |
| 1 | **Прерывание пошагового режима.** Вырабатывается после выполнения каждой машинной команды, если в слове флагов установлен бит пошаговой трассировки TF. Используется для отладки программ. Это прерывание не вырабатывается после выполнения команды MOV в сегментные регистры или после загрузки сегментных регистров командой POP |
| 2 | **Аппаратное немаскируемое прерывание.** Это прерывание может использоваться по-разному в разных машинах. Обычно вырабатывается при ошибке четности в оперативной памяти и при запросе прерывания от сопроцессора |
| 3 | **Прерывание для трассировки.** Это прерывание генерируется при выполнении однобайтовой машинной команды с кодом CCh и обычно используется отладчиками для установки точки прерывания |
| 4 | **Переполнение.** Генерируется машинной командой INTO, если установлен флаг OF. Если флаг не установлен, то команда INTO выполняется как NOP. Это прерывание используется для обработки ошибок при выполнении арифметических операций |
| 5 | **Печать копии экрана.** Генерируется при нажатии на клавиатуре клавиши PrtScr. Обычно используется для печати образа экрана. Для процессора 80286 |

| | |
|---|---|
| | генерируется при выполнении машинной команды BOUND, если проверяемое значение вышло за пределы заданного диапазона |
| 6 | Неопределенный код операции или длина команды больше 10 байт (для процессора 80286) |
| 7 | Особый случай отсутствия математического сопроцессора (процессор 80286) |
| 8 | ***IRQ0*** - прерывание интервального **таймера**, возникает 18,2 раза в секунду |
| 9 | ***IRQ1*** - прерывание от **клавиатуры**. Генерируется при нажатии и при отжатии клавиши. Используется для чтения данных от клавиатуры |
| A | ***IRQ2*** - используется для каскадирования аппаратных прерываний в машинах класса AT |
| B | ***IRQ3*** - прерывание асинхронного порта **COM2** |
| C | ***IRQ4*** - прерывание асинхронного порта **COM1** |
| D | ***IRQ5*** - прерывание от контроллера жесткого диска для XT |
| E | ***IRQ6*** - прерывание генерируется контроллером флоппи-диска после завершения операции |
| F | ***IRQ7*** - прерывание **принтера**. Генерируется принтером, когда он готов к выполнению очередной операции. Многие адаптеры принтера не используют это прерывание |
| 10 | Обслуживание **видеоадаптера** |
| 11 | **Определение конфигурации** устройств в системе |
| 12 | **Определение размера оперативной памяти** в системе |
| 13 | Обслуживание **дисковой** системы |
| 14 | Последовательный **ввод/вывод** |
| 15 | Расширенный сервис для AT-компьютеров |
| 16 | Обслуживание клавиатуры |
| 17 | Обслуживание принтера |
| 18 | Запуск BASIC в ПЗУ, если он есть |
| 19 | Загрузка операционной системы |
| 1A | Обслуживание часов |
| 1B | Обработчик прерывания Ctrl-Break |
| 1C | Прерывание возникает 18.2 раза в секунду, вызывается программно обработчиком прерывания таймера |
| 1D | **Адрес видео таблицы** для контроллера видеоадаптера 6845 |
| 1E | Указатель на таблицу параметров дискеты |
| 1F | Указатель на графическую таблицу для символов с кодами ASCII 128-255 |
| 20-5F | Используется DOS или зарезервировано для DOS |
| 60-67 | Прерывания, зарезервированные для пользователя |
| 68-6F | Не используются |
| 70 | ***IRQ8*** - прерывание от часов реального времени |
| 71 | ***IRQ9*** - прерывание от контроллера EGA |
| 72 | ***IRQ10*** - зарезервировано |
| 73 | ***IRQ11*** - зарезервировано |
| 74 | ***IRQ12*** - зарезервировано |
| 75 | ***IRQ13*** - прерывание от математического сопроцессора |
| 76 | ***IRQ14*** - прерывание от контроллера жесткого диска |
| 77 | ***IRQ15*** – зарезервировано |
| 78 - 7F | Не используются |
| 80-85 | Зарезервированы для BASIC |
| 86-F0 | Используются интерпретатором BASI |
| F1-FF | Не используются |

IRQ0-IRQ15 - это аппаратные прерывания, о них будет рассказано позже.

### *1.2.2.3.. Маскирование прерываний*

Часто при выполнении критических участков программ для того, чтобы гарантировать выполнение определенной последовательности команд целиком приходится запрещать прерывания. Это можно сделать командой ***CLI***. Ее нужно поместить в начало критической последовательности команд, а в конце расположить команду ***STI***, разрешающую процессору воспринимать прерывания. Команда **CLI** запрещает только маскируемые прерывания, немаскируемые всегда обрабатываются процессором.

### *1.2.2.4. Изменение таблицы векторов прерываний*

Вашей программе может потребоваться организовать обработку некоторых прерываний. Для этого программа должна переназначить вектор на свой обработчик. Это можно сделать, изменив содержимое соответствующего элемента таблицы векторов прерываний.

Очень важно не забыть перед завершением работы восстановить содержимое измененных векторов в таблице прерываний, т.к. после завершения работы программы память, которая была ей распределена, считается свободной и может быть использована для загрузки другой программы. Если вы забыли восстановить вектор и пришло прерывание, то система может разрушиться - вектор теперь указывает на область, которая может содержать что угодно.

Поэтому последовательность действий для нерезидентных программ, желающих обрабатывать прерывания, должна быть такой:

1. Прочитать содержимое элемента таблицы векторов прерываний для вектора с нужным вам номером;
2. Запомнить это содержимое (адрес старого обработчика прерывания) в области данных программы;
3. Установить новый адрес в таблице векторов прерываний так, чтобы он соответствовал началу Вашей программы обработки прерывания;
4. Перед завершением работы программы прочитать из области данных адрес старого обработчика прерывания и записать его в таблицу векторов прерываний.

Для облегчения работы по замене прерывания DOS предоставляет в ваше распоряжение специальные функции для чтения элемента таблицы векторов прерывания и для записи в нее нового адреса. Если вы будете использовать эти функции, DOS гарантирует, что операция по замене вектора будет выполнена правильно. вам не надо заботиться о непрерывности процесса замены вектора прерывания.

Для чтения вектора используйте функцию 35h прерывания 21h. Перед ее вызовом регистр **AL** должен содержать номер вектора в таблице. После выполнения функции в регистрах **ES:BX** будет искомый адрес обработчика прерывания.

Функция 25h прерывания 21h устанавливает для вектора с номером, который находится в **AL**, обработчик прерывания **DS:DX**.

Разумеется, вы можете непосредственно обращаться к таблице векторов прерываний, но тогда при записи необходимо замаскировать прерывания командой **CLI**, не забыв разрешить их после записи командой **STI**.

### *1.2.2.5. Особенности обработки аппаратных прерываний.*

Аппаратные прерывания вырабатываются устройствами компьютера, когда возникает необходимость их обслуживания. Например, по прерыванию таймера соответствующий обработчик прерывания увеличивает содержимое ячеек памяти, используемых для хранения времени. В отличие от программных прерываний,

вызываемых запланировано самой прикладной программой, аппаратные прерывания всегда происходят асинхронно по отношению к выполняющимся программам. Кроме того, может возникнуть одновременно сразу несколько прерываний!

Для того, чтобы система "не растерялась", решая какое прерывание обслуживать в первую очередь, существует специальная схема приоритетов. Каждому прерыванию назначается свой уникальный приоритет. Если происходит одновременно несколько прерываний, то система отдает предпочтение самому высокоприоритетному, откладывая на время обработку остальных прерываний.

Таблица

Аппаратные прерывания, расположенные в порядке приоритета

| Номер | Описание |
|---|---|
| 8 | IRQ0 прерывание интервального таймера, возникает 18,2 раза в секунду. |
| 9 | IRQ1 прерывание от клавиатуры. Генерируется при нажатии и при отжатии клавиши. Используется для чтения данных с клавиатуры. |
| A | IRQ2 используется для каскадирования аппаратных прерываний в машинах класса AT. |
| 70 | IRQ8 прерывание от часов реального времени. |
| 71 | IRQ9 прерывание от контроллера EGA. |
| 72 | IRQ10 зарезервировано. |
| 73 | IRQ11 зарезервировано. |
| 74 | IRQ12 зарезервировано. |
| 75 | IRQ13 прерывание от математического сопроцессора. |
| 76 | IRQ14 прерывание от контроллера жесткого диска. |
| 77 | IRQ15 зарезервировано. |
| B | IRQ3 прерывание асинхронного порта COM2. |
| C | IRQ4 прерывание асинхронного порта COM1. |
| D | IRQ5 прерывание от контроллера жесткого диска для XT. |
| E | IRQ6 прерывание генерируется контроллером флоппи диска после завершения операции |
| F | IRQ7 прерывание принтера. Генерируется принтером, когда он готов к выполнению очередной операции. Многие адаптеры принтера не используют это прерывание. |

Из таблицы видно, что самый высокий приоритет у прерываний от интервального таймера, затем идет прерывание от клавиатуры.

Для управления схемами приоритетов необходимо знать внутреннее устройство контроллера прерываний 8259. Поступающие прерывания запоминаются в регистре запроса на прерывание ***IRR***. Каждый бит из восьми в этом регистре соответствует прерыванию. После проверки на обработку в настоящий момент другого прерывания запрашивается информация из регистра обслуживания ***ISR***. Перед выдачей запроса на прерывание в процессор проверяется содержимое восьмибитового регистра маски прерываний ***IMR***. Если прерывание данного уровня не замаскировано, то выдается запрос на прерывание.

### 1.3 Представление данных в ЭВМ

Данные, обрабатываемые МП 8086, можно разделить на три вида:

1. физические и логические коды;
2. числа с фиксированной точкой;
3. числа с плавающей точкой.

С точки зрения размерности (***физическая интерпретация***) микропроцессор аппаратно поддерживает следующие основные типы данных.

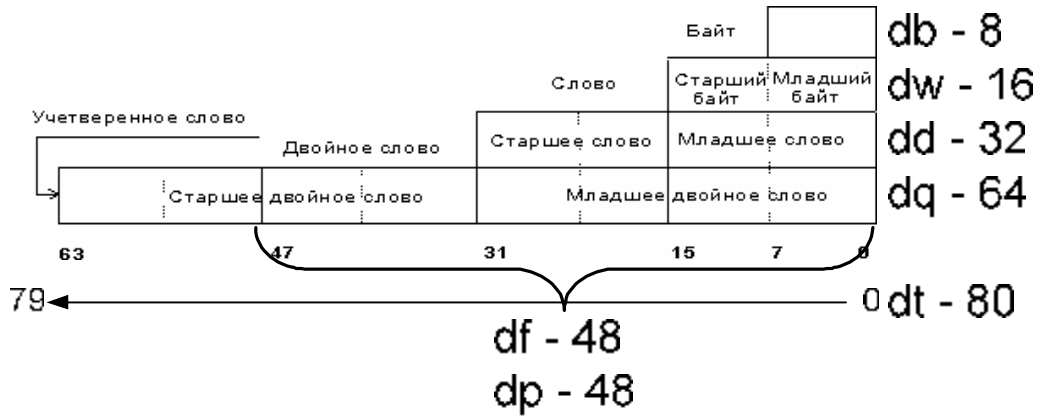
### *1.3.1. Типы данных*

**байт** — восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом;

**слово** — последовательность из двух байт, имеющих последовательные адреса. Размер слова — 16 бит; биты в слове нумеруются от 0 до 15.

**двойное слово** — последовательность из четырех байт (32 бита), расположенных по последовательным адресам. Нумерация этих бит производится от 0 до 31. Адресом двойного слова считается адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.

**учетверенное слово** — последовательность из восьми байт (64 бита), расположенных по последовательным адресам. Адресом учетверенного слова считается адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.




С точки зрения их разрядности, микропроцессор на уровне команд поддерживает **логическую** интерпретацию:

### *1.3.2. Логические коды*

Логическими кодами представляются: символы, числа без знака, битовые величины.

Все устройства ввода/вывода обмениваются данными из ЭВМ символами. Любой текст или число при этом рассматриваются как последовательность символов.

Каждый символ кодируется кодом ACSII (КОИ -- 7) и в памяти ЭВМ занимает один байт. Основная часть символов (латинские буквы, цифры, знаки препинаний) - 128 символов кодируется 7 битами, 8-й всегда равняется нулю. Остальные символы (кирилличные буквы и проч.) кодируются расширенной частью кода. (8-ю битами). Например, большая латинская буква А имеет код $65_{10}$, буква С - $67_{10}$. Потому АС будет иметь вид:

| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| | | | С | | | | | | | | А | | | | |

Цифра 0 имеет код $48_{10}$ = 30h:

| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

Адрес ячейки памяти или байта является числом без знака, которое подается так же, как и числом с фиксированной точкой, только без знака.

При работе с внешними устройствами, превращении данных придется иметь дело с отдельными битами слова. В этом случае содержание ячейки рассматривается как определен код, или битовая величина.

### *1.3.3. Числа с фиксированной точкой*

***Целый тип со знаком*** — двоичное значение со знаком, размером 8, 16 или 32 бита. Знак в этом двоичном числе содержится в 7, 15 или 31-м бите соответственно. Ноль в этих битах в операндах соответствует положительному числу, а единица — отрицательному. Отрицательные числа представляются в дополнительном коде. Числовые диапазоны для этого типа данных следующие:

8-разрядное целое — от –128 до +127;

16-разрядное целое — от –32 768 до +32 767;

32-разрядное целое — от $-2^{31}$ до $+2^{31}-1$.

***Дополнительный код*** отрицательного числа формируется таким образом:

1. представить в соответствующей системе абсолютную величину числа N;
2. для каждого разряда взять его дополнение (если в двоичной системе, то просто заменить 1 на 0, и наоборот;);
3. прибавить единицу, пренебрегая при этом возможным перенесение из старшего разряда.

Например, число $-1607_{10}$ будет иметь такой вид: $-1607_{10}$

1. |N| = 0000011001000111;
2. $\overline{N}$ = 1111100110111000;
3. $\widetilde{N}$ = 1111100110111001.

***Целый тип без знака*** — двоичное значение без знака, размером 8, 16 или 32 бита. Числовой диапазон для этого типа следующий:

байт — от 0 до 255;

слово — от 0 до 65 535;

двойное слово — от 0 до $2^{32}-1$.

***Цепочка*** (***байтовая строка***)— представляющая собой некоторый непрерывный набор байтов, слов или двойных слов максимальной длины до 4 Гбайт.

***Битовое поле*** представляет собой непрерывную последовательность бит, в которой каждый бит является независимым и может рассматриваться как отдельная переменная. Битовое поле может начинаться с любого бита любого байта и содержать до 32 бит.

***Неупакованный двоично-десятичный*** тип — байтовое представление десятичной цифры от 0 до 9. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте. Значение цифры определяется младшим полубайтом.

***Упакованный двоично-десятичный*** тип представляет собой упакованное представление двух десятичных цифр от 0 до 9 в одном байте. Каждая цифра хранится в своем полубайте. Цифра в старшем полубайте (биты 4–7) является старшей.

***Указатель на память двух типов:***

*ближнего типа* — 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента. Эти указатели могут также использоваться в сплошной (плоской) модели памяти, где сегментные составляющие одинаковы;

*дальнего типа* — 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части — селектора, и 32-разрядного смещения.


В памяти ЭВМ целое число представляются в двоичной системе исчисления.

Для удобства реализации арифметических операций целые числа внутри ЭВМ представляются в *дополнительном двоичном коде*.

Положительные числа имеют обычное двоичное представление и могут занимать одно слово или байт. Потому максимальным обычным числом будет $2^{15}$-1=$32767_{10}$. В байт можно записать не больше, чем $2^7 - 1 = 127_{10}$.

Ясно, что в старшем разряде отрицательного числа всегда будет 1, а в положительных - 0. Поэтому этот разряд называется знаковым.

В дополнительном двоичном коде +0 и -0 представляются одинаково.

Обычные целые числа в памяти ЭВМ могут изменяться в диапазоне $-32768_{10} \le N \le 32767_{10}$

Использование дополнительного двоичного кода позволяет заменить операцию вычитания операцией добавления.

Кроме отмеченного представления целых чисел используется еще специальное представление десятичных чисел для двоично-десятичной арифметики. При этом одна десятичная цифра записывается в 8 (НЕупакованная форма) или 4 бита (упакованная форма) в двоичной форме.

### *1.3.4. Числа с плавающей точкой*

В МП 8086 непосредственно не предусматривается действий с плавающей точкой.

Обычные десятичные числа можно записывать как число с порядком, например:

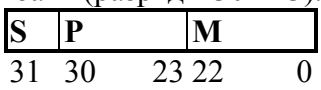
$1234.5 = 123.45*10 = 12.345*10^2 = 1.2345*10^3$

Такая форма выходит неоднозначной. Однако, если число изображать так, чтобы оно было меньше единицы и первая цифра за десятичной точкой была значимой (то есть не нуль), то такое представление будет однозначно и будет называться *нормализованным*. Нормализованное число в памяти подается *мантиссой и порядком*.

Для МП 8086 нормализованным в двоичной системе считается число, целая часть мантиссы которого равняется единице.

Такая форма удобна для компактного представления очень малых или очень больших чисел. Если же число имеет много разрядов, то сохранить их все не удается. Поэтому числа с плавающей точкой представляются в ЭВМ приближенно, а с фиксированной - точно.

В ЭВМ обычное вещественное число представляются в форме с плавающей точкой и занимает два слова. Старший бит первого слова является знаковым (S). Порядок (P) занимает байт (разряды 30 - 23). Мантисса (M) расположена в 23 битах.




Отметим особенности представления порядка и мантиссы числа. Чтобы не отводить один разряд под знак порядка, в эти разряды записывается не настоящий порядок, а со сдвигом $+127_{10}$.

Поскольку мантисса числа записывается в нормализованном виде, то в двоичном представлении целая ее часть будет всегда равняться единице. Эту единицу в памяти НЕ записывают (сдвигают мантиссу на один разряд влево).

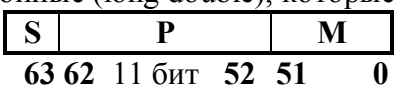
Например:

Число $15.375_{10} = 1111.011_2$.
В нормализованной форме оно будет иметь вид:
$1.111011*2^{11}{}_2$
Внутреннее представление числа:
$S = 0$; $P = 3 + 127 = 130_{10} = 10000010_2$; $M = 1110110\ldots0$.

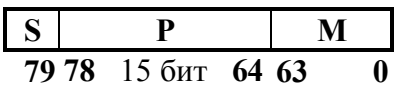
В отличие от чисел с фиксированной точкой отрицательные вещественные числа отличаются от положительных лишь знаком.

Порядок числа определяет его диапазон, а мантисса - точность (количество значимых разрядов). Кроме обычных чисел можно использовать двойные (double) и длинные двойные (long double), которые имеют структуру:

double




long double




В длинных двойных мантисса записывается полностью, а в двойных первая единица мантиссы сохраняется неявно.

Свойства чисел с плавающей точкой:

| Тип | Размер (бит) | Диапазон | | Точность десятичных розрядов |
|---|---|---|---|---|
| | | Минимальный | Максимальный | |
| Float | 32 | $3.4*10^{-38}$ | $3.4*10^{38}$ | 7 |
| Double | 64 | $1.7*10^{-308}$ | $1.7*10^{308}$ | 15 |
| Long double | 80 | $3.4*10^{-4932}$ | $3.4*10^{4932}$ | 19 |

В самом МП 8086 действий над числами с плавающей точкой не предусмотрено. Они выполняются в сопроцессоре 8087, 8187, 8287, или моделируются программно (эмулируются).

# Лекция 4

### *1.3.5. Десятичные числа*

Десятичные числа — специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех бит. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом двоично-десятичном коде (BCD — Binary-Coded Decimal). Микропроцессор хранит BCD-числа в двух форматах (рис. Рис. 1.3.1):

*упакованном формате* — в этом формате каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером 4 бита. При этом код старшей цифры числа занимает старшие 4 бита. Следовательно, диапазон представления десятичного упакованного числа в одном байте составляет от 00 до 99;

*неупакованном формате* — в этом формате каждый байт содержит одну десятичную цифру в четырех младших битах. Старшие четыре бита имеют нулевое значение. Это так называемая зона. Следовательно, диапазон представления десятичного неупакованного числа в одном байте составляет от 0 до 9.

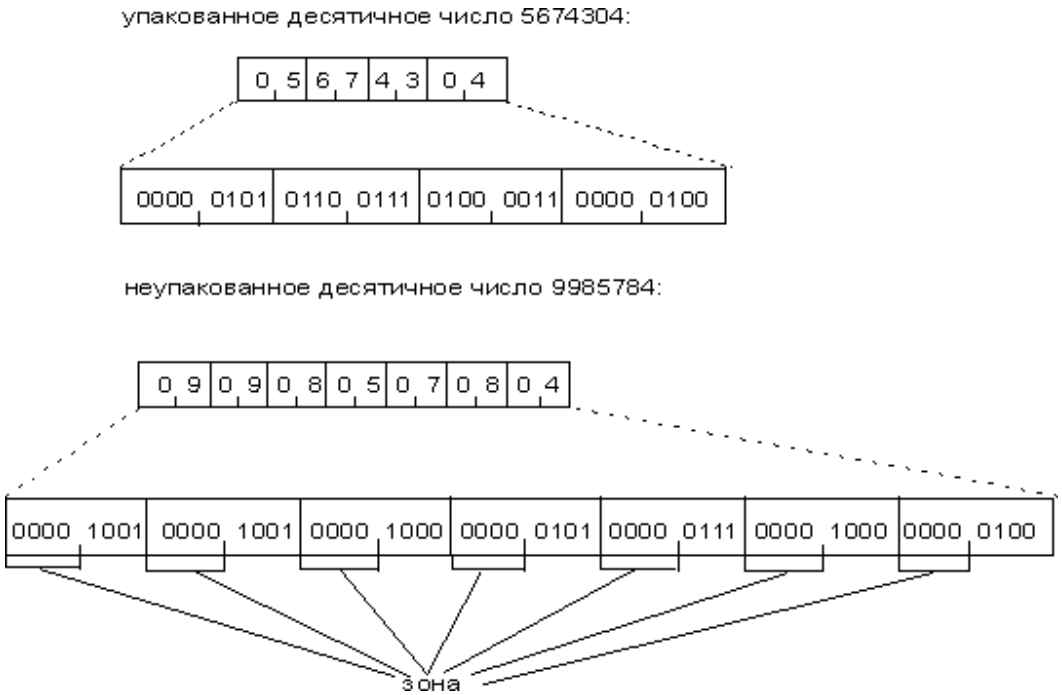



Рис. 1.3.1. Представление числовой информации в двоично-десятичном коде (BCD — Binary-Coded Decimal)

Для этого можно использовать только две директивы описания и инициализации данных — **db** и **dt**. Возможность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип “младший байт по младшему адресу”, что, как мы увидим далее, очень удобно для их обработки. И вообще, при использовании такого типа данных как BCD-числа, порядок описания этих чисел в программе и алгоритм их обработки — это дело вкуса и личных пристрастий программиста. Это станет ясно после того, как мы ниже рассмотрим основы работы с BCD-числами. К

примеру, приведенная в сегменте данных листинга последовательность описаний BCD-чисел будет выглядеть в памяти так, как показано на рис. 1.3.2..

```
DSEG segment para PUBLIC "DATA"
SOURCE db 1,5,9,8
DESTpak Dt 232456791
DEST Dd 232456791
DSEG ENDS

; СЕГМЕНТ КОДУ

CSEG SEGMENT PARA PUBLIC "CODE"
MAIN PROC FAR
ASSUME CS:CSEG,DS:DSEG, SS:STSEG

; АДРЕСА ПОВЕРНЕННЯ

PUSH DS
XOR AX,AX ;AX=0
PUSH AX

;ІНІЦІАЛІЗАЦІЯ DS
```

```
cs:0000 1E              push    ds
cs:0001 33C0            xor     ax,ax
cs:0003 50              push    ax
cs:0004 B8D35B          mov     ax,5BD3
cs:0007 8ED8            mov     ds,ax
cs:0009 B8004C          mov     ax,4C00
cs:000C CD21            int     21
cs:000E 0000            add     [bx+si],al
cs:0010 FB              sti
cs:0011 52              push    dx
cs:0012 0003            add     [bp+di],al
cs:0014 0E              push    cs
cs:0015 0000            add     [bx+si],al
cs:0017 0002            add     [bp+si],al
cs:0019 0017            add     [bx],dl

ds:0000 01 05 09 08 91 67 45 32 ☺♣○◘CgE2
ds:0008 02 00 00 00 00 00 57 02 ☻      W☻
ds:0010 DB 0D 00 00 00 00 00 00 ■♪
ds:0018 00 00 00 00 00 00 00 00
ds:0020 1E 33 C0 50 B8 D3 5B 8E ▲3└P╕╙[O
```

```
DSEG segment para PUBLIC "DATA"
SOURCE db 1,5,9,8
DESTpak Dt 232456791
DEST Dd 232456791
DSEG ENDS

; СЕГМЕНТ КОДУ

CSEG SEGMENT PARA PUBLIC "CODE"
MAIN PROC FAR
ASSUME CS:CSEG,DS:DSEG, SS:STSEG

; АДРЕСА ПОВЕРНЕННЯ

PUSH DS
XOR AX,AX ;AX=0
PUSH AX

;ІНІЦІАЛІЗАЦІЯ DS
```

```
cs:0001 33C0            xor     ax,ax
cs:0003 50              push    ax
cs:0004 B8D35B          mov     ax,5BD3
cs:0007 8ED8            mov     ds,ax
cs:0009 B8004C          mov     ax,4C00
cs:000C CD21            int     21
cs:000E 0000            add     [bx+si],al
cs:0010 FB              sti
cs:0011 52              push    dx
cs:0012 0003            add     [bp+di],al
cs:0014 0E              push    cs
cs:0015 0000            add     [bx+si],al
cs:0017 0002            add     [bp+si],al
cs:0019 0017            add     [bx],dl

ds:0000 01 05 09 08 91 67 45 32 ☺♣○◘CgE2
ds:0008 02 00 00 00 00 00 57 02 ☻      W☻
ds:0010 DB 0D 00 00 00 00 00 00 ■♪
ds:0018 00 00 00 00 00 00 00 00
ds:0020 1E 33 C0 50 B8 D3 5B 8E ▲3└P╕╙[O
```

```
DSEG segment para PUBLIC "DATA"
SOURCE db 1,5,9,8
DESTpak Dt 232456791
DEST Dd 232456791
dfg db 03
DSEG ENDS

; СЕГМЕНТ КОДУ

CSEG SEGMENT PARA PUBLIC "CODE"
MAIN PROC FAR
ASSUME CS:CSEG,DS:DSEG, SS:STSEG

; АДРЕСА ПОВЕРНЕННЯ

PUSH DS
```

```
#lab1#26: PUSH AX
  cs:0003 50              push    ax
#lab1#30: MOV AX,DSEG
  cs:0004 B8D35B          mov     ax,5BD3
#lab1#31: MOV DS,AX
  cs:0007 8ED8            mov     ds,ax
#lab1#33: mov ax,4c00h
  cs:0009 B8004C          mov     ax,4C00
#lab1#34: int 21h
  cs:000C CD21            int     21
  cs:000E 0000            add     [bx+si],al

  ds:0000 01 05 09 08 91 67 45 32 ☺♣○◘CgE2
  ds:0008 02 00 00 00 00 00 57 02 ☻      W☻
  ds:0010 DB 0D 03 00 00 00 00 00 ■♪♥
  ds:0018 00 00 00 00 00 00 00 00
  ds:0020 1E 33 C0 50 B8 D3 5B 8E ▲3└P╕╙[O
```

Рис. 1.3.2. Листинги описаний BCD-чисел

## 1.4. Режимы адресации

При выполнении программы процессор обращается к памяти, где сохраняются команды и данные. В командах превращения данных определяются адреса, где сохраняется соответствующая информация, а в командах передачи управления определяется адрес команды, на которую нужно перейти, то есть, адреса переходов.

Способ или метод определения в команде адреса операнда или адреса перехода, называется *режимом адресации*, просто ли *адресацией*.

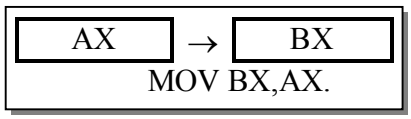
Можно в командах отмечать адреса операндов, то есть, использовать прямую адресацию. Но операнды сохраняются не только в ячейках памяти. Они могут находиться в регистрах общего назначения, сегментных регистрах. Кроме того, операндами могут быть константы, или операнды могут находиться в портах ввода/вывода.

В таких условиях использования только прямой адресации приведет к неэффективным программам. Поэтому в современных ЭВМ используют много других режимов адресации, что позволяет получать высокоэффективные программы для разных приложений.

Режимы адресации МП 8086 можно разделить на 7 групп:

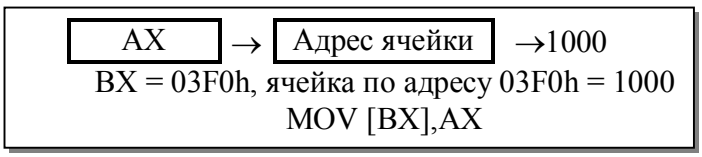
1. регистровая адресация;
2. непосредственная адресация;
3. прямая;
4. непрямая регистровая;
5. адресация за базой;
6. прямая адресация с индексированием;
7. адресация за базой с индексированием.

В последующем будем считать, что когда в команде отмечено имя регистра, то операндом будет его содержание.




Например:

Если же имя регистра заключено в прямоугольные скобки, то значат, что операндом является содержание ячейки, адрес которого сохраняется в регистре.




В первом случае - *прямая* адресация, а во втором – *непрямая(косвенная)*.

### *1.4.1. Регистровая адресация*

Операнд (байт или слово) находится в регистре. Этот способ адресации применим ко всем программно-адресуемым регистрам процессора.


В этом случае операндом является содержание определенного регистра.

16-битовое слово, которое сохраняется в счетчике CX, переписывается в аккумулятор AX. CX останется неизменным, а AX изменится.

Аналогично для байтов MOV AL, BH.

То есть, сам операнд определяет свою длину - слово или байт. Этот метод не нуждается в обращении к памяти, сама команда занимает мало места. Потому выполняется очень быстро и является очень эффективной.

```
inc CH ;Плюс 1 к содержимому CH

push DS ;DS сохраняется в стеке

xchg BX,BP ;BX и BP обмениваются содержимым

mov ES, AX ;Содержимое AX пересылается в ES
```

### *1.4.2. Непосредственная адресация*

В этом случае вместо операнда источника используется непосредственно константа:

```
MOV AX, 60
```

В аккумулятор заносится число 60. Эта константа размещается не в памяти, а в самой машинной команде, то есть, в очереди команд. Потому будет выполняться достаточно быстро. Можно:

```
MOV CL, -50
```

Константа может представляться *литералом*, а может быть и *именуемой*. Имя константе присваивается специальным оператором: EQU:

```
L EQU 256
……
MOV CX,L
```

### *1.4.3. Прямая адресация*

Как отмечалось, в указателе команд **IP** сохраняется относительный адрес команды в сегменте, то есть, количество байт относительно его начала, или *исполнительный адрес*.

Для прямой адресации исполнительный адрес отмечается непосредственно в команде. Если это адрес данных, то МП добавляет ее к содержанию регистра данных **DS**, который сдвигается на 4 бита и получает 20-битовый абсолютный адрес.

Использовать конкретные числовые значения адресов неудобно. Поэтому адрес чаще задается меткой.

Чтобы разместить какое-то число по этому адресу, используются операторы ***DB, DW*** или ***DD*** (Define Byte, Define Word, Define Double Word).

```
TABLE DW 1560;          в ячейку TABLE записано 1560
INDEX DB -126;          в байт INDEX записано -126
```

Тогда можно записать:

```
MOV AX,TABLE;        переслать содержание TABLE в аккумулятор.
```

Отметим особенность такой пересылки. Когда в памяти было записано:

| | |
|---|---|
| **X** | **TABLE** |
| **Y** | **TABLE+1** |

то младший байт X будет переслан в младший байт регистра, а старший - в старший. И получим:

| | |
|---|---|
| **Y** | **X** |
| **AH** | **AL** |

То есть, байты будто поменялись местами (смещение записывается до команды).

### *1.4.4. Непрямая регистровая адресация*

Этот способ адресации использует базовый регистр **BX**, указатель **BP** и индексные регистры SI, BI, где записан адрес операнда:

```
MOV AX,[BX]
```

Для выбора операнда-источника происходит обращение к регистру **BX**, где сохраняется его адрес. Если там записано 2000, то содержание ячейки 2000 пересылается в аккумулятор.

А как в регистр **BX** занести адрес ячейки, например, TABLE? Это можно с помощью операции **OFFSET** (смещение).

Например:

```
MOV BX, OFFSET TABLE
Сравните: MOV BX, TABLE.
```

### *1.4.5. Адресация за базой*

Если необходимо получить доступ к одной ячейке, то ей нужно предоставить имя и использовать прямую адресацию:

```
MOV AX,TABLE
```

При работе с массивом данных помечать своим именем каждое слово нецелесообразно, а достаточно запомнить адрес начала массива, например, в регистре **BX** или **BP**.

Тогда адрес любого элемента массива определится как сумма базового адреса и целой константы ***[BP] + N***, где N -- количество байт от начала массива (смещение). Если начальный адрес массива записать в регистр **BP**, то второе слово можно переслать в аккумулятор так:

```
MOV AX, [BP] + 2
```

Исполнительный адрес будет определен как сумма содержания **BP** и определенного смещения. Такой метод адресации называется *адресацией за базой*.

Выше приведенная запись имеет и другие эквивалентные формы:

```
MOV AX, 2[BP]
MOV AX, [BP + 2]
```

Ясно, что смещение может быть и отрицательным.

То есть, выбрав необходимое смещение, можно произвольно адресовать элементы массива.

### *1.4.6. Прямая адресация с индексированием*

Если зафиксировать базовый адрес элементов данных определенной меткой, тогда добраться до других элементов данных можно с помощью индексных регистров **SI, DI**.

Например:

```
MOV DI, 2
MOV AX, TABLE
```

Исполнительный адрес определится как сумма адреса TABLE и регистра **DI**. В данном случае в аккумулятор **AX** будет переслано второе слово после TABLE.

Это *прямая адресация с индексированием*. Этот метод адресации удобно использовать для регулярной обработки массивов. Если вторую команду разместить в цикле и там же добавлять 2 к **DI**, то можно обрабатывать все слова массива TABLE.

### *1.4.7. Адресация за базой с индексированием*

Для обработки двумерных массивов удобно использовать адресацию за базой с индексированием, когда исполнительный адрес равняется сумме значений базового регистра, индексного регистра и сдвига.

```
MOV AX, VALUE [BX] [DI]
```

Здесь VALUE - именуемая константа, а не адрес ячейки. Вместо имени переменной можно задавать адресную константу. Например:

```
MOV AX, 2[BP] [SI]
```

Операнды в скобках можно записывать по-разному:

```
MOV AX, [BP + 2 + SI];
MOV AX, [SI + BP + 2];
MOV AX, [BP] [SI + 2].
```

Ясно, что сдвига может не быть. Такие основные методы адресации. Однако, это далеко не все методы адресации.

# Лекция 5

## 1.5 Особенности формирования машинных команд

### *1.5.1. Формат машинной команды*

По мнемонике команды транслятор создает машинную команду. Она может занимать от 1 до 6 байт в зависимости от того, какие режимы адресации применяются. Смещение и непосредственные данные также записываются в команду. Если они в границах -128 +127, то занимают байт, иначе - слово.

Кратчайшие команды - в один байт, для тех команд, для которых операнд определяется самой командой (команды для работы с битами регистра флагов или **CLC** - очистить бит флага CF).

Самые длинные команды возникают тогда, когда в них используется 16-битовое смещение (DISP) или 16-битовые непосредственные данные (DATA).

Поэтому для разных команд и форматы команд будут разными, то есть какие коды и где они размещаются.

Рассмотрим формат двухоперандной команды. В первом байте записывается код операции, во втором - режимы адресации. Остальные байты - под смещение (DISP) или непосредственные данные (DATA).

| **КОП** | **S** | **W** |
|---|---|---|

| **MOD** | **REG** | **R/M** |
|---|---|---|

В первом байте, кроме кода операции, есть еще 2 однобитовых индикатора: **W** - определяет, над какой единицей данных выполняется команда. **W** = 1 - над словом, **W** = 0 - над байтом.

Бит **S** показывает, чем является тот регистр - операндом-источником (**S** = 0) или приемником (**S** = 1).

Регистры кодируются таким образом:

| **КОП** \ **W** | **W = 1** | **W = 0** |
|---|---|---|
| 000 | AX | AL |
| 001 | CX | CL |
| 010 | DX | DL |
| 011 | BX | BL |
| 100 | SP | AH |
| 101 | BP | CH |
| 110 | SI | DH |
| 111 | DI | BH |

Сегментные регистры кодируются так:

| | |
|---|---|
| 00 | DS |
| 01 | CS |
| 10 | SS |
| 11 | ES |

Режим адресации второго операнда определяется кодами в полях **MOD** и **R/M** (register-memory).

Поле **MOD** определяет, что именно закодировано в поле **R/M**. Если **MOD** = 00 -- смещения НЕТ; **MOD** = 01 - смещение БАЙТ, **MOD** = 10 - смещение СЛОВО. Когда же **MOD** = 11, то значат, что в поле R/M записан код второго регистра.

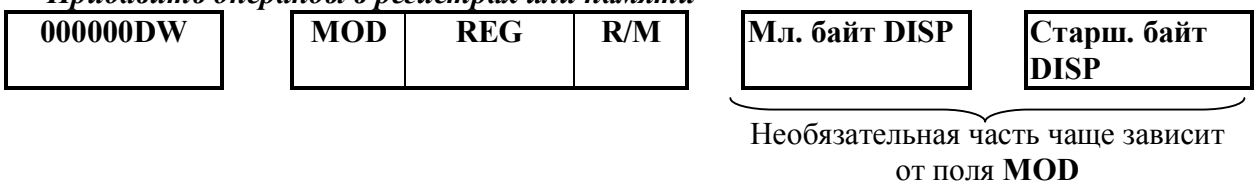
Таблица

Коды режимов адресации

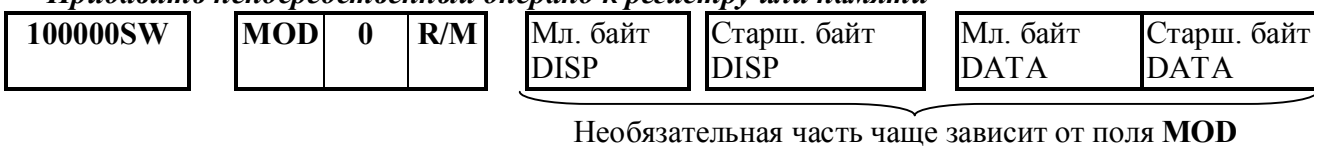
| R/M \ MOD | 00 | 01 | 10 |
|---|---|---|---|
| 000 | [BX] + [SI] | [BX + SI] + DISP 8 | [BX + SI] + DISP 16 |
| 001 | [BX] + [DI] | [BX + DI] + DISP 8 | [BX + DI] + DISP 16 |
| 010 | [BP + SI] | [BP + SI] + DISP 8 | [BP + SI] + DISP 16 |
| 011 | [BP + DI] | [BP + DI] + DISP 8 | [BP + DI] + DISP 16 |
| 100 | [SI] | [SI] + DISP 8 | [SI] + DISP 16 |
| 101 | [DI] | [DI] + DISP 8 | [DI] + DISP 16 |
| 110 | disp 16 | [BP] + DISP 8 | [BP] + DISP 16 |
| 111 | [BX] | [BX] + DISP 8 | [BX] + DISP 16 |

Особенностью МП является то, что для одной команды может быть несколько кодов и форматов в зависимости от режима адресации или операндов, которые там используются. Например, для команды **ADD** (сложить):

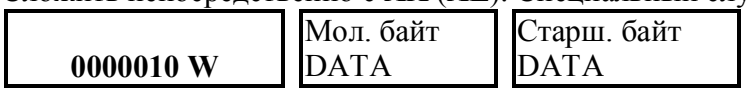
***Прибавить операнды в регистрах или памяти***




***Прибавить непосредственный операнд к регистру или памяти***



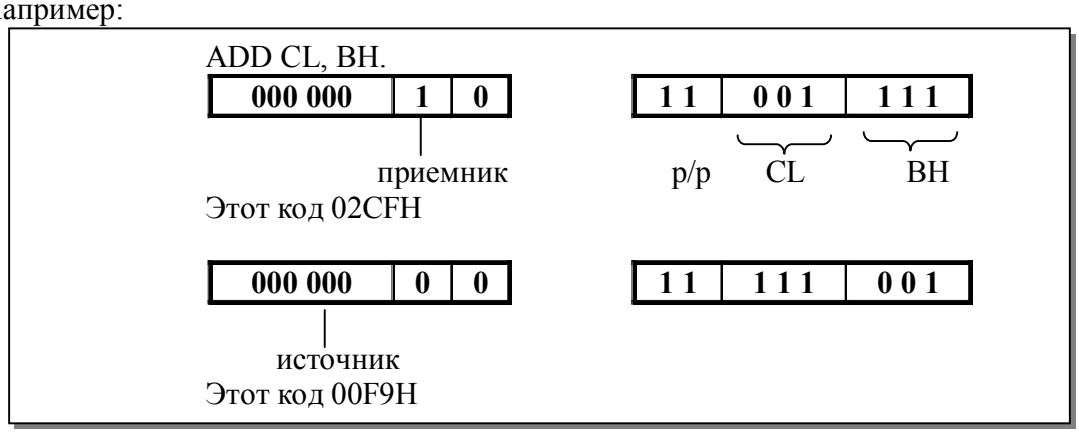

Сложить непосредственно с **AX** (**AL**). Специальный случай для аккумулятора:




То есть, команда **ADD** имеет 3 варианта кода и разные форматы. Как видно, в формате команды существует лишь одно поле для кодировки ячейки памяти. Поэтому в двухадресных командах может быть лишь один операнд - ячейка памяти, а не два. Следовательно, сложить содержание одной ячейки ко второй за одну команду нельзя.

Иногда одна и та же команда может быть закодирована транслятором по-разному.

Например:



Поля из 6 битов для кодировки всех команд недостаточно. Поэтому некоторые команды объединяются в группу и в первом байте кодируется группа команд, а команда образуется во втором байте. Например, форматы команд с непосредственной адресацией:

***регистр - непосредственный операнд***

| КОП | S | W |
|---|---|---|

| 1 1 | КОП | REG |
|---|---|---|

необязательная часть

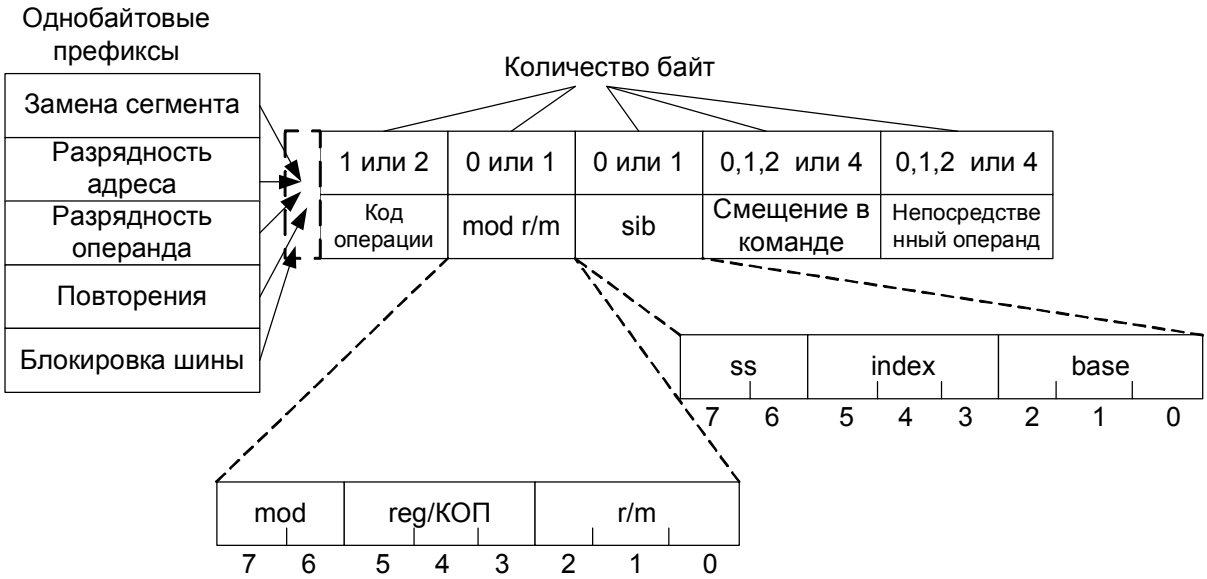
***память - непосредственный операнд***

| КОП | S | W |
|---|---|---|

| 1 1 | КОП | MEM |
|---|---|---|

Кроме отмеченных частей в команде может быть несколько префиксов, что может увеличить размер до 15 байт (рис. 1.5.1).

# Формат машинной команды




**Замена сегмента:**
2eh — замена сегмента cs;
36h — замена сегмента ss;
3eh — замена сегмента ds;
26h — замена сегмента es;
64h — замена сегмента fs;
65h — замена сегмента gs

**Разрядность адреса:**
уточняет разрядность адреса (32 или 16-разрядный)

**Разрядность операнда:**
66h — 16-разрядный;
67h — 32-разрядный

**Префикс повторения:**
- ***безусловные*** (rep — 0f3h), заставляющие повторяться цепочечную команду некоторое количество раз;
- ***условные*** (repe/repz — 0f3h, repne/repnz — 0f2h), которые при зацикливании проверяют некоторые флаги, и в результате проверки возможен досрочный выход из цикла

**Код операции (КОП)**

Обязательный элемент, описывающий операцию, выполняемую командой. Многим командам соответствует несколько кодов операций, каждый из которых определяет нюансы выполнения операции.

**Байт modr/m - режим адресации**

Значения этого байта определяет используемую форму адреса операндов:
**mod** = 00 - поле смещение в команде отсутствует
**mod** = 01 - поле смещение в команде присутствует
**mod** = 11 - операндов в памяти нет: они находятся в регистрах
**reg/КОП** - определяет либо *регистр*, находящийся в команде на месте первого операнда, либо возможное *расширение кода операции*
***r/m*** *используется совместно с полем mod и определяет либо регистр, либо базовые и индексные регистры*

**Байт масштаб-индекс-база (байт sib)**

расширяет возможности адресации операндов:
**ss** – в нем размещается масштабный множитель для индексного компонента index
**index** — используется для хранения номера индексного регистра
**base** — используется для хранения номера базового регистра

**Поле смещения в команде**

8, 16 или 32-разрядное целое число со знаком, представляющее собой, полностью или частично, значение эффективного адреса операнда

**Поле непосредственного операнда**

Необязательное поле, представляющее собой 8, 16 или 32-разрядный непосредственный операнд. Наличие этого поля, отражается на значении байта mod r/m.

Рис. 1.5.1. Формат машинной команды

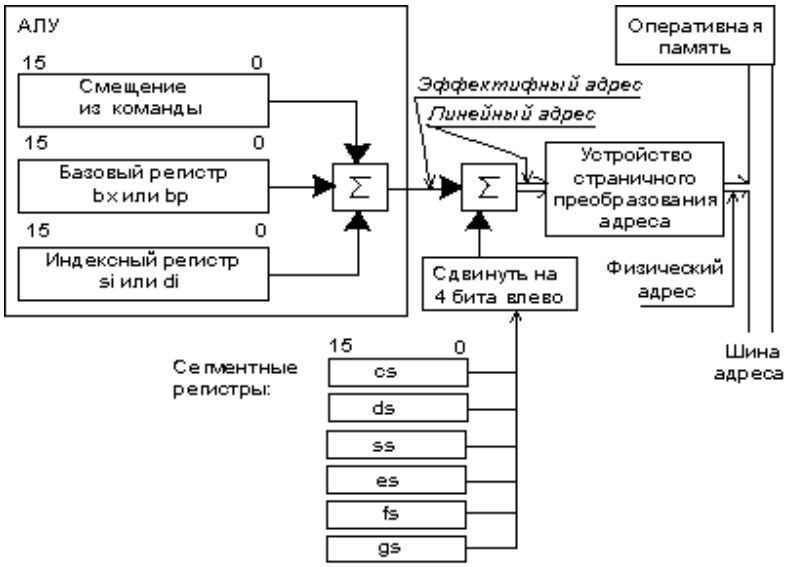



Рис. 1.5.2. Механизм формирования физического адреса в реальном времени

### *1.5.2. Операторы*

Программа на языке Ассемблер состоит из отдельных строк-операторов, которые описывают выполняемые операции. Оператором может быть *команда*, или *псевдооператор* (директива).

*Команда* - это условно (символьное) обозначение машинных команд. То есть, она отмечает ЭВМ что ей нужно выполнить. Команды после трансляции превращаются в машинные коды.

В отличие от команды, *псевдооператоры* или *директивы* имеют вспомогательный характер. Они сообщают транслятору что сделать с командами. Как правило, они не превращаются в машинные команды. Команды выполняются во время вычислений, а псевдооператоры - во время трансляции.

Каждая команда может иметь 4 поля:

**[Метка:] Мнемокод [Операнд] [, Операнд] [; Комментарий]**

Из них обязательным является лишь поле мнемокода, все другие частично или полностью могут отсутствовать.

Например:

```
COUNT: MOV AX,DI; переслать DI в аккумулятор
```

Метка присваивает имя команде. На эту метку можно передавать управление с других мест программы, то есть, на нее могут ссылаться другие команды.

Метка может складываться не больше как из 31 символа и заканчивается “:”. Сюда могут входить большие или малые латинские буквы от A к Z, или от a к z (большие и малые буквы не различаются), цифры от 0 до 9 и специальные знаки ? . @ _ $. Метка может начинаться любым символом, кроме цифры. Знаки (пропуски) препинаний нельзя включить, за исключением подчеркиваний.

*Поле мнемокода* - содержит мнемокод или мнемонику команды от 3 до 6 символов.

Операнды, как отмечалось, имеют свое название: *приемник и источник*. После выполнения команды изменяется приемник, а источник остается неизменным. Операнды отделяются запятой.

*Комментарии* - все, что стоит справа от «**;**». Могут быть в строке, а также занимать целую строку. Поля отделяются пропуском.

*Псевдооператоры* - руководят работой транслятора, а не микропроцессора. С их помощью определяют сегменты и процедуры (подпрограммы), дают имена командам и элементам данных, резервируют рабочие места в памяти.

Псевдооператоры также могут состоять из 4-х полей:

**[Идентификатор] Псевдооператор [операнд] [; коментарий]**

Поля в скобках могут отсутствовать. В Макроассемблере насчитывается до 60 псевдооператоров. Рассмотрим наиболее распространенные.

### *1.5.3. Псевдооператоры данных*

Их можно разделить на **5** групп:

*1. Определение идентификаторов*

Позволяют присвоить символическое имя выражения, константе, адресу, другому символическому имени. После этого можно использовать это имя.

Напрмер:

```
K         EQU   1024; константа
TABLE     EQU   DS: [BP] [SI]; адрес
SPEED     EQU   RATE ; синоним
COUNT     EQU   CX; имя регистра
```

Операндом в этом псевдооператоре в общем случае может быть выражение:

```
DBL  SPEED EQU 2*SPEED
```

То есть, в выражение могут входить некоторые более простые арифметические и логические действия. Если в выражение входит константа, то по умолчанию она считается десятичной 256.

Шестнадцатиричные константы - справа буква H (2FH). Когда такая константа начинается с буквы, то слева нужно ставить 0 - 0FH, а не FH;

Восьмиричные константы - с буквой Q: 256Q;

Двоичные - с буквой B - 01101B.

Ясно, что в отличие от языков высокого уровня, здесь выражение выполняется во время трансляции транслятором.

Кроме псевдооператора EQU можно употреблять “ = ”.

```
CONST = 59;
CONST = 98;
CONST = CONST + 2.
```

Такое имя может задавать лишь числовую константу, а не символьную и какую-то другую. Это имя можно переопределять, а определеннуюо через **EQU** - НЕЛЬЗЯ.

*2. Определение данных*

Когда ячейка используется для хранения данных, ей можно присвоить имя с помощью псевдооператоров **DB**, **DW**, **DD** (Define Byte, Word, Double Word).

Общая структура:

**[имя] псевдооператор выражение [,]**

**[,]** - один или несколько выражений через запятую.

```
MAX DB 57
WU_MAX DW 5692
A_TABLE DW 25 -592, 643, 954 - массив
```

Когда значение переменной предварительно неизвестно, а будет использовано для результатов, то в поле выражения нужно поставить «**?**».

```
LAMBDA DW ?
```

Для текста можно использовать псевдооператор **DB**.

```
POLITE DB ‘Введите данные опять’.
```

Если записываются одинаковые данные, то можно использовать операцию **DUP** (duplicate) - повторить.

```
BETA DW 15 DUP(0)     или     GAMA DW 3 DUP (4DUP(0))
```

Ясно, что эту же операцию можно применить для резервирования памяти:

```
ALPHA DW 20 DUP (?)
```

Переменные можно использовать не только для хранения данных, но и адресов. Если какая-то переменная имеет название THERE, тогда псевдооператоры

```
THERE DB 123
……
NEAR_THERE DW THERE
NEAR_THERE DD THERE
```

Под именем NEAR_THERE записывают 16-битовый адрес THERE, то есть, ее смещение, а под именем FAR_THERE - 32-битовый адрес THERE, то есть, сегмент + смещение.

*3. Псевдооператоры определения сегмента и процедуры*

Как отмечалось, программа может состоять из нескольких сегментов: кода, данных, стека, дополнительного сегмента.

Для раздела программы на сегменты используются псевдооператоры **SEGMENT** и **ENDS**. Их структура:

**Имя SEGMENT [атрибуты]**
**……**
**Имя ENDS**

Например:

```
DATASEG SEGMENT
      A DW 500
      B DW -258
  SQUARE DB 54, 61, 95, 17
DATASEG ENDS
```

В сегменте данных определяются имена данных и резервируется память для результатов.

У псевдооператора **SEGMENT** могут быть 3 атрибута:

**Имя SEGMENT [выравнивание] [объединение] [класс] [размер сегмента]**

*Выравнивания* определяет, из каких адресов можно размещать сегмент. Часто это есть атрибут **PARA**, который значит, что сегмент нужно разместить из граници параграфа, то есть, начальный адрес кратен 16. По умолчанию принимается PARA.

Может быть PAGE, PARA, WORD, BYTE.

*Объединения* определяет способ обработки сегмента при компоновке.

**PRIVATE** - по умолчанию, сегмент должен быть отделен от других сегментов.

**PUBLIC** - все сегменты с одинаковым именем и классом загружаются в смежной области. Все они будут иметь *один начальный адрес*.

**STACK** - для компоновника это аналогично PUBLIC. В любой программе должен быть один сегмент с атрибутом STACK.

*Класс* - этот атрибут может иметь любое правильное имя, ограниченное апострофами. Атрибут используется компоновщиком для обработки сегментов с одинаковыми именами и классами. Часто используются имена 'CODE' и 'STACK'.

```
STACK SEG SEGMENT PARA STACK 'STACK'
    MAS DW 20 DUP (?)
STACK SEG ENDS.
```

Как отмечалось, процессор использует регистр **CS** для адресации сегмента кода, **SS** - сегмента стека, **DS** - данных, **ES** - дополнительный.

Поскольку транслятор не понимает текста, то ему нужно сообщить назначение каждого сегмента. Для этого существует псевдооператор **ASSUME**, который имеет вид:

**ASSUME SS:имя стека, DS:имя данных, CS:имя кода.**

Например:

```
ASSUME SS:STACK SEG, DS:DATA SEG, CS:CODE SEG.
```

Если какой-то сегмент не используется, то его можно пропустить, или записать:

***DS:NOTHING***

Эта директива лишь сообщает *транслятору* что с чем связать. Но не загружает соответствующие адреса в регистры.Это должен сделать сам программист в программе.

Директива реализуется в сегменте кодов вначале.

Сегмент кодов может включать одну или несколько процедур. Отдельная процедура начинается псевдооператором PROC:

**имя PROC [атрибут]**

**имя ENDP** и заканчивается

```
имя ENDP
```

Если процедура предусматривает возвращение к точке вызова, то перед **ENDP** должна стоять директива **RET** (Return From Procedure). Тогда процедура становится подпрограммой.

```
CALC PROC
.....
        RET
CALC ENDP
```

Атрибутом процедуры может быть **NEAR** (близкий) и **FAR** (далекий).

**NEAR**-процедура может быть вызвана только из этого сегмента.

Процедуру с атрибутом **FAR** можно вызывать из любого сегмента команд. Основная процедура должна иметь атрибут FAR.

### *1.5.4. Программные сегменты*

В программе может быть несколько сегментов с одинаковым именем. Считается, что это - один сегмент, который по каким-то причинам записан частями.

Параметры директивы **SEGMENT** нужны для больших программ, которая состоит из нескольких файлов. Для небольшой программы, которая составляет один файл, эти параметры ненужны, кроме некоторых случаев.

Пусть, имеем такой ряд сегментов:

```
A SEGMENT
   A1 DB 400h DUP(?)
   A2 DW 8
A ENDS
;
B SEGMENT
   B1 DW A2
   B2 DD A2
B ENDS
;
C SEGMENT
ASSUME ES:A, DS:B, CS:C
L: MOV AX, A2
   MOV BX, B2
……….
C ENDS
```

Размещаются сегменты на границе параграфа, то есть, из адреса, кратного 16. Если А размещено 1000h, то он займет место к 1402h. Следующий адрес - 1403h не кратная 16, потому сегмент В разместится из адреса 1410h. Под сегмент В будет отведено 6 байт, а сегмент С разместится из адреса 1420h.

***Значение имени сегмента***

Значением имени сегмента является номер, соответствующий сегменту памяти, то есть, первые 16 битов начального адреса данного сегмента. То есть, **А** будет иметь значение 1000h, **В** - 1410h, **С** - 1420h. Храня в тексте имя сегмента, ассемблер будет замещать его на соответствующую величину. Например:

```
MOV BX, B    отвечает    MOV BX, 1410h
```

Сравните MOV BX, A2.

Следовательно, в языке ассемблер имена сегментов являются константными выражениями, а не адресными. Поэтому команда запишет к BX адрес 1410h, а не содержание слова по этому адресу.

***Начальная загрузка сегментных регистров***

Директива **ASSUME** отмечает, с какими сегментными регистрами нужно связать сегментные регистры. Регистры **DS** и **ES** должен загрузить начальными адресами сам программист.

Пусть, регистр **DS** нужно установить на начало сегмента **B**. Поскольку имя сегмента является константой, то непосредственно в **DS** ее переслать нельзя, а нужно через регистр общего назначения:

```
MOV AX, B
MOV DS, AX
```

Аналогично загружается и регистр **ES**.

Регистр **CS** загружать не нужно. Это выполнит сама операционная система перед тем, как передавать управление программе. Относительно регистра стека **SS** существует две возможности:

- во-первых, это можно сделать так, как и для регистров **DS** и **ES**. Но здесь нужно записать начальный адрес в регистр **SS**, а в указатель стека **SP** – количество байт под стек
- во-вторых - это можно поручить операционной системе. Для этого в соответствующей директиве **SEGMENT** нужно отметить атрибут **STACK**.

### 1.6. Структура программы

Взаимное расположение сегментов программы может быть произвольным. Если сегменты данных расположены после сегмента кода, то в сегменте кода есть ссылка по адресам сегмента данных. То есть, сразу нельзя определить смещение этих адресов. Следовательно, будем иметь ссылку вперед.

Потому для уменьшения количества ссылок вперед рекомендуется сегменты данных и стека размещать перед сегментом кода.

Относительно сегмента стека **SS**, то если даже программа и не использует стек, создать такой сегмент в программе нужно. Потому что стек программы используется операционной системой при обработке прерываний, например, при нажатии клавиш. Рекомендовано размер стека -- 128 байт. Поэтому, программа на языке ассемблер имеет такую структуру:

```
Title EXAMPLE   ;заголовок
; сегмент данных
dat segment
mas dw 1-3,56,91
res dw 10 dup(?)
dat ends
; сегмент стека
st segment stack ‘stack’
dw 128 dup(?)
st ends
; сегмент кода
cod segment ‘code’
      ASSUME DS:dat, SS:st, CS:cod
beg proc far
     <операторы>
    ret
beg endp
cod ends
      end beg
```

В общем случае в сегменте данных можно размещать и команды, а в сегменте кода - данные. Но лучше этого не делать, потому что возникнут проблемы с сегментацией.

**END** - конец программы. Если программа - из одного файла, то в этой строке добавляется имя начального выполняемого адреса - beg. Если из нескольких файлов, то только в одной программе отмечается этот адрес. В других - лишь end.

### *1.6.1. Упрощено описание сегментов*

Есть два транслятора из ассемблера - **MASM** фирмы Microsoft, и **TASM** фирмы Borland. Последний может работать в режиме MASM и IDEAL.

Для простых программ, которые состоят из одного сегмента данных, одного сегмента стека, одного сегмента кода есть возможность упростить директивы описания сегмента. Для этого сначала наводится директива **masm** или **ideal** режима работы транслятора **TASM**. Дальше отмечается модель памяти **Model small**; какая частично выполняет функции директивы **ASSUME**.

Например:

```
Masm ; режим работы транслятора TASM
model small ; модель памяти
.data ; заглавие сегмента данных
mas dw 10 DUP (?)
.stack ; заглавие сегмента стека
db 256 dup (?)
.code ; заглавие сегмента кода
main proc
        mov ax @data ; адрес сегмента стека к ax
        mov ds, ax
; текст программы
        mov ax, 4c00h    ; то же, что и RET
         int 21h
main endp
end main
```

Таблица

Упрощенные директивы определения сегмента

| Режим MASM | Режим IDEAL | Описание |
|---|---|---|
| .code [имя] | Codeseg[имя] | Начало или продолжение сегмента кода |
| .data | Dataseg | Начало или продолжение сегмента иниц-х данных |
| .const | Const | Начало или продолжение сегмента констант |
| .data? | Udataseg | Начало или продолжение сегмента неиниц-х данных |
| .stack [размер] | Stack[размер] | Начало сегмента стека |
| .fardata[имя] | Fardata[имя] | Инициализиров-е данные типа FAR |
| .fardata?[имя] | Ufardata[имя] | Неинициализиров-е данные типа FAR |

Идентификаторы, которые создает директива **MODEL**:

@ code -- физический адрес (смещение) сегмента кода
@ data - физический адрес (смещение) сегмента данных
@ fardata -- физический адрес (смещение) сегмента данных типа far
@ fardata? -- физический адрес (смещение) сегмента неинициализированных данных far
@ curseg -- физический адрес (смещение) сегмента неинициализированных данных типа far
@ stack -- физический адрес (смещение) сегмента стека

***Модели памяти***

| **Модель** | **Тип кода** | **Тип данных** | **Назначение модели** |
|---|---|---|---|
| TINY | Near | Near | Код и данные в одной группе DGROUP для создания .com-программ |
| SMALL | Near | Near | 1 сегмент кода. Данные в одну группу DGROUP |
| MEDIUM | Far | Near | Код занимает ***n*** сегментов, по одному в каждом модуле. Все передачи управления типа **far**. Данные в одной группе; все ссылки на них – типа **near** |
| COMPACT | Near | Far | Код в 1 сегменте; ссылка на на данные типа **far** |
| LARGE | Far | Far | Код в ***n*** сегментах, по одному на каждый объединенный модуль |

Модификатор директивы **MOD**el позволяет определить некоторые особенности выбраной модели памяти:

Use 16 - 16-битовые сегменты

Use 32 - 32-битовые сегменты

DOS - программа в MS DOS

Полное и упрощенный описание не исключают друг друга. Но в полном больше возможностей.

**Лекция 6**

## Раздел 2. Команды МП 8088/86

### 2.1. Состав команд

Микропроцессор имеет ***92 команды***, которые можно разделить на 7 групп:

1. Команды пересылки данных между регистрами, ячейками и портами ввода/вывода;
2. Арифметические команды;
3. Команды над битами, которые осуществляют сдвиги и логические операции;
4. Команды передачи управления, вызова процедур и возвращения из процедуры;
5. Команды обработки строк;
6. Команды прерывания для обработки специфических событий;
7. Команды управления процессором - установление и сброс флагов состояния, изменения режима функционирования МП.

#### *2.1.1. Команды пересылки данных*

Команда ***MOV*** - чаще всего употребляется в программе. Ее можно применять для пересылки:

```
MOV AX, CX ; из регистра в регистр
MOV AX, TABLE ; из памяти в регистр
MOV TABLE, AX ; из регистра в память
MOV DS, AX ; из регистра в регистр сегмента
MOV AH, AL ; обмен байтами
MOV AX, -40 ; константу в регистр
MOV BETA, 2Fh ; константу в память
```

***НЕЛЬЗЯ*** осуществлять такие пересылки:

Из памяти в память. Как отмечалось, в двухадресных командах нельзя использовать прямую адресацию в двух операндах. Поэтому пересылку память-память можно осуществить двумя командами:

~~MOV ALPHA, BETA~~

```
MOV AX, BETA
MOV ALPHA, AX
```

Содержание ячеек нельзя пересылать непосредственно в регистр сегмента, а только через регистр общего назначения:

~~MOV DS, BETA~~

```
MOV AX, BETA
MOV DS, AX
```

Нельзя пересылать данные из одного регистра сегмента в другой, а только через регистр общего назначения:

```
MOV AX, DS
MOV ES, AX
```

~~MOV ES, DS~~

Нельзя использовать регистр CS как приемник, то есть, **НЕЛЬЗЯ**:

~~MOV CS, AX~~

Стек автоматически создается для работы с подпрограммами. Но поскольку в МП лишь 4 регистра общего назначения, то часто придется запоминать содержание регистров, чтобы освободить регистры и выполнить другие действия. Именно для этого используются команды:

***PUSH источник*** ; заслать в стек

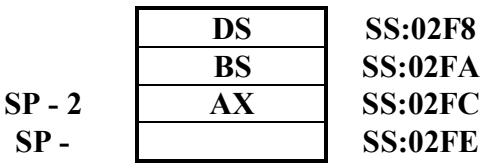
***POP приемник*** ; считать из стека

Например:

```
PUSH SI
PUSH DS
PUSH CX
PUSH ALPHA
PUSH DELTA [BI + SI]
```

Как отмечалось, данные засылаютсяя на вершину стека, поэтому при их считывании необходимо сдерживаться соответствующей последовательности. Например:

```
PUSH AX
PUSH BX
PUSH DS
```




Поскольку сверху находится **DS**, а ниже - **BX** и **AX**, то возобновление регистров нужно осуществить в противоположном порядке

```
POP DS
POP BX
POP AX
```

Команды PUSH-POP можно использовать для обмена между сегментными регистрами:

```
PUSH DS
POP ES
```

При этом не используется РЗП. Но выполнение команд будет дольше: пара PUSH-POP реализуется за 26 тактов, а две команды MOV за 4 такта.

### *2.1.2. Команда обмена XCHG*

Название происходит от английского слова exchange - обменять. Используется для обмена содержанием двух регистров или регистра и памяти.

~~XCHG CS, DS~~

```
XCHG BX, AX
XCHG AH, BL
XCHG AX, TABLE
XCHG TABLE, AX
```

НЕЛЬЗЯ использовать для обмена между сегментными регистрами.

### *2.1.3. Команды обмена с портами*

***IN аккумулятор, порт***

***OUT порт, акумулятор***

Аккумулятор - это регистр **AX** при обмене **словами** и **AL** при обмене **байтами**. Порт определяется своим номером от 0 до 256. Можно непосредственно отмечать номер порта или дать ему имя.

```
PORT_NUM EQU 210
IN AX, 200
IN AL, PORT_NUM
OUT DX, AX
OUT 200, AL
```

Также номер порта можно записать в регистр **DX**.

### *2.1.4. Команда LEA - загрузки эффективного адреса*

***LEA*** ( load effective address - загрузить исполнительный адрес ) пересылает смещение ячейки памяти в:

1. определенный 16-битовый регистр ЗУ
2. регистр указателя
3. индексный регистр

***LEA регистр 16, память 16***

В отличие от команды MOV с операцией OFFSET, операнд память 16 может быть индексированным, что обеспечивает гибкость адресации.
Например:

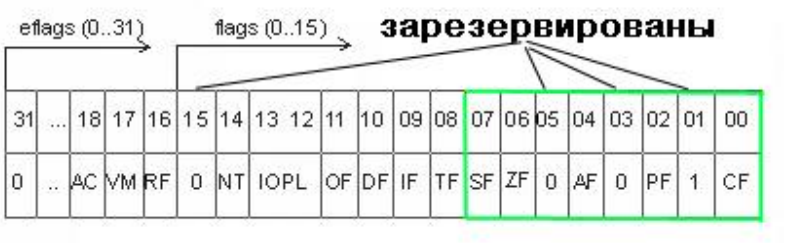
```
LEA BP, TABLE [DI]
```

Если в **DI** содержится 8, то в **BP** будет заслан адрес TABLE + 8.

### *2.1.5. Команды пересылки флагов*

Можно пересылать в регистр **AH** младший байт регистра флагов **F** командой **LAHF**;

```
LAHF; F→AH
```




Обратная пересылка с **AH** в младший байт регистра F - SAHF:

```
SAHF; AH→F
```

Содержание регистра флагов можно также пересылать в стек командой **PUSHF**, а в обратном направлении - **POPF**.

Это нужно для защиты регистра флагов от изменения при вызове процедур. Если нет уверенности, что процедура не изменит регистр флагов, то его нужно защитить (сохранить).

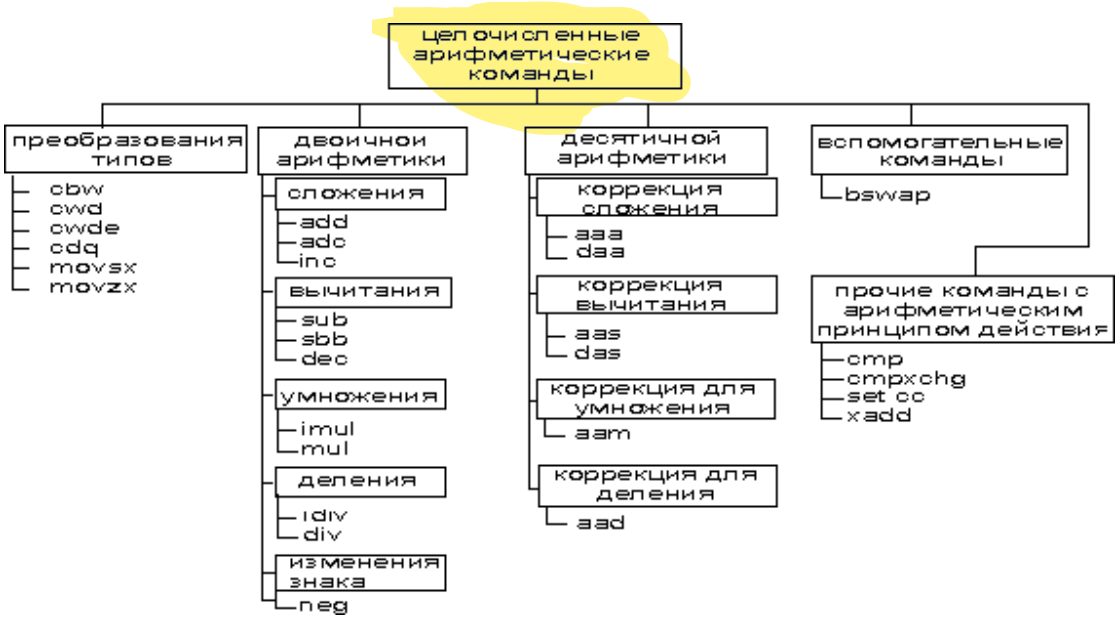
Пример:

```
PUSH AX
PUSH DI
PUSHF
CALL SORT
POPF
POP DI
POP AX
```

# Лекция 7

## 2.2 Арифметические команды

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд.




В МП 8088/86 есть группа команд для реализации 4 арифметических действий (добавление, вычитание, умножение, деление) над целыми числами. При этом число может быть записано как в байт, так и в слово. Содержание байта или слова можно рассматривать как число со знаком, или без него. Некоторые разновидности этих команд позволяют реализовать арифметические действия для чисел повышенной точности. Кроме этого, с помощью обычных арифметических операций можно реализовать действия над числами в ВСД-формате. Ясно, что результат будет правильным. Однако его можно скорректировать. Поэтому предусмотрено для такого случая команды коррекции результатов. Их рассматривать не будем. Напомним, что большинство двухоперандных команд действуют так, что изменяется операнд-приемник, а операнд-источник остается неизменным.

*1 — после выполнения команды флаг устанавливается (равен 1);*
*0 — после выполнения команды флаг сбрасывается (равен 0);*
*r — значение флага зависит от результата работы команды;*
*? — после выполнения команды флаг не определен;*
*пробел — после выполнения команды флаг не изменяется;*

### *2.2.1. Команды добавления*

Существует три команды:
**ADD** (add -- прибавить);
**ADC** (add with carry) - прибавить с переносом;
**INC** (increment) - прибавить единицу.

**ADD операнд_1, операнд_2** — команда сложения с принципом действия:
***операнд_1 = операнд_1 + операнд_2***

```
ADD AX, CX              AX + CX → AX
```

В этой команде могут наводиться 2 регистра общего назначения, один операнд слово памяти (со всеми возможными режимами адресации) и непосредственные данные как источник.

```
ADD AX, ALPHA
ADD ALPHA, AX
ADD AH, 20
ADD ALPHA, 30
```

Эта команда влияет на 6 флагов.

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| r | r | r | r | r | r |

**ADC операнд_1, операнд_2** — команда сложения с учетом флага переноса **cf**.
Принцип действия команды:
***операнд_1 = операнд_1 + операнд_2 + значение_cf***

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| r | r | r | r | r | r |

**INC операнд** - увеличение значения операнда в памяти или регистре на 1

| 11 | 07 | 06 | 04 | 02 |
|---|---|---|---|---|
| OF | SF | ZF | AF | PF |
| r | r | r | r | r |

### *2.2.2. Команды вычитания*

Аналогично к командам добавления, существуют две команды вычитания:
**SUB** (substract) – вычесть;
**SBB** (substract with borrow) - вычесть с заемом;
**DEC** (DECrement) - вычесть единицу.

**SUB операнд_1, операнд_2** - целочисленное вычитание
***операнд_1=операнд_1-операнд_2***

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| r | r | r | r | r | r |

**SBB операнд_1, операнд_2** - целочисленное вычитание с учетом результата предыдущего вычитания командами sbb и sub (по состоянию флага переноса **cf**):

1. выполнить сложение операнд_2 = операнд_2+(**cf**);
2. выполнить вычитание операнд_1 = операнд_1-операнд_2;

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| r | r | r | r | r | r |

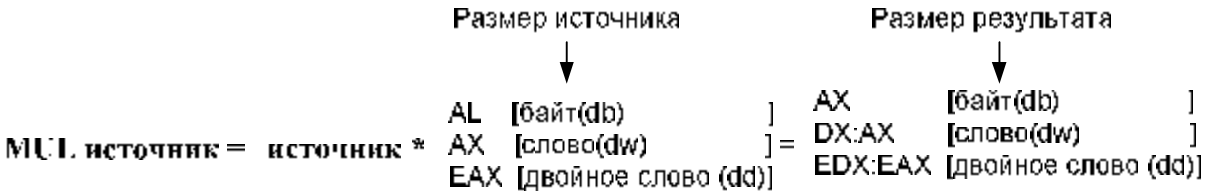
**DEC операнд** - - уменьшение значения операнда в памяти или регистре на 1

| 11 | 07 | 06 | 04 | 02 |
|---|---|---|---|---|
| OF | SF | ZF | AF | PF |
| r | r | r | r | r |

### *2.2.3. Команды умножения*

Существует две команды:
**MUL** (multiply) - умножение без знака;
**IMUL** (integer multiply) - умножение со знаком.

**MUL источник**



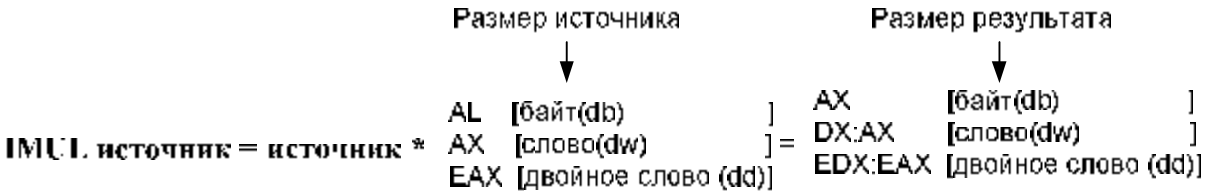

Состояние флагов после выполнения команды
(если ***старшая половина результата нулевая AH=0;DX=0;EDX=0***):

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| 1 | ? | ? | ? | ? | 0 |

Состояние флагов после выполнения команды:
(если ***старшая половина результата ненулевая AH≠0;DX≠0;EDX≠0***):

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| 1 | ? | ? | ? | ? | 1 |

**IMUL источник**




**IMUL множ_1, множ_2** **(** IMUL AX, 5 AX = AX*5 **)**
**IMUL рез-т, множ_1, множ_2** **(** IMUL DI, AX, 5 DI = AX*5 **)**

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| r | ? | ? | ? | ? | r |

### *2.2.4.* Команды деления

Отметим, что деление выполняется как деление целых чисел, то есть, получим частное (целое) и остаток (целое). Следовательно, 19 : 5 = 3 и 4 - остаток. Никаких вещественных чисел не получим.

**DIV** (divide) - деления чисел без знака;

**IDIV** (integer divide) - деление целых чисел (со знаком).

**DIV делитель**

| делимое / делитель | Размер делимого: Байт (db) | Слово (dw) | Двойное слово (dd) |
|---|---|---|---|
| **делимое** | AX | DX:AX | EDX:EAX |
| частное | AL | AX | EAX |
| остаток | AH | DX | EDX |

Результаты команд деления на **флаги не влияет**.

Когда же частицу полностью нельзя разместить в отведенном ей слове или байте, то осуществляется прерывание типа 0 - деление на 0.

**IDIV делитель**

Остаток всегда имеет знак делимого. Знак частного зависит от состояния знаковых битов (старших разрядов) делимого и делителя.

### *2.2.5. Команды изменения знака*

**NEG** (NEGate operand) - изменить знак операнда

**NEG источник**

Состояние флагов после выполнения команды (если результат нулевой):

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| r | r | r | r | r | 0 |

Состояние флагов после выполнения команды (если результат ненулевой):

| 11 | 07 | 06 | 04 | 02 | 00 |
|---|---|---|---|---|---|
| OF | SF | ZF | AF | PF | CF |
| r | r | r | r | r | 1 |

### *2.2.6. Команды расширения знака*

Чтобы правильно подать код числа, которое записано в младшем байте, при считывании его из целого слова и код числа, записанного в младшее слово, при считывании его из младшего слова, нужно расширить знак числа соответственно на старший байт и старшее слово.

Для этого существуют две команды:

**CBW** (convert byte to word) - преобразование байта в слово;

**CWD**(convert word to double word) - преобразование слова в двойное слово;

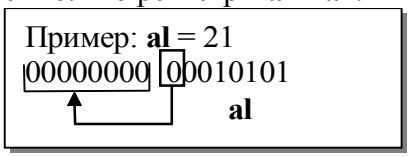
**CWDE** (convert word to double word Extended) - преобразование слова в двойное слово;

**CDQ** (Convert Double word to Quad word) - преобразование двойного слова в учетверенное слово.

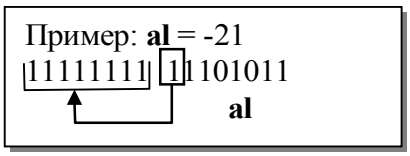
*Алгоритм работы:*

**CBW** — при работе команда использует только регистры **al** и **ax**:

анализ знакового бита регистра **al**:
если знаковый бит **al**=0, то **ah**=00h;




анализ знакового бита регистра **al**:
если знаковый бит **al**=1, то **ah**=0ffh




**CWD** — при работе команда использует только регистры **ax** и **dx**:
анализ знакового бита регистра al:
если знаковый бит **ax**=0, то **dx**=00h;
если знаковый бит **ax**=1, то **dx**=0ffh.

**CWDE** — при работе команда использует только регистры **ax** и **eax**:
анализ знакового бита регистра ax:
если знаковый бит **ax**=0, то установить старшее слово **eax**=0000h;
если знаковый бит **ax**=1, то установить старшее слово **eax**=0ffffh.
выполнение команды ***не влияет на флаги***

**CDQ** — при работе команда использует только регистры **eax** и **edx**:
анализ знакового бита регистра eax:
если знаковый бит **eax** =0, то установить старшее слово **edx** =0000h;
если знаковый бит **eax** =1, то установить старшее слово **edx** =0ffffh.
выполнение команды ***НЕ влияет на флаги***

*Пример:*

Данные команды используются для приведения операндов к нужной размерности с учетом знака. Такая необходимость может, в частности, возникнуть при программировании арифметических операций.

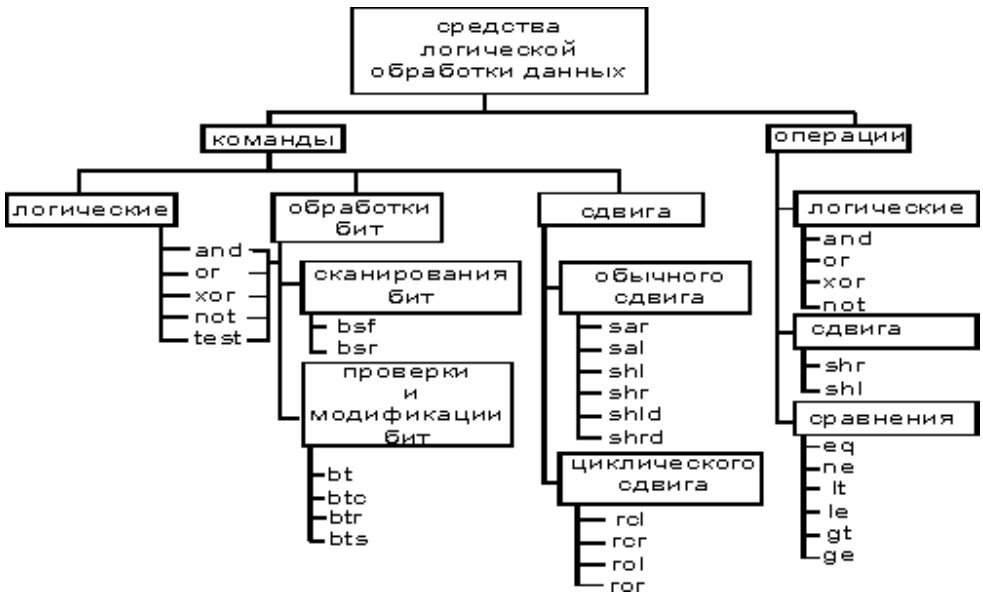
```
.386    ;только для cwde, cwd была для i8086
        mov     ebx,10fecd23h
        mov     ax,-3   ;ax=1111 1111 1111 1101
        cwde    ;eax=1111 1111 1111 1111 1111 1111 1111 1101
        add     eax,ebx
```

# Лекция 8

## 2.3. Команды логической обработки данных

Команды разделяются на 4 группы:

1. логические команды;
2. команды сдвига;
3. команды циклического сдвига;
4. команды обработки бит данных.




### *2.3.1. Логические команды* (AND, OR, XOR, NOT, TEST)

Эти команды реализуют поразрядные операции, то есть, i-тый разряд результата зависит от i-тых разрядов операндов. Операции выполняются параллельно над всеми разрядами. “Истина” - когда будет 1 хотя бы в одном разряде результата, и “Ложь” - когда во всех битах результата - нули.

Команды изменяют все флаги условий, но чаще обращаем внимание на бит **ZF**. Если результат - “истина”, то ZF = 0, а если “ Ложь”, то ZF = 1.

Операндами логических операций могут быть слова или байты, но **НЕ** одновременно.

**Команда побитовое ”AND”**

**AND приемник, источник** - операция логического умножения. Команда выполняет поразрядно логическую операцию “И” (конъюнкцию) над битами операндов **приемник** и **источник**. Результат записывается на место **приемника.**

*Например:*

```
AND AX, BX; AX*BX - результат записать в AX
```

В бите приемника устанавливается 1 тогда, когда в соответствующих битах источника и приемника были 1. Если же в бите источника был 0, то в соответствующем бите приемника установится 0, независимо от того, что там было. Поэтому, команда **AND** используется для

селективного установления 0 в тех битах приемника, которым отвечает 0 в источнике. Подбирая соответствующие биты источника, влияем на определенные биты приемника. Такие действия часто используются в операциях над битами для управления устройствами обмена. При этом оператор источник называется маской, а сама операция - *маскированием*.

Например:

```
AND  10010101 – приемник
     01011011 – источник (маска)
     00010001
          ↑   ↑
     состояние изменилось
```

```
; AX=95h, BX=5Bh
AND AX, BX; AX = 11h
```

Операндами команды **AND** могут быть **байты** или **слова**. Могут использоваться два регистра, регистр со словом (байтом памяти), или непосредственное значение:

```
AND AL, M_BYTE
AND M_BYTE, AL
AND TABLE [BX], MASK
AND BL, 1101B
```

Следовательно, команда **AND** изменяет приемник. Однако, когда бит изменяется лишь самим устройством, то можно использовать команду **AND** для проверки состояния устройства.

Например, порт 200 соединенный с 16-битовым регистром внешнего устройства и 6-й бит показывает, включено (1) или НЕ включено (0) это устройство.

```
CHECK: IN AX, 200
       AND AX, 1 000 000B
       JZ CHECK ; НЕ включено
```

Программа может работать дальше, когда лишь устройство включено:

Если устройство НЕ включено, то в 6-ом бите - 0, и результат команды **AND** - 0, следовательно, **ZF** = 1 и выполняется команда JZ.

Как только в бите станет 1, **ZF** = 0, и команда JZ НЕ выполняется.

| 1 | 7 | 6 | 2 | 0 |
|---|---|---|---|---|
| F | F | F | F | F |
| | | | | |

**Команда побитовое ”OR ”**

**OR приемник, источник** — операция логического сложения.

Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов приемник и источник. Результат записывается на место приемник:

```
OR AX, BX ;AX+BX - результат записать в AX
```

**OR** **10**01**0**0**0**1 – приемник
0**1**01**1**0**1**1 – источник (маска)
1**1**011**0**1
состояние изменилось

Следовательно, команду **OR** употребляют для селективного установления 1 в приемнике.

```
; AX=91h, BX=5Bh
OR AX, BX; AX = DBh
```

**Команда побитовое ”XOR ”**

**XOR приемник, источник** — операция логического исключающего сложения.

Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов приемник и источник. Результат записывается на место приемник.

Устанавливает 1 в те биты приемника, которые отличаются от битов источника. Пример:

```
XOR AX, TEST_PAR
JZ ALPHA
```

Если хотя бы в одном бите не совпадают коды, то результат команды **XOR**=1 и ZF=0, следовательно, команда **JZ** не будет выполнена. Она будет выполняться лишь тогда, когда коды полностью совпадают. Команда **XOR** изменяет приемник.

**XOR** 1**1**01**0**01**1** – приемник
0**1**00**1**00**1** – источник (маска)
1**0**011**1**01**0**
состояние изменилось

```
; AX=93h, BX=49h
XOR AX, BX; AX = 9Ah
```

| 1 | 7 | 6 | 4 | 2 | 0 |
|---|---|---|---|---|---|
| F | F | F | F | F | F |
| | | | | | |

**Команда побитовое ” NOT ”**

**NOT операнд** — операция логического отрицания.

Результат записывается на место операнда. Команда NOT изменяет все биты на противоположные:

Пример:

```
flag    db      0ffh ; значение флага — истина
...
cycl:
...
        CMP     flag,0
        JE      m1
...
m1:     not     flag    ;установить флаг в истину
```

выполнение команды ***НЕ влияет на флаги***

```
; AX=10011010b
NOT AX; AX = 01100101b
```

**Команда побитовое ” TEST ”**

**TEST приемник, источник** — операция “проверить” (способом логического умножения).

Команда выполняет поразрядно логическую ***операцию И*** над битами операндов **приемник и источник**. Состояние операндов остается прежним, изменяются только флаги ***zf, sf,*** и ***pf***, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

Если хотя бы одна пара битов равняется 1, то результат команды равняется 1, следовательно, **ZF** = 0.

Например:

```
TEST    al,01h
JNZ     m1      ;переход, если нулевой бит al равен 1
```

**TEST** 0**11**011**0**1 – приемник
0**11**00**1**00 – источник (маска)
0**11**00**1**00
↑↑ ↑
проверены биты

| 1 | 7 | 6 | 2 | 0 |
|---|---|---|---|---|
| F | F | F | F | F |
| | | | | |

### *2.3.2. Команды сдвига*
### (SHL, SHR, SAL, SAR, SHLD, SHRD, ROL, ROR, RCL, RCR)

К этой группе принадлежит 10 команд.

1. **6** сдвигают операнд (**SHL, SHR, SAL, SAR, SHLD, SHRD**)
2. **4** вращают или циклически сдвигают (**ROL, ROR, RCL, RCR**).

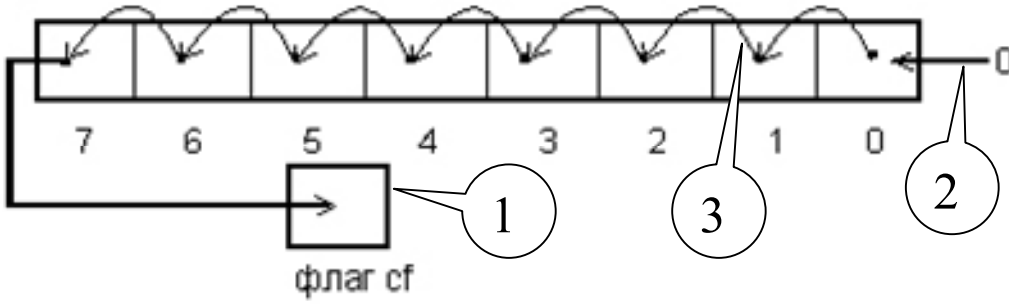
**SHL, SHR, SAL, SAR, SHLD, SHRD**

Алгоритм:

1. очередной "выдвигаемый" бит устанавливает флаг **CF;**
2. бит, вводимый в операнд с другого конца, имеет значение 0;
3. при сдвиге очередного бита он переходит во флаг **cf**, при этом значение предыдущего сдвинутого бита теряется!

**SHL операнд, счетчик_сдвигов** (Shift Logical Left) - логический сдвиг ***влево***.

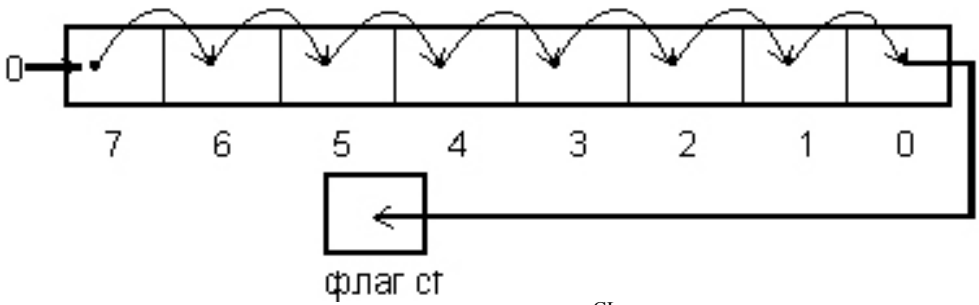
Содержимое операнда сдвигается влево на количество битов, определяемое значением **счетчик_сдвигов**. Справа (в позицию младшего бита) вписываются нули;




SHL AX, CL -- умножить AX без знака на $2^{CL}$

**SHR операнд, счетчик_сдвигов** (Shift Logical Right) — логический сдвиг ***вправо***.

Содержимое операнда сдвигается вправо на количество битов, определяемое значением **счетчик_сдвигов**. Слева (в позицию старшего, знакового бита) вписываются нули. На рис. показан принцип работы этих команд.

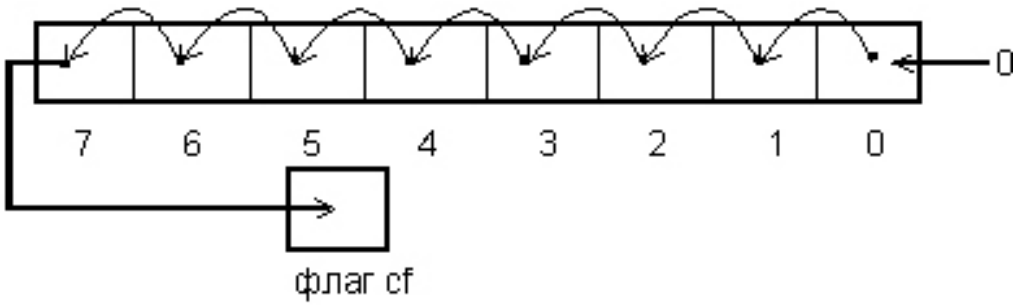



SHR AX, CL -- разделить AX без знака на $2^{CL}$

**SAL операнд, счетчик_сдвигов** (Shift Arithmetic Left) — арифметический сдвиг ***влево***.

Содержимое операнда сдвигается влево на количество битов, определяемое значением **счетчик_сдвигов**. Справа (в позицию младшего бита) вписываются нули. Команда **SAL** не

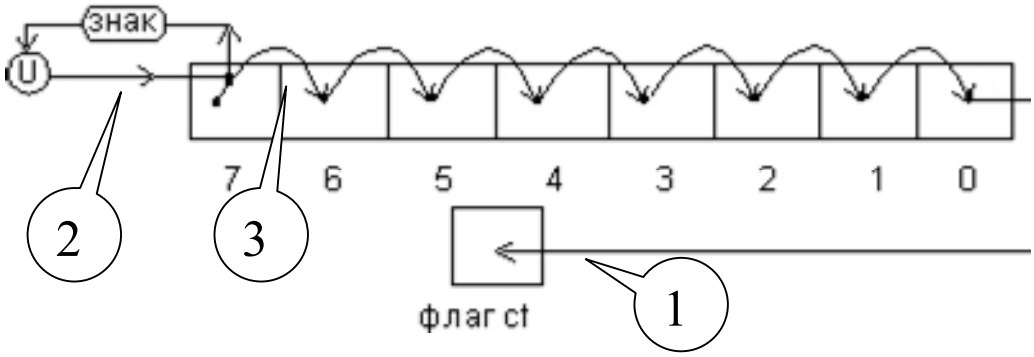
сохраняет знака, но устанавливает флаг ***cf*** в случае смены знака очередным выдвигаемым битом. В остальном команда SAL полностью аналогична команде **SHL**;




SAL AX, CL -- умножить AX со знаком на $2^{CL}$

**SAR операнд, счетчик_сдвигов** (Shift Arithmetic Right) — арифметический сдвиг ***вправо***.

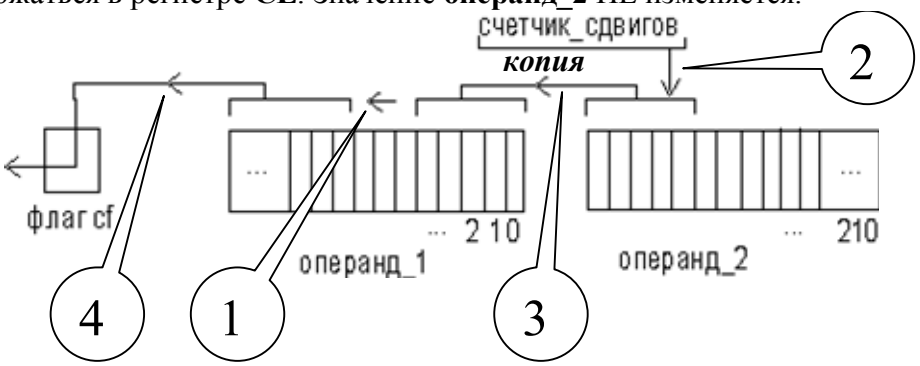
Содержимое операнда сдвигается вправо на количество битов, определяемое значением **счетчик_сдвигов**. Команда **SAR** сохраняет знак, восстанавливая его после сдвига каждого очередного бита.




SAR AX, CL - разделить AX со знаком на $2^{CL}$.

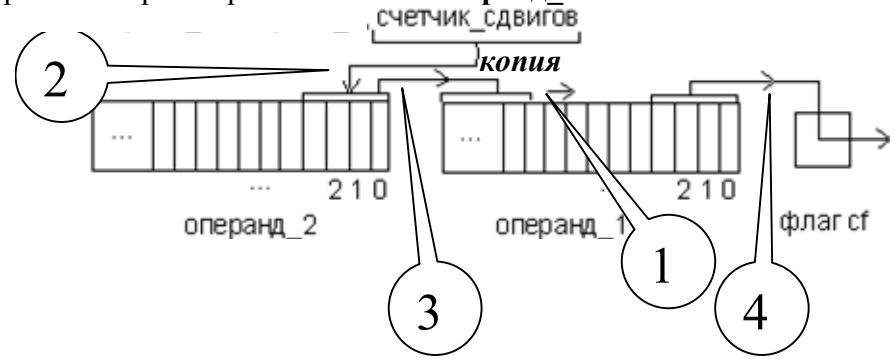
**SHLD операнд_1,операнд_2, счетчик_сдвигов** — сдвиг ***влево*** двойной точности.

Команда **SHLD** производит замену путем сдвига битов операнда **операнд_1** влево, заполняя его биты справа значениями битов, вытесняемых из **операнд_2** согласно схеме на рис. Количество сдвигаемых бит определяется значением **счетчик_сдвигов**, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственным операндом или содержаться в регистре **CL**. Значение **операнд_2** НЕ изменяется.



**SHRD операнд_1,операнд_2,счетчик_сдвигов** — сдвиг ***вправо*** двойной точности.

Команда производит замену путем сдвига битов операнда **операнд_1** вправо, заполняя его биты слева значениями битов, вытесняемых из **операнд_2** согласно схеме на рис. Количество сдвигаемых бит определяется значением счетчик_сдвигов, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственным операндом или содержаться в регистре **cl**. Значение **операнд_2** НЕ изменяется.




### *2.3.3. Команды циклического сдвига*

#### *1. Команды простого циклического сдвига*

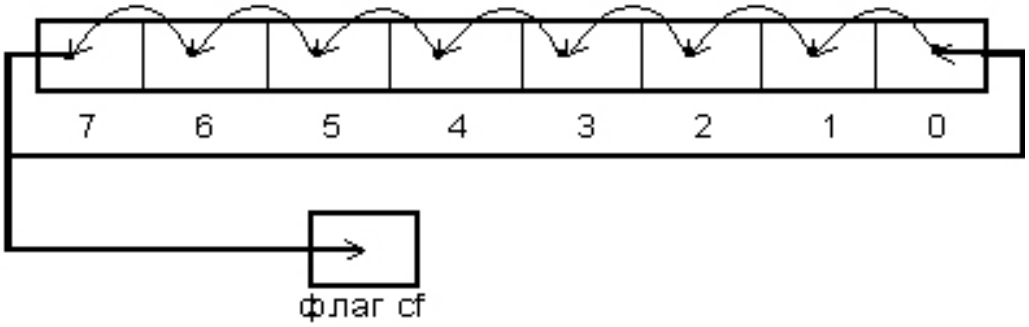
**ROL, ROR**

Алгоритм:

1. сдвиг всех битов операнда влево на один разряд, при этом старший бит операнда вдвигается в операнд справа и становится значением младшего бита операнда;
2. одновременно выдвигаемый бит становится значением флага переноса **cf**;
3. указанные выше два действия повторяются количество раз, равное значению второго операнда.

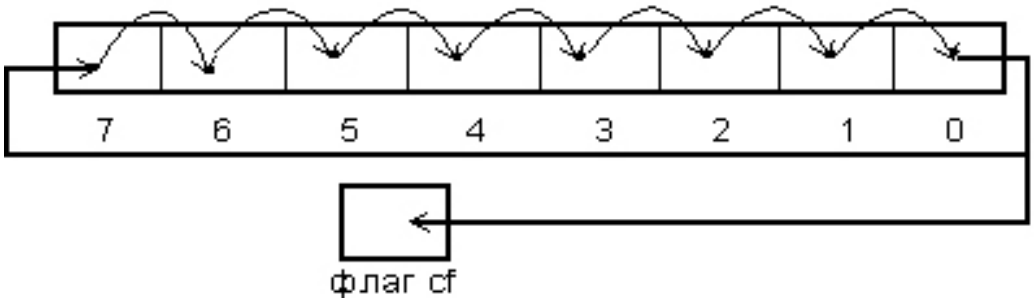
**ROL операнд, счетчик_сдвигов** (Rotate Left) — циклический сдвиг ***влево***.

Содержимое операнда сдвигается влево на количество бит, определяемое операндом **счетчик_сдвигов**. Сдвигаемые влево биты записываются в тот же операнд справа.




**ROR операнд, счетчик_сдвигов** (Rotate Right) — циклический сдвиг ***вправо***.

Содержимое операнда сдвигается вправо на количество бит, определяемое операндом **счетчик_сдвигов**. Сдвигаемые вправо биты записываются в тот же операнд слева.




***2. Команды циклического сдвига через флаг переноса cf***

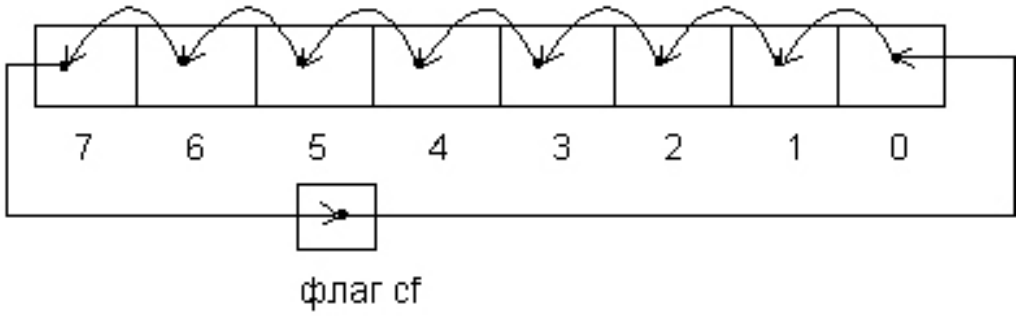
**RCL, RCR**

Алгоритм:

1. сдвиг всех битов операнда влево на один разряд, при этом старший бит операнда становится значением флага переноса **cf**;
2. одновременно старое значение флага переноса **cf** сдвигается в операнд справа и становится значением младшего бита операнда;
3. указанные выше два действия повторяются количество раз, равное значению второго операнда команды **rcl**.

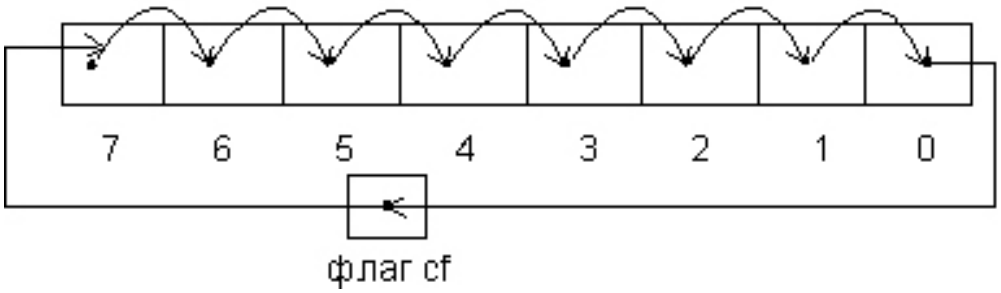
**RCL операнд, счетчик_сдвигов** (Rotate through Carry Left) — циклический сдвиг ***влево*** через перенос.

Содержимое операнда сдвигается влево на количество бит, определяемое операндом **счетчик_сдвигов**. Сдвигаемые биты поочередно становятся значением флага переноса **cf**.




**RCR операнд, счетчик_сдвигов** (Rotate through Carry Right) — циклический сдвиг ***вправо*** через перенос.

Содержимое операнда сдвигается вправо на количество бит, определяемое операндом **счетчик_сдвигов**. Сдвигаемые биты поочередно становятся значением флага переноса **cf**.




Эти команды выполняются намного быстрее, чем **MUL** и **DIV**

# Лекция 9

## 2.4. Команды передачи управления

Команды передачи управления подразделяются на:

1. ***безусловные*** — в данной точке необходимо передать управление не той команде, которая идет следующей, а другой, которая находится на некотором удалении от текущей команды;
2. ***условные*** — решение о том, какая команда будет выполняться следующей, принимается на основе анализа некоторых условий или данных;
3. ***управления циклом***;
4. ***вызов процедур***.

### *2.4.1. Команда безусловного перехода JMP*

**JMP [модификатор] адрес_перехода** - безусловный переход без сохранения информации о точке возврата.

*Модификатор* может принимать следующие значения:

- ***near ptr*** — прямой переход на метку внутри текущего сегмента кода. Модифицируется только регистр **eip/ip** (в зависимости от типа сегмента кода use16/32) на основе указанного в команде адреса (метки) или выражения;

```
JMP pt ;Переход на метку pt в пределах текущего сегмента
JMP near ptr pt ;то же самое
```

- ***far ptr*** — прямой переход на метку в другом сегменте кода. Адрес перехода задается в виде непосредственного операнда или адреса (метки) и состоит из 16-битного селектора **cs** и 16/32-битного смещения **ip/eip**;

```
JMP far ptr farpt ;Переход на метку farpt в другом программном
сегменте
JMP farpt ;Переход на метку farpt в другом
;программном сегменте, если farpt
;объявлена дальней меткой директивой farpt label far
```

- ***word ptr*** — косвенный переход на метку внутри текущего сегмента кода. Модифицируется (значением смещения из памяти по указанному в команде адресу, или из регистра) только **eip/ip**. Размер смещения 16 или 32 бит;

```
;В полях данных:
addr dw pt ;Ячейка с адресом точки перехода
;В программном сегменте:
jmp DS:addr ;Переход в точку pt
JMP word ptr addr ;То же самое
```

- ***dword ptr*** — косвенный переход на метку в другом сегменте кода. Модифицируются (значением из памяти, **из регистра нельзя**) оба регистра, **cs** и **eip/ip**. Первое слово/двойное слово этого адреса представляет смещение и загружается в **ip/eip**; второе/третье слово загружается в **cs**.

```
; В полях данных:
addr dd pt ;Поле с двухсловным
;адресом точки перехода ;В программном сегменте:
jmp DS:addr ;Переход в точку pt
JMP dword ptr addr ;То же самое
```

Кроме прямого перехода, можно осуществить непрямой переход. При этом в команде **JMP** отмечается имя слова, где хранится адрес перехода, или имя регистра, где записывается адрес перехода.

***Примеры косвенных ближних переходов***

*Пример 1*

```
mov BX,offset pt ;BX=адрес точки перехода
jmp BX ;Переход в точку pt
```

*Пример 2*

```
; В полях данных:
addr dw pt ;Ячейка с адресом точки перехода
;В программном сегменте:
mov DI,offset addr ;B1=адрес ячейки с адресом
;точки перехода
jmp [DI] ;Переход в точку pt
```

*Пример 3*

```
;В полях данных:
tbl dw ptl ;Ячейка с адресом 1
dw pt2 ;Ячейка с адресом 2
dw pt3 ;Ячейка с адресом 3
;В программном сегменте:
mov BX,offset tbl ;BX=aflpec таблицы адресов переходов
mov SI, 4 ;31=смещение к адресу pt3
call [BX][SI] ;Переход в точку pt3
```

### *2.4.2. Команды условного перехода*

Всего предусматривается ***17 команд*** условного перехода. Но некоторые команды имеют несколько синонимов (мнемоник). Поэтому иногда говорят о ***31 команде***.

Эти команды позволяют проверить:

1. отношение между операндами со знаком (“больше — меньше”);
2. отношение между операндами без знака (“выше — ниже”);
3. состояния арифметических флагов **zf**, **sf**, **cf**, **of**, **pf** (но Не **af**).

Команды условного перехода имеют одинаковый синтаксис:
**JCC метка_перехода**

Как видно, мнемокод всех команд начинается с “**J**” — от слова jump (прыжок), **CC** — определяет конкретное условие, анализируемое командой.

Что касается операнда **метка_перехода**, то эта метка может находится *только в пределах текущего сегмента* кода, межсегментная передача управления в условных переходах не допускается.

Команда сравнения **CMP** имеет интересный принцип работы. Он абсолютно такой же, как и у команды вычитания.

**CMP операнд_1, операнд_2** (compare) — сравнивает два операнда, вычитая **операнд_2** из **операнд_1**, не изменяя их и по результату устанавливает флаги.

Таблица

Значение аббревиатур в названии команды JCC

| Мнемоническое обозначение | Английский | Русский | Тип операндов |
|---|---|---|---|
| E или e | equal | Равно | Любые |
| N или n | not | Не | Любые |
| G или g | greater | Больше | Числа со знаком |
| L или l | less | Меньше | Числа со знаком |
| A или a | above | Выше, в смысле “больше” | Числа без знака |
| B или b | below | Ниже, в смысле “меньше” | Числа без знака |

**Команды реакции на арифметические сравнения со знаком**

Для таких сравнений используются слова “меньше” (less) и “больше” (greater). Ясно, что можно проверить 6 условий:

| Типы операндов | Мнемокод команды условного перехода | Критерий условного перехода | Значения флагов для осуществления перехода |
|---|---|---|---|
| Любые | JE | операнд_1 = операнд_2 | zf = 1 |
| Любые | JNE | операнд_1<>операнд_2 | zf = 0 |
| Со знаком | JL / JNGE | операнд_1 < операнд_2 | sf <> of |
| Со знаком | JLE / JNG | операнд_1 <= операнд_2 | sf <> of or zf = 1 |
| Со знаком | JG / JNLE | операнд_1 > операнд_2 | sf = of and zf = 0 |
| Со знаком | JGE / JNL | операнд_1 => операнд_2 | sf = of |

*Например:*

```
CMP AX, BX
JL LABEL2; переход, если AX<BX
```

**Команды реакции на арифметические сравнения без знака**

| Типы операндов | Мнемокод команды условного перехода | Критерий условного перехода | Значения флагов для осуществления перехода |
|---|---|---|---|
| Любые | JE | операнд_1 = операнд_2 | zf = 1 |
| Любые | JNE | операнд_1<>операнд_2 | zf = 0 |
| Без знака | JB / JNAE | операнд_1 < операнд_2 | cf = 1 |
| Без знака | JBE / JNA | операнд_1 <= операнд_2 | cf = 1 or zf=1 |
| Без знака | JA / JNBE | операнд_1 > операнд_2 | cf = 0 and zf = 0 |
| Без знака | JAE / JNB | операнд_1 => операнд_2 | cf = 0 |

Для таких сравнений используются слова “выше” (above) и ”ниже” (below), после сравнения (**CMP**) адресов:

**Команды проверки отдельных флагов и регистров**

Мнемоническое обозначение некоторых команд условного перехода отражает название флага, с которым они работают, и имеет следующую структуру: первым идет символ “**J**” (jump, переход), вторым — либо обозначение флага, либо символ отрицания “**N**”, после которого стоит название флага.

| Название флага | № бита в eflags/flag | Команда условного перехода | Значение флага для осуществления перехода |
|---|---|---|---|
| Флаг переноса cf | 1 | JC | cf = 1 |
| Флаг четности pf | 2 | JP | pf = 1 |
| Флаг нуля zf | 6 | JZ | zf = 1 |
| Флаг знака sf | 7 | JS | sf = 1 |
| Флаг переполнения of | 11 | JO | of = 1 |
| Флаг переноса cf | 1 | JNC | cf = 0 |
| Флаг четности pf | 2 | JNP | pf = 0 |
| Флаг нуля zf | 6 | JNZ | zf = 0 |
| Флаг знака sf | 7 | JNS | sf = 0 |
| Флаг переполнения of | 11 | JNO | of = 0 |

| Типы операндов | Команда условного перехода | Критерий условного перехода | Зн. регистров для перехода |
|---|---|---|---|
| Любые | JCXZ | Jump if **cx** is Zero | cx = 0 |
| Любые | JECXZ | Jump Equal **ecx** Zero | ecx = 0 |

*Например:*

```
CMP AX, BX
JE cycl
JCXZ    m1  ; обойти цикл, если cx=0
cycl:;некоторый цикл
LOOP  cycl
m1:     ...
```

### 2.4.3. Команды управления циклами

Поскольку в программах часто придется реализовывать циклические процедуры, то для этого существует несколько команд.

**LOOP метка** - уменьшает регистр **CX** на 1 и осуществляет переход на определенную метку, если содержание **CX ≠ 0**, если **CX** = 0, осуществляется переход на следующую за **LOOP** команду.

Эти команды используют регистр **CX** как счетчик цикла.

Следовательно, **LOOP** должна завершать цикл. В начале цикла в **CX** необходимо занести количество повторений. То есть, фрагмент может иметь такой вид:

```
MOV CX, LOOP_COUNT
BEGIN_LOOP:
<тело цикла>
LOOP BEGIN_LOOP
```

Осуществится столько повторений, какое число занесли в регистр **CX**. Ясно, что в теле цикла регистр **CX** дополнительно изменять **НЕЛЬЗЯ**.

Поскольку при реализации циклов проверяется содержание регистра **CX**, то для этого существует специальная команда условного перехода **JCXZ** (if CX is zero), когда содержание **CX** равняется нулю.

Что будет, когда в регистр **CX** попадет не какое-то число, а ноль? В этом случае будет 65536 повторений. Чтобы этого не случилось, можно включить специальную проверку **CX** на ноль:

```
MOV CX, LOOP_COUNT
JCXZ END_LOOP
BEGIN LOOP: <тело>
LOOP BEGIN_LOOP
END_LOOP: <продолжение>
```

Существует две разновидности команды **LOOP**, которые учитывают дополнительные условия.

**LOOPE/LOOPZ метка -** команда продолжается до тех пор, пока ***CX ≠ 0*** или пока ***ZF=1.***

Таким образом, эту команду удобно применять для поиска первого ненулевого элемента в массиве без перебора всего массива до конца.

**LOOPNE/LOOPNZ метка -** команда продолжает цикл пока ***CX ≠ 0*** или пока ***ZF = 0***.

Эта команда удобная для поиска первого нулевого элемента массива.

*Например:*

```
; начальный адрес массива в BX
; конечный адрес массива в DI
SUB DI, BX; разница адресов
INC DI; количество байт в массиве
MOV CX, DI; счетчик цикла
DEC BX
NEXT: INC BX; к следующему элементу
        CMP BYTE PTR [BX], 0; сравнить с нулем
        LOOPE NEXT ; если 0 - продолжаем
        JNZ NZ_FOUND; найден ненулевой элемент
NZ_FOUND:
```

В приведенном фрагменте использован псевдооператор **PTR**. Его назначение - локально отменять типы **DB, DW, DD** или атрибуты дистанции **NEAR, FAR**. Употребляется с атрибутами **BYTE, WORD** и т.п. Здесь он используется для того, чтобы адрес **BX** толковать, как адрес байта или слова.

Отметим, что команды цикла, как и другие команды условного перехода, действуют в границах -128, 127 байт. Если объем команд больший, то тогда нужно иначе организовать фрагмент с использованием команды **JMP**.

```
model small
.stack 100h
.data
mas   db 1,0,9,8,0,7,8,0,2,0
          db 1,0,9,8,0,7,8,0,2,0
          db 1,0,9,8,0,7,8,?,2,0
          db 1,0,9,8,0,7,6,?,3,0
          db 1,0,9,8,0,7,8,0,2,0
. code
start:
 mov ax,@data
 mov ds.ax
 xor ax, ax
 lea bx, mas
 mov cx, 5
 cycl_l:
  push cx
  xor si , si
  mov cx, 10
  cycl_2:
    cmp byte ptr [bx+si], 0
    jne no_zero
    mov byte ptr [bx+si], 0ffh
    no_zero:
    inc si
  loop cycl_2
 pop cx
 add bx, 10
 loop cycl_1
exit:
mov ax,4c00h
int 21h
end start
```

### 2.4.4. Команды вызова процедур

**CALL цель** - вызов процедуры или задачи

```
                    TITLE   CALLPROC (EXE) Вызов процедур
0000                STACKSG SEGMENT PARA STACK 'Stack'
0000    20 [ ???? ]         DW      32 DUP(?)
0040                STACKG  ENDS

0000                CODESG  SEGMENT PARA 'Code'
0000                BEGIN   PROC    FAR
                            ASSUME  CS:CODESG,SS:STACKSG
0000  1E                     PUSH    DS
0001  2B C0                  SUB     AX,AX
0003  50                     PUSH    AX
0004  E8 0008 R              CALL    B10     ;Вызвать B10
                    ;       ...
0007  CB                     RET             ;Завершить программу
0008                BEGIN   ENDP
                            ;--------------------------------------
0008                B10     PROC
0008  E8 000C R              CALL    C10     ;Вызвать C10
                    ;       ...
000B  C3                     RET             ;Вернуться в
000C                B10     ENDP            ; вызывающую программу
                    ;---------------------------------------------
000C                C10     PROC
                    ;       ...
000C  C3                     RET             ;Вернуться в
000D                C10     ENDP            ; вызывающую программу
                    ;---------------------------------------------
000D                CODESG  ENDS
                            END     BEGIN
```

# Лекция 10

## 2.5. Команды обработки строк (цепочки) символов

Под *строкой* символов здесь понимается последовательность байт, а *цепочка* — это более общее название для случаев, когда элементы последовательности имеют размер больше байта — слово или двойное слово.

Таким образом, цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательности элементов следующего размера:

- 8 бит — байт;
- 16 бит — слово;
- 32 бита — двойное слово.

Содержимое этих блоков для микропроцессора не имеет никакого значения. Это могут быть символы, числа и все что угодно. Главное, чтобы размерность элементов совпадала с одной из перечисленных и эти элементы находились в соседних ячейках памяти.

Предусмотрено **7** основных операций (примитивов):

1. пересылка элементов;
2. сравнение элементов;
3. сканирование элементов;
4. загрузка элементов;
5. хранение элементов;
6. получение элементов цепочки из порта ввода-вывода;
7. вывод элементов цепочки в порт ввода-вывода

### *2.5.1. Пересылка*

Команда **MOVS:**
**MOVS адрес_приемника, адрес_источника (MOVe String) —** переслать цепочку;
**MOVSB (MOVe String Byte) —** переслать цепочку байт;
**MOVSW (MOVe String Word)** — переслать цепочку слов;
**MOVSD (MOVe String Double word) —** переслать цепочку двойных слов.

Команда **MOVS** копирует байт, слово или двойное слово из цепочки, адресуемой операндом **адрес_источника**, в цепочку, адресуемую операндом **адрес_приемника**.

При трансляции в зависимости от типа операндов транслятор преобразует ее в одну из трех машинных команд: **MOVSB, MOVSW** или **MOVSD.**

Если перед командой написать префикс **REP**, то одной командой можно переслать до 64 Кбайт данных (если размер адреса в сегменте 16 бит — use16) или до 4 Гбайт данных (если размер адреса в сегменте 32 бит - use32).

***Алгоритм:***

1. Установить значение флага **DF** в зависимости от того, в каком направлении будут обрабатываться элементы цепочки — в направлении возрастания (**DF=0**) или убывания адресов ((**DF=1**));
2. Загрузить указатели на адреса цепочек в памяти в пары регистров **DS**:**(E)SI** и **ES: (E)DI;**

3. Загрузить в регистр **ECX/CX** количество элементов, подлежащих обработке;
4. Выдать команду **MOVS** с префиксом **REP.**

Следовательно, перед выполнением команд обработки строк нужно соответственно установить состояние флага **DF** с помощью команды:

**STD** (set direction flag) - DF = 1 - в направлении убывания адресов.

**CLD** (clear direction flag) (DF = 0) - в направлении возрастания адресов.

## ***Префиксы повторения***

Вместо команд **LOOP** здесь употребляются специальные префиксы повторения. Количество повторений предварительно записывается в регистр **CX**.

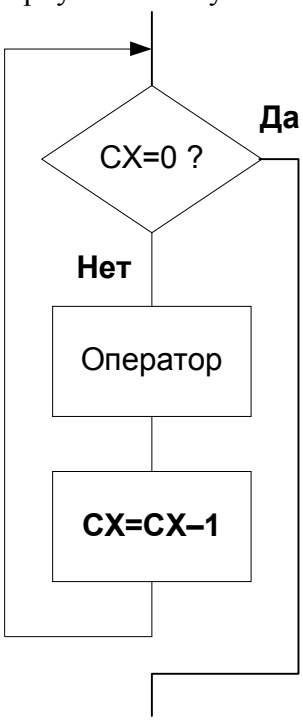
Например:

```
MOV CX, 50
REP MOVS DEST, SOURCE
```

**REP** используется перед строковыми командами и их краткими эквивалентами: **movs, stos, ins, outs**

***Алгоритм:***

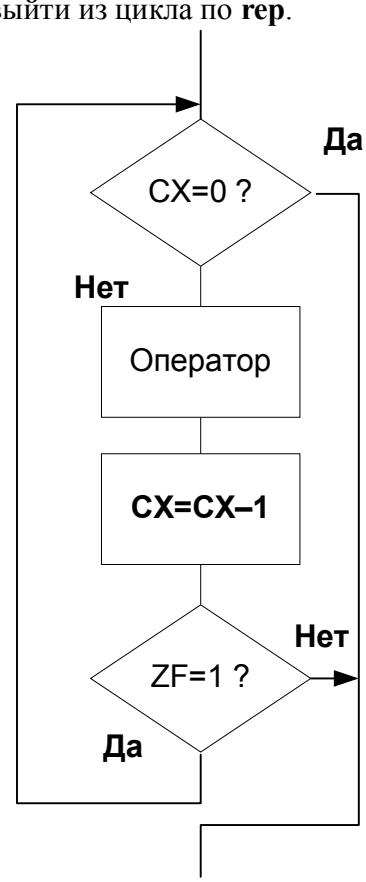
1. анализ содержимого **CX;**
2. если **CX<>0**, то выполнить строковую команду, следующую за данным префиксом и перейти к шагу 4;
3. если **CX=0**, то передать управление команде, следующей за данной строковой командой (выйти из цикла по **REP**);
4. уменьшить значение **CX=CX–1** и вернуться к шагу 1.




**REPE** и **REPZ** используются перед следующими цепочечными командами и их краткими эквивалентами: **cmps, scas**.

***Алгоритм:***

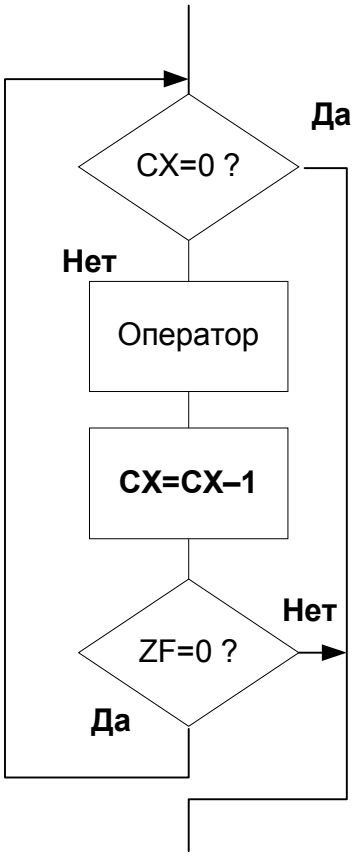
1. анализ содержимого **CX ;**
2. если **CX <>0** , то выполнить цепочечную команду, следующую за данным префиксом, и перейти к шагу 4;
3. если **CX =0** или **ZF=0**, то передать управление команде, следующей за данной цепочечной командой и перейти к шагу 6;
4. уменьшить значение **CX =CX -1**
5. если **ZF=1** вернуться к шагу 1
6. выйти из цикла по **rep**.




**REPNE** и **REPNZ** также имеют один код операции и имеют смысл при использовании перед эквивалентами: **cmps, scas**.

***Алгоритм:***

1. анализ содержимого **CX ;**
2. если **CX <>0** , то выполнить цепочечную команду, следующую за данным префиксом, и перейти к шагу 4;
3. если **CX =0** или **ZF=0**, то передать управление команде, следующей за данной цепочечной командой и перейти к шагу 6;
4. уменьшить значение **CX =CX -1**
5. если **ZF=0** вернуться к шагу 1
6. выйти из цикла по **rep**.




Фрагмент Пересылки строк командой **movs** будет иметь вид:

```
MASM
MODEL   small
STACK   256
.data
source  db      'Тестируемая строка','$'
;строка-источник
dest    db      19 DUP (' ')    ;строка-приёмник
.code
        assume  ds:@data,es:@data
main:   ;точка входа в программу
        mov     ax,@data        ; загрузка сегментных регистров
        mov     ds,ax   ; настройка регистров DS и ES       ;на адрес сегмента данных
        mov     es,ax
        cld     ; сброс флага DF — обработка строки от начала к концу
        lea     si,source       ;загрузка в si смещения строки-источника
        lea     di,dest ; загрузка в DS смещения строки-приёмника
        mov     cx,20   ; для префикса rep — счетчик повторений (длина строки)
rep     movs    dest, source    ;пересылка строки
        lea     dx, dest
        mov     ah,09h  ;вывод на экран строки-приёмника
        int     21h
exit:
        mov     ax,4c00h ; тоже самое, что и RET
        int     21h
end     main
```

### *2.5.2. Сравнения строк*

**CMPS адрес_приемника, адрес_источника** - сравнивает байты или слова
**CMPSB**
**CMPSW**
**CMPSD**

**Адрес_источника** находится в сегменте данных и адресуется с помощью регистров **DS** и **SI**. **Адрес_приемника** - в дополнительном сегменте и адресуется с помощью регистров **ES** и **DI.**

Выполняется как вычитание, но в отличие от **CMP**, ***от источника отнимается приемник***. Это необходимо учитывать для следующих команд условного перехода.

***Алгоритм:***

1. Загрузить адрес источника — в пару регистров **DS:ESI/SI**;
2. Загрузить адрес назначения — в пару регистров **ES:EDI/DI**;
3. Выполнить вычитание элементов (источник – приемник);
4. В зависимости от состояния флага **DF** изменить значение регистров **ESI/SI** и **EDI/DI**:
5. Если **DF**=0, то увеличить содержимое этих регистров на длину элемента последовательности;
6. Если **DF**=1, то уменьшить содержимое этих регистров на длину элемента последовательности;
7. В зависимости от результата вычитания установить флаги:
8. Если очередные элементы цепочек не равны, то **CF**=1, **ZF**=0;
9. Если очередные элементы цепочек или цепочки в целом равны, то **CF**=0, **ZF**=1;
10. При наличии префикса выполнить определяемые им действия (см. команды REPE/REPNE).

Чаще с данной командой для повторения употребляется префикс **REPE** (до первого отличия) или **REPNE** (до первого совпадания).

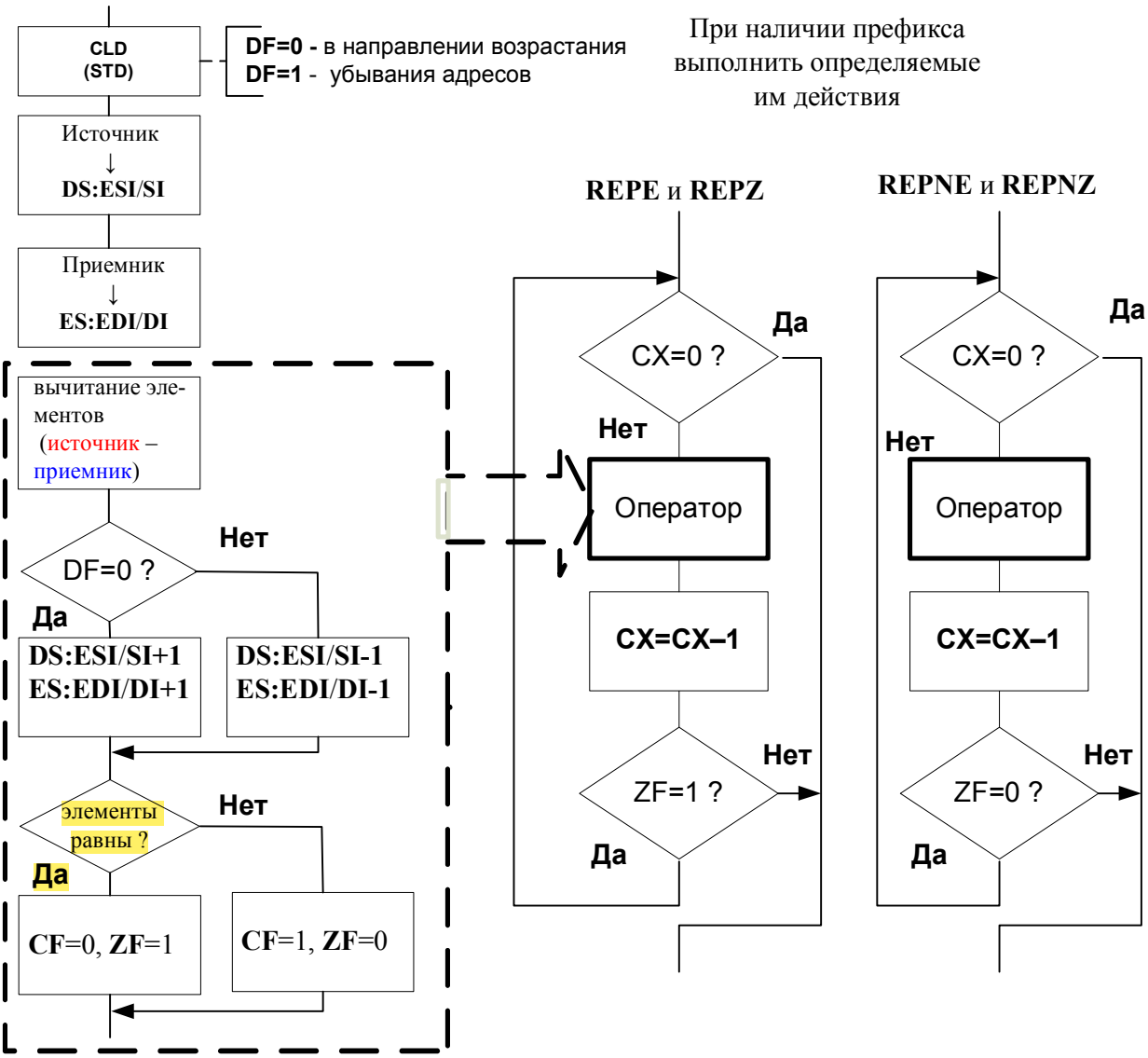
```
CLD
MOV CX,50
REPE CMPS DEST, SOURCE; до первого отличия
```

Выход из цикла происходит по двум причинам - массив пересмотрен полностью, или есть различия. Поэтому после **CMPS** необходимо реагировать на соответствующую причину, используя команды условного перехода.

Следовательно, после приведенного фрагмента нужно поставить переход на метку:

```
JNE FOUND; отличие найдено
……
FOUND:      ; да,.продлить обработку
```

Команда **CMPS** превращается транслятором: на **CMPSB** или на **CMPSW**, аналогично команде **MOVS**.




Например, можно определить число разных пар в строках:

```
MOV AX, 0; количество различных пар
CLD
LEA SI, SOURCE
LEA DI, ES:DEST
MOV CX, N
COMP: REPE CMPSB
JE FIN; переход на FIN, если строки равны
INC AX; следующая несовпадающая пара
CMP CX, 0; остались символы?
JNE COMP; продолжить, если строки не пусты
FIN:
```

### *2.5.3. Сканирования строк*

**SCAS адрес_приемника** - позволяет найти значение (байт или слово) в строке-приемнике, который находится в дополнительном сегменте, начало которого фиксирует регистр ES.

**SCASB**
**SCASW**
**SCASD**

При этом заданное значение должно находиться в регистре **AX** (при поиске слова) или в **AL** (при поиске символа).

То есть, это фактически команды сравнения с содержанием аккумулятора.

Например, найти первую точку и заменить на '*':

```
CLD
LEA DI, ES: STRING
MOV AL, '.'
MOV CX, 50
REPNE SCAS STRING ; или SCAS B
JNE FIN ; точки нет
MOV BYTE PTR ES:[DI -1], '*'
FIN:
```

Это будет поиск в строке STRING первой точки. При выходе из цикла в регистре **DI** будет адрес следующего за точкой '.' байта.

### *2.5.4. Загрузка строки*

После поиска слова или символа с помощью сканирования с ним что-то нужно сделать.

**LODS адрес_источника -** команда загрузки строки

**LODSB**
**LODSW**
**LODSD**

Пересылается операнд строка_источник, который адресован регистром **SI** из сегмента данных в регистр **AL** (при пересылке байта) или в регистр **AX** (пересылка слова), а затем изменяет регистр **SI** так, чтобы он показывал на следующий элемент строки. То есть, регистр **SI** увеличивается на 1 или 2, когда **DF** = 0 и уменьшается, когда **DF** = 1.

Следовательно, команды **LODS** эквивалентные двум:

```
MOV AL, [SI]
INC SI
```

Пример:

```
CLD
LEA DI, ES DEST ; смещение адреса DEST
LEA SI, SOURCE ; смещение адреса SOURCE
MOV CX, 100
REPE CMPSB; проверка отмены
        JCXZ MATCH; несовпадения нет?
        DEC SI; скорректировать SI
        LODS SOURCE; загрузить отличный элемент в регистр AL
MATCH; несовпадения нет
```

### *2.5.5. Команда сохранения строки*

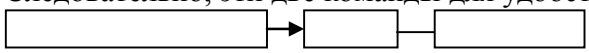
**STOS адрес_приемника** - противоположная к **LODS,** пересылает байт из **AL** или слово с **AX** в строку-приемник, которая находится в дополнительном сегменте и адресуется регистром **DI**. После этого изменяет регистр **DI** так, что он показывает на следующий элемент строки.

**STOSB**
**STOSW**
**STOSD**

Командой удобно заполнять строки определенными значениями. Например, в строке MY_STRING осуществляется поиск среди 100 слов первого ненулевого элемента. Если такой элемент найден, то следующие за ним 5 слов заполняются нулями.

```
CLD
LEA DI, ES: MY_STRING ; адрес строки
MOV AX, 0;
MOV CX, 100  ; счетчик поиска
REPNE SCASW ; сканирование строки
JCXZ NOT_SCAN  ; найдено ненулевое слово?
MOV CX, 5  ; да!
REP STOS MY_STRING
NOT SCAN
```

Следовательно, эти две команды для удобства, когда нашли что-то, то:




### 2.5.6. Получение элементов цепочки из порта ввода-вывода

**INS адрес_приемника, номер_порта** - ввести элементы из порта ввода-вывода в цепочку.

**INSB**
**INSW**
**INSD**

Эта команда вводит элемент из порта, номер которого находится в регистре **dx**, в элемент цепочки, адрес которого определяется операндом **адрес_приемника**.

Пример: Введем 10 байт из порта 5000h в область памяти pole.

```
.data
pole    db      10 dup (‘ ‘)
.code
...
        push    ds
        pop     es      ;настройка es на ds
        mov     dx, 5000h
        lea     di, pole
        mov     cx,10
rep     insb
...
```

### *2.5.7. Вывод элемента цепочки в порт ввода-вывода*

**OUTS номер_порта, адрес_источника (Output String) —** вывести элементы из цепочки в порт ввода-вывода.
**OUTSB**
**OUTSW**
**OUTSD**

Эта команда выводит элемент цепочки в порт, номер которого находится в регистре **dx**. Адрес элемента цепочки определяется операндом **адрес_источника**. Несмотря на то, что цепочка, из которой выводится элемент, адресуется указанием этого операнда, значение адреса должно быть явно сформировано в паре регистров **ds:esi/si**.

В качестве примера рассмотрим фрагмент программы, которая выводит последовательность символов в порт ввода-вывода, соответствующего принтеру (**номер 378 (lpt1)).**

```
.data
str_pech        db      'Текст для печати'
.code
...
        mov     dx,378h
        lea     di,str_pech
        mov     cx,16
rep     outsb
...
```

### *Команды загрузки адресных пар в регистры*

Использование строчных команд нуждается в определенном количестве установочных команд. Их можно сократить с помощью двух команд, которые устанавливают пары регистров **DS:SI** и **ES:SI** на обработку строк.

Это команды: **LDS, LES, LFS, LGS, LSS**

**LDS приемник, источник** - получение полного указателя в виде сегментной составляющей и смещения.

**Приемник** - имя регистра, а **источник** - память - адрес двойного слова памяти, которое задает абсолютный адрес. Команда записывает в регистр смещения **ofs**, а в регистр **DS** - номер этого сегмента.

*Алгоритм:*

Алгоритм работы команды зависит от действующего режима адресации (use16 или use32):

- если **use16**, то загрузить первые ***два*** байта из ячейки памяти источник в 16-разрядный регистр, указанный операндом **приемник**. Следующие два байта в области **источник** должны содержать сегментную составляющую некоторого адреса; они загружаются в регистр **DS/ES/FS/GS/SS –** в зависимости от команды;
- если use32, то загрузить первые ***четыре*** байта из ячейки памяти источник в 32-разрядный регистр, указанный операндом **приемник**. Следующие два байта в области источник должны содержать сегментную составляющую, или селектор, некоторого адреса; они загружаются в регистр **DS/ES/FS/GS/SS**.

Пример:

```
ADR DD ALPHA; [ADR] → ofs [ADR + 2]:SEG
LDS SI, ADR;  SI - ofs, DS → SEG
```

Вторая команда:

**LES приемник, источник -** (Load pointer using **ES**) аналогичная к первой, только номер сегмента загружается в регистра **ES**.

Например:

```
DATA1 SEGMENT
        S1 DB 200DUP(?)
        AS DD S2
DATA1 ENDS
DATA2 SEGMENT
        S2 DB 200 DUP(?)
DATA2 ENDS
; копировать массив S1 в S2
CODE SEGMENT
         ASSUME CS: CODE, DS:DATA1
; пусть в этот момент DS = DATA1
        CLD
         LEA SI, S1 ; DS:SI = начало S1
         LES DI, AS ; ES:DI = начало S2
        MOV CX, 200
        REP MOVSB; копировать S1 в S2
```

### *2.6. Команды управления микропроцессором*

1. команды управления флагами;
2. команды внешней синхронизации;

3. команды холостого хода.

### *2.6.1. Команды управления флагами*

Позволяют влиять на состояние трех флагов регистра флагов: **CF** (carry flag) - переноса, **DF** (direction) – направления и прерывания **IF** (interupt F).

Структура команд очень простая: установить бит в единицу (Set), сбросить в 0 (Clear). Соответствующие две буквы является начальными для команды. Третья буква определяет, какой именно бит нужно обработать. Поэтому, имеем 6 команд:

| | | |
|---|---|---|
| **STC** | **STD** | **STI** |
| **CLC** | **CLD** | **CLI** |

И седьмая команда - изменить значение биту переноса на противоположное:

**CMC** (CoMpliment Carr Flag).

Команды управления битом Carry употребляются перед выполнением команд циклического сдвига с переносом **RCR** (**RLC)** - операция циклического сдвига операнда вправо (влево) через флаг переноса **CF** .

Команды управления битом направления, как видно, используются перед командами обработки строк и задают направление модификации индексных регистров **DI** и **SI** (**DF** = 0 - в сторону увеличения, **DF** = 1 - в сторону уменьшения).

Команды управления битом **IF** используются, например, при обработке маскируемых прерываний. Если во время работы программы никакое маскируемое прерывание не позволяется, то нужно установить **IF** в 0. Напомним, что при этом немаскированные прерывания позволяются.

### *2.6.2.* Команды внешней синхронизации

Используются для синхронизации работы МП с внешними событиями.

Команда **HLT** (halt -- остановить) - останавливает работу микропроцессора, он переходит на холостой ход и не выполняет никаких команд. Из этого состояния его можно вывести сигналами на входах **RESET**, **NMI**, **INTR**. Если для возобновления работы микропроцессора используется прерывание, то сохраненное значение пары **cs:eip/ip** указывает на команду, следующую за **hlt**. В микропроцессоре не предусмотрено специальных средств для подобного переключения. Сброс микропроцессора можно инициировать, если вывести байт со значением **0feh** в порт клавиатуры **64h**. После этого микропроцессор переходит в реальный режим и управление получает программа **BIOS**, которая анализирует байт отключения в **CMOS**-памяти по адресу **0fh**. Для нас интерес представляют два значения этого байта — **5h** и **0ah**:

**5h** — сброс микропроцессора инициирует инициализацию программируемого контроллера прерываний на значение базового вектора **08h**. Далее управление передается по адресу, который находится в ячейке области данных **BIOS 0040:0067**;

**0ah** — сброс микропроцессора инициирует непосредственно передачу управления по адресу в ячейке области данных **BIOS 0040:0067** (то есть без перепрограммирования контроллера прерываний).

Таким образом, если вы не используете прерываний, то достаточно установить байт **0fh** в **CMOS**-памяти в **0ah**. Предварительно, конечно, вы должны инициализировать ячейку области данных **BIOS 0040:0067** значением адреса, по которому необходимо передать управление после сброса. Для программирования **CMOS**-памяти используются номера портов **070h** и **071h**. Вначале в порт **070h** заносится нужный номер ячейки **CMOS**-памяти, а затем в порт **071h** — новое значение этой ячейки.

Если прерывания заблокированы во время останова, ЭВМ полностью "замирает". В этой ситуации единственная возможность запустить ЭВМ заново - выключить питание и включить его снова. Однако, если прерывания были разрешены в момент останова микропроцессора, они продолжают восприниматься и управление будет передаваться обработчику прерываний. После выполнения команды **IRET** в обработчике программа продолжает выполнение с ячейки, следующей за командой **HLT**. Команду **HLT** можно использовать в мультизадачных системах, чтобы завершить текущую активную задачу, но это не всегда лучший способ такого завершения. Разработчики персональной ЭВМ используют команду останова только тогда, когда возникает катастрофическая ошибка оборудования и дальнейшая работа бессмысленна.

Пример:

```
;работаем в реальном режиме, готовимся к переходу в защищенный режим:
      push   es
      mov    ax,40h
      mov    es,ax
      mov    word ptr es:[67h],offset ret_real
;ret_real — метка в программе, с которой должно
;начаться выполнение программы после сброса
      mov    es:[69h],cs
      mov    al,0fh  ;будем обращаться к ячейке 0fh в CMOS
      out    70h,al
      jmp    $+2     ;чуть задержимся, чтобы аппаратура отработала
;сброс без перепрограммирования контроллера
      mov    al,0ah
      out    71h,al
;переходим в защищенный режим установкой бита 0 cr0 в 1
;работаем в защищенном режиме готовимся перейти обратно в реальный режим
      mov    al,01fch
      out    64h,al  ;сброс микропроцессора  hlt
;остановка до физического окончания процесса сброса
      ret_real:      ...     ;метка, на которую будет передано управление после сброса
```

После этого МП переходит выполнять команду, следующую за **HLT**.

Команда **WAIT** (wait -- ожидать) - переводит МП на холостой ход, но при этом через каждые 5 тактов проверяется активность входной линии **TEST**. Если этот вывод активен во время выполнения **WAIT**, то остановка **НЕ** происходит.

Команда **ECS** (escape -- побег) - обеспечивает передачу команды МП 88/86 внешним процессорам, например, арифметическому сопроцессору 8087. Сам же МП 88 ничего не делает, только читает какие-то данные и отбрасывает.

Команды **WAIT** и **ESC** используются для работы с сопроцессором 8087.

Префикс **LOCK** - это командный префикс, подобно **REP**-префиксу, он предназначен для работы в мультипроцессорных системах, когда несколько процессоров работают с одним и тем же участком памяти. Префикс **LOCK** принуждает захватить линии управления и тем самым получить исключительное право доступа к памяти на время обработки команды с префиксом.

```
MOV AL,1
LOCK XCHG AL,FLAG_BYTE
CMP AL,1
```

В общей области FLAG_BYTE установлено 1, если в ней работает другой МП, и 0, если никакой МП не работает. Когда там есть 1, то МП будет ожидать, пока область не освободится.

Пустая команда **NOP** - нечего не выполняется 2 такта, реализуется как **XCHG AX, AX**.

Эту команду можно использовать:

- в программах реального времени, когда нужен выдержать точную длительность фрагментов.
- при налаживании программы, когда отдельные фрагменты могут изменяться.

Например:

```
JZ ONE
....
ONE: MOV AX, SUM[BX]
```

Лучше:

```
JZ ONE
.....
ONE: NOP
MOV AX, SUM[BX]
```

# Лекция 11

# Раздел 3. Процедуры и макрокоманды

## 3.1 Процедуры и особенности их использования

Процедурой (или подпрограммой) называется фрагмент программы, к которому можно перейти, и из которого можно вернуться туда, откуда осуществляется переход.

Переход к процедуре называется вызовом, а переход назад - возвращением. Возвращение осуществляется к команде, следующей после вызова.

Эти функции выполняют две команды:

CALL (call a procedure) и

RET (return from procedure - вернуться из процедуры).

Команда CALL имеет формат:

**CALL имя**

где **имя** - это имя процедуры, которая вызывается.

Сама же процедура, как отмечалось, имеет вид:

```
NAME PROC
……
RET
NAME ENDP
```

Имя процедуры считается меткой, которая принадлежит к первой команде процедуры. Поэтому имя процедуры можно отмечать в командах перехода, который означает переход на первую команду процедуры.

В отличие от языков высокого уровня, имена в процедурах (переменные, метки) не локализуются внутри их, следовательно должны быть уникальными для соответствующего сегмента.

Где размещать процедуру? Где угодно. Но так, чтобы процедура (подпрограмма) не выполнялась, если к ней не обращаются. Поэтому можно рекомендовать 3 варианта:

1. В одном сегменте после основной программы;
2. В одном сегменте перед точкой входа;
3. В отдельном сегменте.

Тогда можно в конце основной программы поставить команду FINISH.

```
A SEGMENT                B SEGMENT              C SEGMENT
BEG: <осн. прогр.>
FINISH                   Процедура              Процедура

Процедура                BEG:                     C ENDS
                         <осн. прогр.>          D SEGMENT
                                                  BEG:<осн. прогр.>
A ENDS                   B ENDS                 D ENDS
END BEG                  END BEG                END BEG
```

Если процедура предназначена для использования и в других сегментах, то она имеет атрибут дистанции **FAR**. Главная процедура также должна иметь атрибут **FAR**. По умолчанию процедура имеет атрибут **NEAR**.

При выполнении команды **CALL *NAME*** сначала нужно занести адрес возвращения стек, а затем сделать переход на имя ***NAME***.

Для **NEAR**-процедуры в стек достаточно занести лишь адрес смещения.

Следовательно, в этом случае команда **CALL NAME** *эквивалентна*

```
PUSH IP
JMP NAME
```

Для **FAR**-процедуры адрес возвращения является абсолютным адресом и состоит из двух слов **CS:IP**. Поэтому команда **CALL *NAME*** будет эквивалентна *трем командам*:

```
PUSH CS
PUSH IP
JMP NAME
```

Другими словами:

1. Сначала в стек заносится адрес начала сегмента кода, который содержится в регистре **CS**
2. Затем заносится содержание регистра указателя команд **IP** - смещение.

Команда **RET** возвращает управление к точке вызова процедуры. Эквивалентна двум командам:

```
POP IP
POP CS
```

Если процедура имеет атрибут **NEAR** - выполняется только первая команда. То есть, особенности выполнения команды **RET** определяются заглавием процедуры - наличием аттрибута **NEAR** или **FAR**.

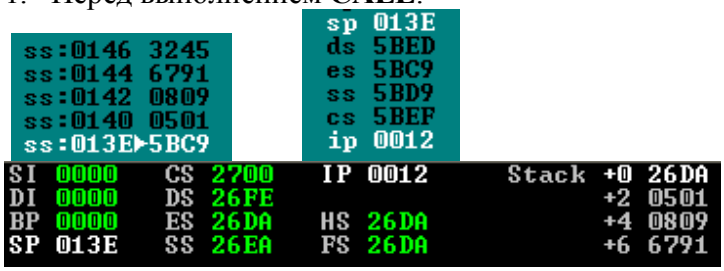
Пример: Пусть, первый фрагмент программы транслирован по таким адресам:

```
0012  E8 0007        CALL MY_PROC; вызов процедуры
0015  8B C3          MOV AX, BX; вернуться сюда из процедуры
0017  B8 4C00        MOV AX,4c00h
001A  CD 21          INT 21h
001C                 MAIN ENDP
                     ; описание процедуры
001C                 MY_PROC PROC; начало процедуры
001C  B1 0A           MOV CL, 10; первая команда процедуры
001E  C3              RET; вернуться к точке вызова
001F                  MY_PROC ENDP
```

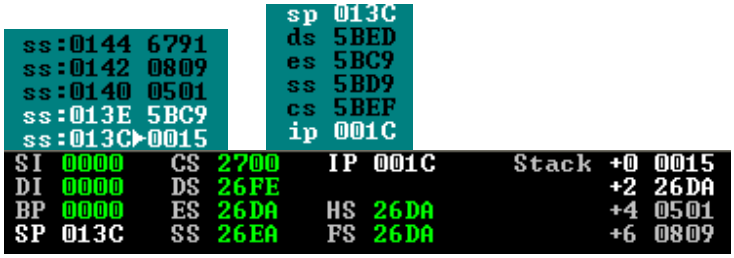
По умолчанию MY_PROC имеет атрибут **NEAR**.

Влияние процедуры на стек:

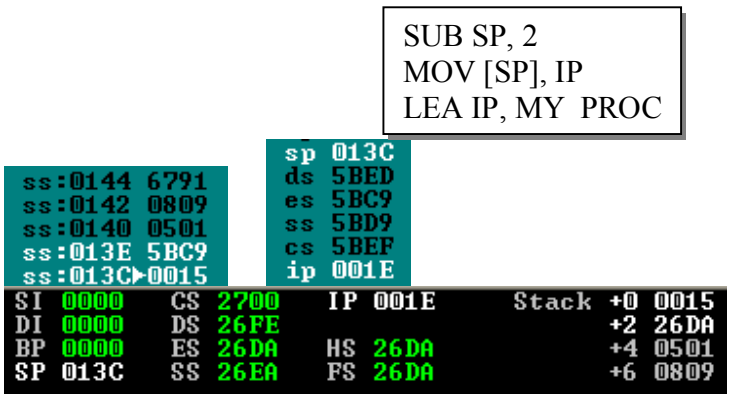
1. Перед выполнением **CALL**:



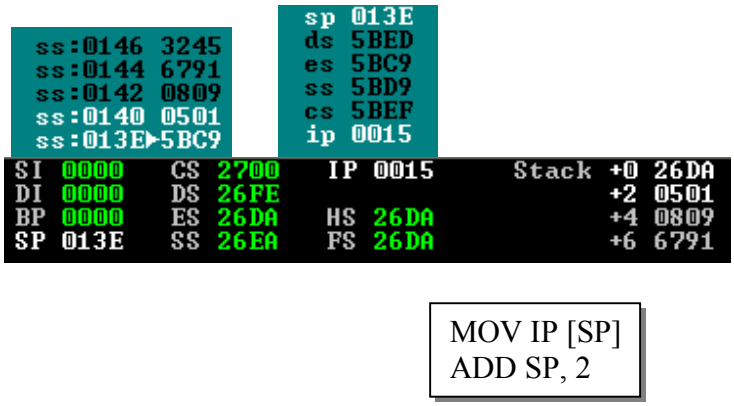

После выполнения CALL:




Перед выполнением **RET**:




После выполнения **RET**:




Продлится выполнение, начиная с команды NEXT:

Поскольку значение IP изменилось, то начнется выполнение процедуры, начиная с команды MOV CL, 10 и до команды **RET**.

Процедура может размещаться в том же сегменте, откуда осуществляются к ней обращения, или в другом сегменте. В первом случае атрибутом процедуры может быть **NEAR** и обращение также типа **NEAR**. Во втором случае атрибутом процедуры будет FAR и обращение также будет **FAR**. Но если процедура все-таки имеет атрибут FAR, то обращение к ней также должно иметь атрибут **FAR**.

Например:

```
SEGX SEGMENT
.........
SUBT PROC FAR
.........
RET
SUBT ENDP
.........
CALL FAR PTR SUBT
...........
SEGX ENDS
SEGY SEGMENT
.........
CALL FAR PTR SUBT
...........
SEGY ENDS
```

То есть, в обращении к процедуре могут быть атрибуты **FAR PTR** или **NEAR PTR**. В данном случае процедура определена перед ее вызовом, то есть, осуществляется вызов “назад”. Поэтому если не отметить **FAR PTR**, то ассемблер все равно правильно реализует и вызовы и возвращения.

В данном случае их можно оставить для ясности. Но когда процедура реализуется после вызова, то есть, осуществляется вызов “вперед”, то эти атрибуты обязательны.

Однако таких действий маловато. Для вызова процедуры из сегмента SEGY в нем обязательно нужно объяснить, что имя SUBT является внешним. То есть, включить директиву EXTRN SUBT: FAR.

В сегменте SEGX нужно объяснить, что к имени SUBT может быть доступ извне, то есть, нужно включить директиву Public SUBT.

Отметим, что имя SUBT находится в том же сегменте, что и директива Public. Поэтому в ней отмечать тип имени не нужно (транслятор разберется). В директиве EXTRN отметить тип нужно. По аналогии с командой **JMP** в процедуре возможны обращения:

1. прямые;
2. непрямые;
3. близкие;
4. далекие.

Не позволяются короткие **SHORT**, потому что считается, что процедура не располагается рядом с командой вызова. Поэтому, возможны такие варианты вызова:

```
CALL P; близкий переход
CALL FAR PTR Q; далекий переход
CALL Q; близкий переход
CALL NEA; близкий непрямой вызов
CALL FA; далекий непрямой вызов
LEA BX,Q
CALL [BX]; близкий вызов процедуры Q
CALL DWORD PTR [BX]; далекий непрямой вызов
Пример:
NEAR DW P
FA DD Q
P PROC
P1: ....
P ENDP
Q PROC FAR
Q1: .....
Q ENDP
```

# Лекция 12

## 3.2. Передача параметров в процедуру

Как видим, аналогии с передачей параметров в процедуру как в языках высокого уровня нет. Следовательно, нет списка параметров. Поэтому программист сам должен позаботиться о такой передаче как в процедуру, так и из нее.

Можно использовать несколько вариантов:

1. Через регистры;
2. Через стек;
3. Через общие области памяти

### *3.2.1. Передача параметров через регистры*

Основная программа размещает параметры в определенных регистрах, а процедура их оттуда берет. Процедура записывает результаты в регистры, а программа их потом считывает.

Например:

```
определить c = max(a,b); a запишем в AX, b - в BX, а результат c - в AX.
основная программа
MOV AX, A
MOV BX, B
CALL MAX
MOV C, AX; защита AX
…….
MAX PROC FAR
CMP AX, BX
JGE MAX1
MOV AX, BX
MAX1: RET
MAX ENDP
```

***Защита регистров в процедурах***

Регистров в МП мало, поэтому процедура и программа используют одни и те же регистры. Чтобы процедура не изменяла содержание регистров в главной программе, нужно их защитить. То есть, переслать содержание регистров в стек, а затем обновить в обратном порядке. Это рекомендуется делать даже тогда, когда программа и процедура используют разные регистры. Со временем основная программа может измениться, и тогда процедура будет “мешать”.

Для облегчения таких действий в МП, начиная с 80186, внедрены команды **PUSHA** и **POPA**, которые записывают и считывают из стека содержание сразу 8 регистров: **AX, BX, CX, DX, DI, SI, SP, BP**. Также разрешено пересылать константы в стек - **PUSH 125**.

***Передача параметров сложных типов***

Кроме параметров значений, в процедуру можно передавать адреса. Когда параметром является сложный тип - структура или массив, то на машинном уровне передают не сами данные, а начальный адрес. Зная количество элементов и начальный адрес или поля, можно получить доступ к каждому элементу.

Например:

```
для двух массивов чисел без знака
X DB 100 DUP(?)
Y DB 50 DUP(?)
Определить  max (X[i])+max(Y[i]).
```

Ясно, что лучше сделать процедуру определения максимального элемента. У нее будет два параметра: начальный адрес и количество элементов.

```
; процедура MAX: AL = макс. элемент

MAX PROC
        PUSH BX ; начальный адрес
        PUSH CX
         MOV AL,0
MAX1: CMP [BX],AL
            JLE MAX2
            MOV AL[BX]
MAX2: INC BX; к следующему эл-ту
           LOOP MAX1
            POP CX
            POP BX
            RET
MAX ENDP
; Основная программа
LEA BX,X
MOV CX,100
CALL MAX
MOV DL,AL; защита AL
LEA BX,Y
```

***Защита регистров***

Если нет гарантии, что процедура может изменить содержание регистров, которые должны сохраниться после выхода из процедуры такими, как непосредственно перед вызовом процедуры, то их нужно на время выполнения заслать в стек.

Например:

```
все регистры общего назначения:
PUSH AX
PUSH BX
PUSH CX
PUSH DX
```

Перед выходом из процедуры - командой RET нужно эти регистры обновить:

```
POP DX
POP CX
POP BX
POP AX
```

Это делаем в обратном порядке. Это же касается индексных регистров и младшего байта регистра флагов.

### *3.2.2. Передача параметров через стек*

Если в процедуре много параметров, то может не хватить регистров. Поэтому в этом случае можно использовать стек. Перед вызовом процедуры записываем параметры в стек в порядке, который определяет программист.

При этом применяется своеобразная методика работы со стеком не с помощью команд **push** и **pop**, а с помощью команд **mov** с косвенной адресацией через регистр **BP**, который архитектурно предназначен именно для адресации в стеке.

Пример:

Нужно, чтобы параметр (условная величина задержки) передавался в подпрограмму через стек. Вызов подпрограммы delay выполняется следующим образом:

```
0001  50 55 8B EC C7 46 02+  push 2000 ;Проталкиваем в стек значение параметра 07D0 5D
000B  E8 0005          call delay   ;Вызываем подпрограмму delay
000E  B8 4C00          mov ax,4c00h
0011  CD 21            int 21h
0013                   MAIN ENDP
0013                   delay proc          ;Процедура-подпрограмма
0013  51               push CX             ;Сохраним CX      основной программы
0014  55               push BP             ;Сохраним BP
0015  8B EC            mov BP,SP           ;Настроим BP      на текущую вершину стека
0017  8B 4E 06         mov CX, [BP+6]      ;Скопируем из стека параметр
001A  51               del1:  push CX      ;Сохраним его
001B  B9 0000          mov CX,0            ;Счетчик внутреннего      цикла
001E  E2 FE            del2:  loop del2    ;Внутренний цикл(64К      шагов)
0020  59               pop CX              ;Восстановим      внешний      счетчик
0021  E2 F7            loop del1           ;Внешний цикл
0023  5D               pop BP              ;Восстановим      BP
0024  59               pop CX              ;и CX программы
0025  C2 0002          ret  2              ;Возврат и снятие со стека ненужного параметра
0028                   delay ENDP
```

Команда **call**, передавая управление подпрограмме, сохраняет в стеке адрес возврата в основную программу. Подпрограмма сохраняет в стеке еще два 16-разрядных регистра. В результате стек оказывается в состоянии:


После сохранения в стеке исходного содержимого регистра **BP** (в основной программе нашего примера этот регистр не используется, однако в общем случае это может быть и не так), в регистр **BP** копируется содержимое указателя стека, после чего в **BP** оказывается смещение вершины стека. Далее командой mov в регистр **CX** заносится содержимое ячейки стека, на 6 байтов ниже текущей вершины. В этом месте стека как раз находится передаваемый в подпрограмму параметр, как это показано в левом столбце рис.

```
CPU 80486                                                    1
#lec12ed#29: push BP ;сохраним BP                    ax 0E35   c=0
  cs:0014 55              push   bp                  bx 2DB3   z=0
#lec12ed#30: mov BP,SP ;Настроим BP на текущую вершину стека  cx 2DB3   s=0
  cs:0015▶8BEC            mov    bp,sp               dx 24C4   o=0
#lec12ed#31: mov CX, [BP+6] ;Скопируем из стека параметр 2000 si 3092   p=0
  cs:0017 8B4E06          mov    cx,[bp+06]          di 3092   a=0
#lec12ed#del1: del1: push CX ;Сохраним его           bp 0100   i=1
  cs:001A 51              push   cx                  sp 0136   d=0
#lec12ed#33: mov CX,0 ;счетчик внутреннего цикла     ds 5BCE
  cs:001B B90000          mov    cx,0000             es 5BCE
#lec12ed#del2: del2: loop del2 ;внутренний цикл(64k шагов - 6 ss 5BDE
  cs:001E E2FE            loop   #lec12ed#del2 (001E) cs 5BF4
#lec12ed#35: pop CX ;восстановим внешний счетчик     ip 0015
  cs:0020 59              pop    cx
#lec12ed#36: loop del1 ;внешний цикл CX=CX-1

  ds:0000 CD 20 FF 9F 00 9A F0 FE =  Я ъЁ▮           ss:013E 5BCE
  ds:0008 1D F0 E0 01 68 22 AA 01 ↔Ёр☺h"к☺          ss:013C 07D0
  ds:0010 68 22 89 02 C3 1C 35 0E h"Й☻├∟5♫          ss:013A 000E
  ds:0018 01 01 01 00 02 FF FF FF ☺☺☺ ☻              ss:0138 2DB3
  ds:0020 FF FF FF FF FF FF FF FF                    ss:0136▶0100
```

Конкретную величину смещения относительно вершины стека надо для каждой подпрограммы определять индивидуально, исходя из того, сколько слов сохранено ею в стеке к этому моменту. Напомним, что при использовании косвенной адресации с регистром **BP** в качестве базового, по умолчанию адресуется стек, что в данном случае и требуется.

Выполнив возложенную на нее задачу, подпрограмма восстанавливает сохраненные ранее регистры и осуществляет возврат в основную программу с помощью команды **ret**, в качестве аргумента которой указывается число байтов, занимаемых в стеке отправленными туда перед вызовом подпрограммы параметрами. В нашем случае единственный параметр занимает 2 байта. Если здесь использовать обычную команду **ret** без аргумента, то после возврата в основную программу параметр останется в стеке, и его надо будет оттуда извлекать (между прочим, не очень понятно, куда именно, ведь все регистры у нас могут быть заняты). Команда же с аргументом, осуществив возврат в вызывающую программу, увеличивает содержимое указателя стека на значение ее аргумента, тем самым осуществляя логическое снятие параметра. Физически этот параметр, как, впрочем, и все остальные данные, помещенные в стек, остается в стеке и будет затерт при дальнейших обращениях к стеку.

Разумеется, в стек можно было поместить не один, а сколько угодно параметров. Тогда для их чтения надо было использовать несколько команд **mov** со значениями смещения **BP+6, BP+8, BP+0Ah** и т.д.(см. видео приложение к лекции)

Рассмотренная методика может быть использована и при дальних вызовах подпрограмм, но в этом случае необходимо учитывать, что дальняя команда **call** сохраняет в стеке не одно, а два слова, что повлияет на величину рассчитываемого смещения относительно вершины стека.

### *3.2.3. Передача параметров через общие области памяти*

Остановимся на последнем варианте. Если процедура и вызов к ней размещены в разных сегментах кода, то данные можно разместить в самом начале и использовать их имена с директивами **PUBLIC** и **EXTRN**.

Директива **EXTRN** предназначена для объявления некоторого имени внешним по отношению к данному модулю. Это имя в другом модуле должно быть объявлено в директиве **PUBLIC**.

Директива **PUBLIC** предназначена для объявления некоторого имени, определенного в этом модуле и видимого в других модулях.

Синтаксис этих директив следующий:

**extrn имя:тип, имя:тип**

**public имя,...,имя**

Здесь **имя** — идентификатор, определенный в другом модуле. В качестве идентификатора могут выступать:

- имена переменных, определенных директивами типа **DB**, **DW** и т. д.;
- имена процедур;
- имена констант, определенных операторами = и **EQU**.

Аргумент тип определяет тип идентификатора. Указание типа необходимо для того, чтобы транслятор правильно сформировал соответствующую машинную команду. Действительные адреса вычисляются на этапе редактирования, когда будут разрешаться внешние ссылки.

Возможные значения типа определяются допустимыми типами объектов для этих директив:

- имя переменной - тип может принимать значения **DB**, **DW** и т. д.;
- имя процедуры - тип может принимать значения **near** или **far**;
- имя константы - тип должен быть **abs**.

Покажем принцип использования директив **EXTRN** и **PUBLIC** на схеме связи модулей 1 и 2.

```
;Модуль 1
masm
.model small
.stack 256
.data
.code
my_proc_1 proc
my_proc_1 endp
my_proc_2 proc
my_proc_2 endp
public my_proc_1 ;объявляем процедуру my_proc_1 видимой извне

start:
mov ax,@data
end start
```

```
;Модуль 2
masm
.model small
.stack 256
.data
. code
extrn my_proc_1 ;объявляем процедуру roy_proc_1 внешней
start:
mov ax,@data
call my_proc_1 ;вызов my_proc_1 из модуля 1
end start
```

---

Если в процедуре, описанном в одном сегменте есть описание переменных, но в главной программе, где осуществляется вызов процедуры, можно использовать и другие имена этих элементов данных.

Например:

```
добавление элементов массива:
DATA SEGMENT COMMON
ARY DW 100 DUP(?)
COUNT DW ?
SUM DW 0
DATA ENDS
```

Например:

NUM вместо ARY,
N вместо COUNT и
TOTAL вместо SUM.

Для этого этот сегмент нужно определить так:

```
DATA SEGMENT COMMON
NUM DW 100 DUP(?)
N DW ?
TOTAL DW 0
DATA ENDS
```

Во время редактирования связей сегменты ***DATA*** совмещаются и первые 100 слов ***DATA*** будут иметь в разных сегментах одинаковые адреса.

# Лекция 13

## 3.3 Макрокоманды

Для коротких фрагментов программ их организация в виде подпрограмм (процедур) может быть неэффективной. Например, добавление содержания двух слов и результат записывается в третье слово может быть реализовано так:

```
ADD_WORDS PROC
MOV AX, BX
ADD AX, CX
RET
ADD  WORDS ENDP
```

До этого добавляются три команды передачи параметров - две до вызова процедуры и одна после:

```
MOV BX, ARG1
MOV CX, ARG2
CALL ADD_WORDS
MOV SUM, AX
```

Как видим, на две команды нужно дополнительно еще 5 команд, из которых вызов процедуры реализуется как 2 команды **PUSH** и **JMP**. Следовательно, такая реализация неэффективна. Лучше просто записать 5 команд в определенном месте.

Но в макроассемблере существует для этого случая еще одно средство - макрокоманды. Макрокоманды позволяют обращаться к группе команд как к одной.

Для этого они предварительно должны быть записаны в виде макроопределения.

Макроопределение состоит из:

- Заглавие;
- Тело;
- Заключительная строка

Заглавие состоит из имен, ключевого слова MACRO и списка формальных параметров.

**Имя MACRO список параметров**

***Тело*** - это последовательность команд макроассемблера.

Список формальных параметров может отсутствовать. Если формальные параметры отмечены, то тело макроопределения их использует.

Заключительная строка макроопределения имеет вид:

**ENDM ;** без указания имени макроопределения.

Следовательно, приведенный выше фрагмент может быть реализован в виде макроопределения:

```
ADD_WORDS MACRO ARG1, ARG2, SUM
MOV AX, ARG1
ADD AX, ARG2
MOV SUM, AX
ENDM
```

Макроопределения размещаются в любом месте программы, но к вызову макрокоманды.

Обращение к макрокоманде имеет вид:

**Имя список факт.**

В нашем случае:

```
ADD_WORDS TERM1, TERM2, COST
```

Ясно, что таких обращений в программе может быть несколько. Как выполняется макрокоманда? На место вызова макрокоманды вставляется тело макроопределения с заменой формализированных параметров фактическими.

То есть, вместо одной строки обращения будет размещено 3 строки:

```
MOV AX, TERM1
ADD AX, TERM2
MOV COST, AX
```

В результате этого транслятор создает макрорасширение в каждом месте, где был вызов макрокоманды. То есть, программа будто расширяется, увеличивается. После этого макроопределения не нужно и может быть изъято. Дальше скомпонованная программа выполняется последовательно без переходов к другим частям. То есть, скорость выполнения будет высокой, но использование макрокоманд приводит к увеличению объема программы.

Таким образом видим, что вызов макрокоманды это есть **НЕ** указание микропроцессору, а директива транслятору.

Макрокоманды динамические: за счет изменения параметров можно и объекты, и сами действия. Для процедур можно изменять лишь объекты.

Макрокоманды выполняются быстрее процедур, нет потребности в переходах и возвращениях. Но использование макрокоманд увеличивает объем памяти: в теле программы макрорасширения дублируются столько раз, сколько были вызваны. Процедура же в памяти записывается один раз.

Макроопределение можно записать в библиотеку и использовать при разработке новых программ.

### *3.3.1 Псевдооператоры макроассемблера*

Разделяются на 4 группы:

1. общего назначения
2. повторение
3. условные
4. управление листингом

### *3.3.1.1. Псевдооператоры общего назначения*

Директива **LOCAL**

Пусть, есть макроопределение, где какое-то слово уменьшается до 0, чем реализуется определенная задержка.

```
DELAY_1 MACRO COUNT
LOCAL NEXT
.....
PUSH CX
MOV CX, COUNT
NEXT: LOOP NEXT
POP CX
ENDM
```

Если к этой макрокоманде обращаются в программе несколько раз, то в каждом макрорасширении появится метка NEXT, что недопустимо, потому что каждая метка должна быть уникальной.

Для того, чтобы в каждом расширении создавались уникальные метки, нужно применить директиву **LOCAL**.

После этого первый раз будет создана метка ??0000, второй -- ??0001, третий -- ??0002 и т.д. Директива **LOCAL** размещается сразу за заглавием макрорасширения.

### *3.3.1.2. Псевдооператоры повторения*

Предназначенные для повторения нескольких команд, которые начинаются заглавием и заканчиваются словом **ENDM**.

Существует 4 вида таких псевдооператоров:

1. **WHILE;**
2. **REPT;**
3. **IRP;**
4. **IRPC**

Директивы **WHILE** и **REPT** применяют для повторения определенное количество раз некоторой последовательности строк.

**WHILE константное_выражение**
**последовательность_строк**
**ENDM**

При использовании директивы **WHILE** макрогенератор транслятора будет повторять **последовательность_строк** до тех пор, пока значение **константное_выражение** не станет равно *нулю*. Это значение вычисляется каждый раз перед очередной итерацией цикла повторения (то есть значение **константное_выражение** должно подвергаться изменению внутри **последовательность_строк** в процессе макрогенерации).

**REPT константное_выражение**
**последовательность строк**
**ENDM**

Директива REPT, подобно директиве WHILE, повторяет последовательность_строк столько раз, сколько это определено значением **константное_выражение** и автоматически уменьшает на единицу значение **константное_выражение** после каждой итерации.

Например:

```
;Использование директив повторения
;prg_13_3.asm
def_sto_1 macro id_table,ln:=<5>
;макрос резервирования памяти длиной len.
;Используется WHILE
 id_table label byte
  len=ln
  while len
  db 0
  len=len-1
 endm
endm
 def_sto_2 macro id_table,len
;макрос резервирования памяти длиной len
  id_table label byte
  rept len
  db 0
 endm
endm

data segment para public 'data'
 def_sto_1 tab_1, 10
 def_sto_2 tab_2, 10
data ends
end
```

```
   21   0000                    data segment para public 'data'
   22                            def_sto_1  tab_1, 10
1  23  0000                     tab_1 label byte
2  24  0000  00                 db 0
2  25  0001  00                 db 0
2  26  0002  00                 db 0
2  27  0003  00                 db 0
2  28  0004  00                 db 0
2  29  0005  00                 db 0
2  30  0006  00                 db 0
2  31  0007  00                 db 0
2  32  0008  00                 db 0
2  33  0009  00                 db 0
   34                            def_sto_2  tab_2, 10
1  35  000A                     tab_2 label byte
2  36  000A  00                 db 0
2  37  000B  00                 db 0
2  38  000C  00                 db 0
2  39  000D  00                 db 0
2  40  000E  00                 db 0
2  41  000F  00                 db 0
2  42  0010  00                 db 0
2  43  0011  00                 db 0
2  44  0012  00                 db 0
2  45  0013  00                 db 0
   46                          data ends
```

**IRP формальный_аргумент, <строка_символов_1, ..., строка_символов_N>**
**последовательность_строк**
**ENDM**

Действие данной директивы заключается в том, что она повторяет **последовательность_строк** ***N*** раз, то есть столько раз, сколько **строк_символов** заключено в угловые скобки во втором операнде директивы **IRP**. Повторение **последовательности_строк** сопровождается заменой в ней **формального_аргумента** строкой символов из второго операнда.

Так, при первой генерации **последовательности_строк формальный_аргумент** в них заменяется на **строка_символов_1**.

Если есть **строка_символов_2**, то это приводит к генерации второй копии **последовательности_строк**, в которой **формальный_аргумент** заменяется на **строка_символов_2**. Эти действия продолжаются до **строка_символов_N** включительно.

Например:

```
IRP ini,<1,2,3,4,5>
 db ini
endm
```

Макрогенератором будет сгенерировано следующее макрорасширение:

```
db 1
db 2
db 3
db 4
db 5
```

**IRPC формальный_аргумент, строка_символов**
**последовательность строк**
**ENDM**

Действие данной директивы подобно **IRP**, но отличается тем, что она на каждой очередной итерации заменяет **формальный_аргумент** очередным символом из **строка_символов**.

Понятно, что количество повторений **последовательность_строк** будет определяться количеством символов в **строка_символов**.

Например:

```
irpc char,HELLO
      db char
endm
```

В процессе макрогенерации эта директива развернется в следующую последовательность строк:

```
DB 'H'
DB 'E'
DB 'L'
DB 'L'
DB 'O'
```

```
DB H
DB E
DB L
DB L
DB O
```

Нужно отметить, что псевдооператоры повторения несколько отличаются от команды **LOOP** и префикса **REP**. **LOOP** выполняется подобно процедуре, то есть, управление передается на начало цикла. Директивы повторения выполняются подобно макрокомандам: дублируется фрагмент соответствующее число раз.

### *3.3.1.3. Условные псевдооператоры*

Макроассемблер поддерживает несколько условных директив, которые полезны внутри макроопределений. Каждая директива **IF** должна иметь ее соответствующую директиву **ENDIF** для завершения условного блока и возможно внутри **ELSE**.

То есть, структура условного блока:

**IFxxx логическое_выражение_или_аргументы**
**фрагмент_программы_1**
**ELSE**
**фрагмент_программы_2**
**ENDIF**

Всего имеется **10** типов условных директив компиляции. Их логично попарно объединить в четыре группы:

1. Директивы **IF** и **IFE** —по результату вычисления логического выражения;
2. Директивы **IFDEF** и **IFNDEF** —по факту определения символического имени;
3. Директивы **IFB** и **IFNB** —по факту определения фактического аргумента при вызове макрокоманды;
4. Директивы **IFIDN**, **IFIDNI**, **IFDIF** и **IFDIFI** —по результату сравнения строк символов.

**IF(E) логическое_выражение**
**фрагмент_программы_1**
**ELSE**
**фрагмент_программы_2**
**ENDIF**

Обработка этих директив макроассемблером заключается в вычислении логического_выражения и включении в объектный модуль **фрагмент_программы_1** или **фрагмент_программы_2** в зависимости от того, в какой директиве **IF** или **IFE** это выражение встретилось:

- если в директиве **IF** логическое выражение *истинно*, то в объектный модуль помещается **фрагмент_программы_1**.

Если логическое выражение *ложно*, то при наличии директивы **ELSE** в объектный код помещается **фрагмент_программы_2**. Если же директивы **ELSE** нет, то вся часть программы между директивами **IF** и **ENDIF** игнорируется и в объектный модуль ничего не включается. Кстати сказать, понятие истинности и ложности значения **логического_выражения** весьма условно. Ложным оно будет считаться, если его значение равно нулю, а истинным — при любом значении, отличном от нуля.

- директива **IFE** аналогично директиве **IF** анализирует значение логического_выражения. Но теперь для включения **фрагмент_программы_1** в объектный модуль требуется, чтобы **логическое_выражение** имело значение “*ложь*”.

Директивы IF и IFE очень удобно использовать при необходимости изменения текста программы в зависимости от некоторых условий.

```
<1>...
<2>debug equ 1
<3>...
<4>.code
<5>...
<6>if debug
<7> ;любые команды и директивы ассемблера
<8> ;(вывод на печать или монитор)
<9>endif
```

**IF(N)DEF символическое_имя**
**фрагмент_программы_1**
**ELSE**
**фрагмент_программы_2**
**ENDIF**

Данные директивы позволяют управлять трансляцией фрагментов программы в зависимости от того, определено или нет в программе некоторое символическое_имя. Директива **IFDEF** проверяет, описано или нет в программе символическое_имя, и если это так, то в объектный модуль помещается **фрагмент_программы_1**. В противном случае, при наличии директивы **ELSE**, в объектный код помещается **фрагмент_программы_2**.

Если же директивы **ELSE** нет (и **символическое_имя** в программе не описано), то вся часть программы между директивами **IF** и **ENDIF** игнорируется и в объектный модуль не включается.

Действие **IFNDEF** обратно **IFDEF**. Если **символического_имени** в программе нет, то транслируется **фрагмент_программы_1**. Если оно присутствует, то при наличии **ELSE** транслируется **фрагмент_программы_2**. Если **ELSE** отсутствует, а **символическое_имя** в программе определено, то часть программы, заключенная между **IFNDEF** и **ENDIF**, игнорируется.

Пример:

Рассмотрим ситуацию, когда в объектный модуль программы должен быть включен один из трех фрагментов кода. Какой из трех фрагментов будет включен в объектный модуль, зависит от значения некоторого идентификатора sw:

1. если sw = 0, то сгенерировать фрагмент для вычисления выражения y = x*2**n;
2. если sw = 1, то сгенерировать фрагмент для вычисления выражения y = x/2**n;
3. если sw не определен, то ничего не генерировать.

```
IFNDEF sw   ;если sw не определено, то выйти из макроса
 EXITM
else        ;иначе — на вычисление
 movcl,n
 ife sw
  sal x,cl   ;умножение на степень 2 сдвигом влево
 else
  sar x,cl ;деление на степень 2 сдвигом вправо
 endif
ENDIF
```

**IF(N)B аргумент**
**фрагмент_программы_1**
**ELSE**
**фрагмент_программы_2**
**ENDIF**

Данные директивы используются для проверки фактических параметров, передаваемых в макрос. При вызове макрокоманды они анализируют значение аргумента, и в зависимости от того, равно оно пробелу или нет, транслируется либо **фрагмент_программы_1**, либо **фрагмент_программы_2**. Какой именно фрагмент будет выбран, зависит от кода директивы:

Директива **IFB** проверяет *равенство* аргумента *пробелу.* В качестве аргумента могут выступать имя или число.

- Если его значение равно пробелу (то есть фактический аргумент при вызове макрокоманды не был задан), то транслируется и помещается в объектный модуль **фрагмент_программы_1**.
- В противном случае, при наличии директивы **ELSE**, в объектный код помещается **фрагмент_программы_2**. Если же директивы **ELSE** нет, то при равенстве аргумента пробелу вся часть программы между директивами **IFB** и **ENDIF** игнорируется и в объектный модуль НЕ включается.

Действие **IFNB** обратно **IFB**. Если значение аргумента в программе НЕ РАВНО ПРОБЕЛУ, то транслируется **фрагмент_программы_1**.

Пример: строки в макроопределении, которые будут проверять, указывается ли фактический аргумент при вызове соответствующей макрокоманды:

```
Show macro reg
 IFB
  display 'не задан регистр'
  exitm
 ENDIF
...
endm
```

Если теперь в сегменте кода вызвать макрос *show* без аргументов, то будет выведено сообщение о том, что не задан регистр и генерация макрорасширения будет прекращена директивой **exitm**.

**IFIDN(I) аргумент_1, аргумент_2**
**фрагмент_программы_1**
**ELSE**
**фрагмент_программы_2**
**ENDIF**

В этих директивах проверяются **аргумент_1** и **аргумент_2** как строки символов. Какой именно код — **фрагмент_программы_1** или **фрагмент_программы_2** — будет транслироваться по результатам сравнения, зависит от кода директивы.

Парность этих директив объясняется тем, что они позволяют учитывать, либо не учитывать различие строчных и прописных букв. Так, директивы **IFIDNI** и **IFDIFI** *игнорируют* это различие, а **IFIDN** и **IFDIF** — учитывают.

Директива **IFIDN(I)** сравнивает символьные значения **аргумент_1** и **аргумент_2**.

Если результат сравнения положительный (>0), то **фрагмент_программы_1** транслируется и помещается в объектный модуль. В противном случае, при наличии директивы **ELSE**, в объектный код помещается **фрагмент_программы_2**.

Если же директивы **ELSE** нет, то вся часть программы между директивами **IFIDN(I)** и **ENDIF** игнорируется и в объектный модуль не включается.

**IFDIF(I) аргумент_1, аргумент_2**
**фрагмент_программы_1**
**ELSE**
**фрагмент_программы_2**
**ENDIF**

Действие **IFDIF(I)** обратно **IFIDN(I)**.

Если результат сравнения отрицательный (<0) ***(строки не совпадают***), транслируется **фрагмент_программы_1**. В противном случае все происходит аналогично рассмотренным ранее директивам.

Как мы уже упоминали, эти директивы удобно применять для проверки фактических аргументов макрокоманд.

Пример: Проверим, какой из регистров — **al** или **ah** — передан в макрос в качестве параметра (проверка проводится без учета различия строчных и прописных букв):

```
make_signed_word   macro   signed_byte
IFDIFI   <al>, <signed_byte>        ; убедиться, что операндом НЕ является al
        mov   al, signed_byte
ENDIF
        cwb
        endm
```

### *3.3.1.4. Операции в макроопределениях*

Есть **4** операции в макроопределениях.

1. ;; -- комментарий, который не включается в листинг;
2. & -- конкатенация;
3. Переопределение;
4. Отмена.

**&** -- конкатенация, позволяет задавать модифицированные метки и операторы;

Например:

```
DEF_TABLE MACRO SUFFIX, LENGTH
TABLE& SUFFIX DB LENGTH DUP(?)
ENDM
```

```
DEF_TABLE A,5
……..
TABLE A DB 5 DUP(?)
```

Если отметить вызов DEF_TABLE A,5, то создастся расширение
TABLE A DB 5 DUP(?)

**Переопределение**. Если в программе **описать** макрокоманду **с тем же именем**, которое было раньше у определенной макрокоманды, то предыдущее ее определение уже НЕ действует:

```
A MACRO Y
 INC Y
 INC BX
ENDM
A BX
A MACRO X, Z
 CMP X, 0
 CMP BH, 0
 JE Z
 JE EL
ENDM
A BM, EL
```

**Отмена.** Макроопределение можно уничтожить директивой
**PURGE <имя макроса>**
Из этого момента обращаться к отмеченной макрокоманде нельзя.

### *3.3.1.5. Использование библиотек макрокоманд*

Кроме макрокоманд, определенных в программе, можно использовать макрокоманды из библиотеки MACRO.LIB, или создать свои собственные и туда записать. Тогда не нужно наводить определения, а вызывать их из библиотеки с помощью директивы:

**INCLUDE <имя файла>**

Лучше это сделать в условной директиве поскольку макроопределение считывается на первом проходе транслятора. Если какие-то макроопределения ненужны, то их можно удалить:

```
IF1
 INCLUDE MACRO.LIB
ENDIF
```

```
PURGE MAC1, MAC2, MAC3
```

Всего есть возле **40** стандартных макрокоманд в виде .ASM файла. Они выполняют много функций, аналогичных функциям DOS или BIOS.

Директиву INCLUDE можно использовать и для включения других файлов

Вместо этой директивы ассемблер тидставить весь текст файла

INCLUDE A: MACROS.TXT

# Лекция 14

# Раздел 4. Структуры и записи

## 4.1. Записи

### *4.1.1. Описание типа записи*

Записи с битовыми полями - это тип записей, в которых каждое поле имеет указанную в битах длину и инициализировано некоторым значением. Размер записи с битовыми полями определяется суммой размеров ее полей.

Преимуществом данных такого типа является возможность хранения информации в наиболее компактной форме. Например, при необходимости сохранения значений 16-ти флагов, каждый из которых может иметь два состояния, традиционно используется область памяти из 16-ти байтов или слов, а при использовании записей с битовыми полями потребуется всего 16 битов.

Объявление записи в режиме Ideal имеет следующий синтаксис:

**имя RECORD [поле[,поле...]]**

Каждое "***поле***" имеет следующий синтаксис:

**имя_поля : выражение_для_размера[=значение]**

Параметр "***имя_поля***" - это символьное имя поля записи.

TASM для размещения этого поля выделяет область памяти размером, полученным в результате вычисления "***выражения_для_размера***". "***Значение***" описывает данные, которыми инициализируется поле (Это значение будет присваиваться полю при каждом создании экземпляра записи такого типа). "***Значение***" и "***выражение_для_размера***" не могут быть относительными адресами.

Например:

```
REC RECORD Am:3, Bm:3 = 7
DATE RECORD Yea:7, Mon:4, Day:5
```

Имена полей записи являются глобальными идентификаторами и не могут переопределяться. Значение имени поля при использовании в выражении является счетчиком сдвига для пересылки поля в крайнее правое положение.

"***Имя***" является символьным именем данных типа запись, по которому к ним можно обращаться в тексте программы. Кроме того, эти имена можно использовать для создания переменных и размещения их в памяти.

Данные типа запись (не отдельные поля !!!) являются переопределяемыми, т.е. в тексте программы можно несколько раз объявлять данные такого типа с одним и тем же именем.

Для определения данных типа запись TASM поддерживает многострочный синтаксис.

### *4.1.2. Описание переменных записей*

При создании экземпляра данных типа записи в качестве директивы объявления и распределения данных используется имя данных типа записи.

Синтаксис описания переменных имеет вид:

**имя_переменной имя_типа <нач_знач[; нач_знач]>**

где **нач_знач** – это:

- константное выражение;
- или «?»;
- или пусто.

```
R1 Rec <5,6> ; Am=5, Bm = 6
R2 Rec <,?>
VIC DATE <49,7,8> Yea=49, Mon=7, Day=8
```

Если **пусто**, то берется начальное значение из описания типа. Предусмотрены и массивы записей:

```
X DATE 100 DUP (<>) ; массив
```

Например, если в программе определен тип:

```
MyRec RECORD Val:3=4,Mode:2,Asize:4=15
MTest MyRec ?
```

то оператором MTest MyRec ? будет создан экземпляр записи типа MyRec, определяемый переменной MTest.

Экземпляр записи всегда имеет размер байта, слова или двойного слова, в зависимости от числа битов в определении записи.

### *4.1.3. Работа с полями записей*

Если сумма значений ширины полей <= 8, то размер экземпляра записи будет 1 байт, >8 и <=16 - слово, >16 и <=32 - двойное слово. Первое описанное поле помещается в старшие значащие биты записи, следующие поля помещаются в следующие направо биты. Если описанные поля не занимают полные 8, 16 или 32 бита, полная запись сдвигается направо так, что последний бит последнего поля становится младшим битом записи. Неиспользованные биты в старшей части инициализируются нулями.

MTest

| **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | | | | | | | Val | | | Mode | | Asize | | | |

Если при инициализации экземпляра данных типа запись какие-то значения инициализации полей записи опущены, то TASM заменяет их значением 0. Наиболее простой метод инициализации экземпляра записи заключается в присвоении полям экземпляра значений, указанных в определении типа записи, например:

```
MyRec{}
```

эквивалентно

```
DW (4 SHL 6) + (0 SHL 4) + (15 SHL 0)
```

### *4.1.4. Значение имени поля*

Пустые фигурные скобки представляют нулевое значение инициализации записи. Значения, указанные в инициализаторе, определяют, какие из значений инициализации полей будут перекрываться и с какими значениями. При этом используется синтаксис:

**{[имя_поля=выражение[, имя_поля=выражение..]]}**

Здесь "***имя_поля***" - имя поля в записи, "***выражение***" - значение, которое должно быть назначено указанному полю в экземпляре записи. Значение «?» эквивалентно 0. Все значение, не указанные в инициализаторе, устанавливаются TASM равным значениям, указанным в определении записи.

Например:

```
MyRec {Val=2,ASize=?}
```

эквивалентно

```
DW (2 SHL 6) + (0 SHL 4) + (0 SHL 0)
```

Другим методом инициализации экземпляра записи является применение инициализатора с **угловыми скобками** (<>). Значения, указанные в инициализаторе не имеют названий, но они должны быть заданы в том же порядке, что и соответствующие поля в определении записи. Синтаксис инициализатора такого типа:

**<[выражение[, выражение..]]>**

где "***выражение***" - задаваемое значение для соответствующего поля в определении записи. Пропущенное значение указывает, что будет инициализироваться значение, принятое по умолчанию (из определения записи). Ключевое слово ? указывает, что значение инициализации записи будет 0.

Например:

```
MyRec <,2,?> ; Mode=2,Asize=0
```

эквивалентно

```
DW (4 SHL 6) + (2 SHL 4) + (0 SHL 0)
```

Если в инициализаторе указано меньше значений, чем необходимо для инициализации всех полей экземпляра записи, TASM для всех остальных полей использует значения, принятые по умолчанию. Так, оператор

```
MyRec <1> ; Val=1
```

эквивалентен оператору

Если имя поля структуры имеет значение смещения поля в байтах от начала структуры, то имени поля записи ассемблер присваивает также определенное число. Оно равняется количеству битов, на которое поле нужно сдвинуть справа, чтобы оно оказалось прижатым к правому краю слова.

Встречая имя поля записи в тексте программы, ассемблер заменяет его своим значением. Это нужно для подготовки выполнения команд сдвига после выделения поля командой **AND**. В предыдущем фрагменте поле ***Day*** является крайним дело, и, чтобы проверить его содержание, не нужно его сдвигать справа.

Ясно, если необходимо проверить поле Mon, то его предварительно нужно сдвинуть справа на столько бит, сколько занимает поле Day, или на значение Mon = 5.

Следовательно, проверить, поле Mon = 7?

```
MOV AX, VIC
AND AX, MASK Mon; AX: 0M0
MOV CL, Mon; константа 5 к CL
SHR AX, CL; AX: 00M
CMP AX,7
JE Yes
No:
```

Оператор **MASK** возвращает *битовую маску поля*. Оператор **WIDTH** возвращает *длину поля* в битах.

***Оператор MASK***

Синтаксис

**MASK имя_поля_записи**

или

**MASK имя_записи**

Оператор MASK **имя_поля_записи** возвращает битовую маску, равную 1, для битовых позиций, занятых заданным полем записи, и равную 0 для остальных битовых позиций.

Оператор **MASK имя_записи** возвращает битовую маску, в которой биты, зарезервированные для битовых полей, установлены в 1, остальные - в 0.

Пример:

```
.DATA
COLOR    RECORD BLINK:1,BACK:3,INTENSE:1,FORE:3
MESSAGE  COLOR <0,5,1,1>
.CODE
MOV AH,MESSAGE            ;загрузка первоначального      0101 1001
AND AH,NOT MASK BACK;закрыть маску BACK     AND 1000 1111
                          ;                                0000 1001
OR  AH,MASK BLINK         ;открыть маску BLINK     OR   1000 0000
                          ;                                1000 1001
XOR AH,MASK INTENSE       ;                        XOR 0000 1000
                          ;                                1000 0001
```

***Оператор WIDTH***

Синтаксис

**WIDTH имя_записи -** возвращает общее количество битов, зарезервированное в описании записи;

**WIDTH имя_поля записи -** возвращает общее количество битов, зарезервированное для поля в описании записи;

Кроме того, как уже упоминалось, имя поля при использовании в выражении является счетчиком сдвига для пересылки поля в крайнее правое положение.

Пример

```
.DATA
COLOR    RECORD BLINK:1,BACK:3,INTENSE:1,FORE:3
WBLINK   EQU WIDTH BLINK; "WBLINK"=1 "BLINK"=7
;(показать от куда)
WBACK    EQU WIDTH BACK ; "WBACK"=3 "BACK"=4
WINTENSE EQU WIDTH INTENSE; "WINTENSE"=1 "INTENST"=3
WFORE    EQU WIDTH FORE         ; "WFORE"=3  "FORE"=0
WCOLOR   EQU WIDTH COLOR     ; "WCOLOR"=8
PROMPT   COLOR <1,5,1,1>
.CODE
.....
IF (WIDTH COLOR) GE 8      ;если COLOR=16 бит, загрузить
MOV AX,PROMPT              ;в 16-битовый регистр
ELSE                       ;иначе
MOV AL,PROMPT               ;загрузить в 8-битовый регистр
XOR AH,AH                  ;очистить старший 8-битовый
END IF                     ;регистр
```

Еще один пример использования

Пример:

```
.DATA
COLOR    RECORD BLINK:1,BACK:3,INTENSE:1,FORE:3
CURSOR   COLOR <1,5,1,1>
.CODE
;УВЕЛИЧИТЬ НА 1 "BACK" В "CURSOR" БЕЗ ИЗМЕНИЯ ДРУГИХ
;ПОЛЕЙ
MOV AL,CURSOR       ;загрузить значение из памяти
MOV AH,AL           ;создать рабочую копию   1101 1001   AH/AL
AND AL,NOT MASK BACK ;                  AND  1000 1111   MASK
                    ;                       1000 1001
;замаскировать старшие биты для запоминания старого CURSOR
MOV CL,BACK       ;загрузить позицию бита
SHR AH,CL         ;сдвинуть вправо          0000 0101    AH
INC AH            ;увеличить на 1           0000 0110    AH
SHL AH,CL         ;сдвинуть назад влево     0110 0000    AH
AND AH,MASK BACK  ;                  AND    0111 0000 MASK
                  ;                         0110 0000    AH
OR AH,AL          ;                   OR    1000 1001    AL
                  ;                         1110 1001    AH
MOV CURSOR,AH
```

## Дополнительные возможности обработки

Понимая важность типа данных «запись» для эффективного программирования, разработчики транслятора TASM, начиная с версии 3.0, включили в систему его команд две дополнительные команды на правах директив. Последнее означает, что эти команды внешне имеют формат обычных команд ассемблера, но после трансляции они приводятся к одной или нескольким машинным командам. Введение этих команд в язык TASM повышает наглядность работы с записями, оптимизирует код и уменьшает размер программы. Эти команды позволяют скрыть от программиста действия по выделению и установке отдельных полей записи.

Для установки значения и выборки значения некоторого поля записи используются команда **SETFIELD** и **GETFIELD**

**SETFIELD имя_элемента_записи приемник, регистр_источник –** устанавливает по смещению **имя_элемента_записи** в **приемник** значение, которое равно **регистр_источник.**

Работа команды **SETFIELD** заключается в следующем. Местоположение записи определяется операндом приемник, который может представлять собой имя регистра или адрес памяти. Операнд **имя_элемента_записи** определяет элемент записи, с которым ведется работа (по сути, если вы были внимательны, он определяет смещение элемента в записи относительно младшего разряда). Новое значение, в которое необходимо установить указанный элемент записи, должно содержаться в операнде **регистр_источник**. Обрабатывая данную команду, транслятор генерирует последовательность команд, которые выполняют следующие действия.

1. Сдвиг содержимого операнда **регистр_источник** влево на количество разрядов, соответствующее расположению **элемента_ записи**;
2. Выполнение логической операции **OR** над операндами **приемник** и **регистр_источник**. Результат операции помещается в операнд **приемник**.

Важно отметить, что **SETFIELD** не производит предварительной очистки элемента, в результате после логического сложения командой **OR** возможно наложение старого содержимого элемента и нового устанавливаемого значения. Поэтому требуется предварительно подготовить поле в записи путем его обнуления.

**GETFIELD имя_элемента_записи регистр_приемник, источник** - извлекает значение по смещению **имя_элемента_записи**, обнаруженное в **источнике** или по **адресу памяти**, и устанавливает в это значение в соответствующую по смещению **имя_элемента_записи** часть **регистра_приемника**.

Действие команды **GETFIELD** обратно действию **SETFIELD**. В качестве операнда **источник** может быть указан либо регистр, либо адрес памяти. В регистр, указанный операндом **регистр_приемник**, помещается результат работы команды — значение элемента записи. Интересная особенность связана с операндом **регистр_приемник**. Команда **GETFIELD** всегда использует 16-разрядный регистр, даже если вы укажете в этой команде имя 8-разрядного регистра.

Обрабатывая данную команду, транслятор генерирует последовательность команд, которые выполняют следующие действия.

1. Пересылка значания **источник** в **имя_элемента_записи регистр_приемник**

2. Выполнение логической операции **AND** над операндами **регистр_приемник** и **маской** (еденичные биты по смещению **имя_элемента_записи**). Результат операции помещается в операнд **регистр_приемник**.

В качестве примера применения команд **SETFIELD** и **GETFIELD** рассмотрим пример.

```
masm
model small
stack 256
iotest record i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
.data
flag   iotest <>
.code
main:
        mov    ax,@data
        mov    ds,ax
        mov    al,flag  ;al=108=06ch=01101100
        mov    bl,3
        setfield i5 al,bl ;al=111=06fh=01101111
        mov    al,110
        xor    bl,bl
        getfield        i5 bl,al ;i5=00  bl=10=2h=2  al=06eh=01101110
        mov    bl,1
        setfield i4 al,bl ;al=110=06eh=01101110 bl=100=4h=4
        setfield i5 al,bl ;al=108=06eh=01101110 bl=100=4h=4
exit:
        mov    ax,4c00h        ; стандартный выход
        int    21h
end main                       ;конец программы
```

### *4.3. Команды прерываний*

Их выполнения имеет много общего с командой вызова процедуры. Здесь так же происходит переход к группе команд, которая начинается с определенного адреса, и называется программой обработки прерывания.

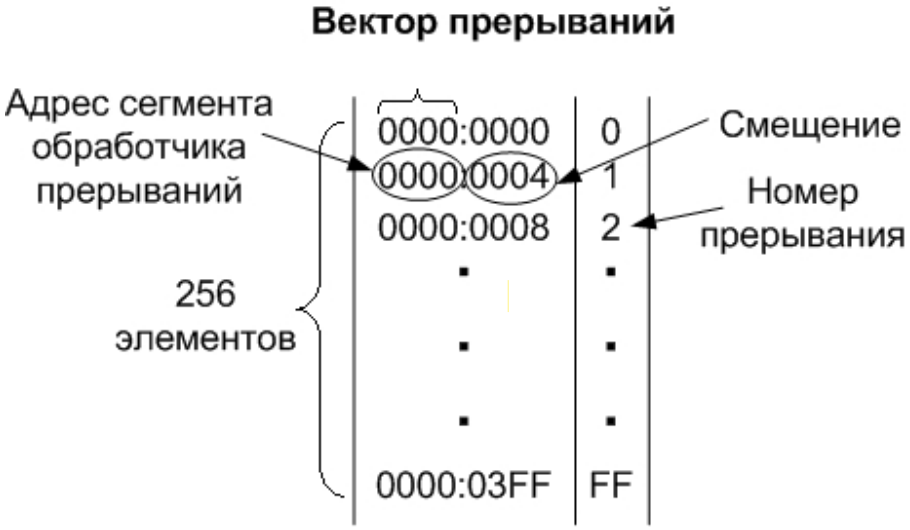
Если при вызове процедуры можно использовать разные режимы адресации, чтобы определить адрес процедуры, то переход к обработке прерывания есть непрямым и осуществляется через вектор прерывания. Вектор прерывания занимает два слова и хранит значение регистра кода **CS** и указателя команд **IP**, с которых начинается соответствующая программа обработки прерывания. Всего может быть **256** прерываний, каждое характеризуется своим номером, например, 21H. Адрес вектора прерывания определяется как номер, умноженный на 4: 4*21H.

Где находятся векторы прерывания в памяти?

Они находятся на основной системной плате - памяти RAM 64Кбайт. Самый первый 1Кбайт занимают векторы прерываний.

Главной исполнительной программой ЭВМ является базовая система введения-выведения (Basic Input/Output System) BIOS.

Она запоминает символы, которые набирают на клавиатуре, отображает символы на экране, осуществляет обмен данными между устройствами ЭВМ: дисководами, принтером и другими.




Таблица

Назначение некоторых наиболее важных векторов

| Номер | Описание |
|---|---|
| 0 | **Ошибка деления.** Вызывается автоматически после выполнения команд DIV или IDIV, если в результате деления происходит переполнение (например, при делении на 0). DOS обычно при обработке этого прерывания выводит сообщение об ошибке и останавливает выполнение программы. Для процессора 8086 при этом адрес возврата указывает на следующую после команды деления команду, а в процессоре 80286 - на первый байт команды, вызвавшей прерывание |
| 1 | **Прерывание пошагового режима.** Вырабатывается после выполнения каждой машинной команды, если в слове флагов установлен бит пошаговой трассировки TF. Используется для отладки программ. Это прерывание не |

| | |
|---|---|
| | вырабатывается после выполнения команды MOV в сегментные регистры или после загрузки сегментных регистров командой POP |
| 2 | **Аппаратное немаскируемое прерывание.** Это прерывание может использоваться по-разному в разных машинах. Обычно вырабатывается при ошибке четности в оперативной памяти и при запросе прерывания от сопроцессора |
| 3 | **Прерывание для трассировки.** Это прерывание генерируется при выполнении однобайтовой машинной команды с кодом CCh и обычно используется отладчиками для установки точки прерывания |
| 4 | **Переполнение.** Генерируется машинной командой INTO, если установлен флаг OF. Если флаг не установлен, то команда INTO выполняется как NOP. Это прерывание используется для обработки ошибок при выполнении арифметических операций |
| 5 | **Печать копии экрана.** Генерируется при нажатии на клавиатуре клавиши PrtScr. Обычно используется для печати образа экрана. Для процессора 80286 генерируется при выполнении машинной команды BOUND, если проверяемое значение вышло за пределы заданного диапазона |
| 6 | Неопределенный код операции или длина команды больше 10 байт (для процессора 80286) |
| 7 | Особый случай отсутствия математического сопроцессора (процессор 80286) |
| 8 | ***IRQ0*** - прерывание интервального **таймера**, возникает 18,2 раза в секунду |
| 9 | ***IRQ1*** - прерывание от **клавиатуры** |
| A | ***IRQ2*** - используется для каскадирования аппаратных прерываний |
| B | ***IRQ3*** - прерывание асинхронного порта **COM2** |
| C | ***IRQ4*** - прерывание асинхронного порта **COM1** |
| D | ***IRQ5*** - прерывание от контроллера жесткого диска для XT |
| E | ***IRQ6*** - прерывание генерируется контроллером флоппи-диска |
| F | ***IRQ7*** - прерывание **принтера** |
| 10 | Обслуживание **видеоадаптера** |
| 11 | **Определение конфигурации** устройств в системе |
| 12 | **Определение размера оперативной памяти** в системе |
| 13 | Обслуживание **дисковой** системы |
| 14 | Последовательный **ввод/вывод** |
| 15 | Расширенный сервис для AT-компьютеров |
| 16 | Обслуживание клавиатуры |
| 17 | Обслуживание принтера |
| 18 | Запуск BASIC в ПЗУ, если он есть |
| 19 | Загрузка операционной системы |
| 1A | Обслуживание часов |
| 1B | Обработчик прерывания Ctrl-Break |
| 1C | Прерывание возникает 18.2 раза в секунду |
| 1D | **Адрес видео таблицы** для контроллера видеоадаптера 6845 |
| 1E | Указатель на таблицу параметров дискеты |
| 1F | Указатель на графическую таблицу для символов с кодами ASCII 128-255 |
| 20-5F | Используется DOS или зарезервировано для DOS |
| 60-67 | Прерывания, зарезервированные для пользователя |
| 68-6F | Не используются |

| | |
|---|---|
| 70 | ***IRQ8*** - прерывание от часов реального времени |
| 71 | ***IRQ9*** - прерывание от контроллера EGA |
| 72 | ***IRQ10*** - зарезервировано |
| 73 | ***IRQ11*** - зарезервировано |
| 74 | ***IRQ12*** - зарезервировано |
| 75 | ***IRQ13*** - прерывание от математического сопроцессора |
| 76 | ***IRQ14*** - прерывание от контроллера жесткого диска |
| 77 | ***IRQ15*** – зарезервировано |
| 78 - 7F | Не используются |
| 80-85 | Зарезервированы для BASIC |
| 86-F0 | Используются интерпретатором BASI |
| F1-FF | Не используются |

Таблица

Резервирование групп прерываний

| | |
|---|---|
| 00000 | Векторы прерываний BIOS (0H - 1FH) |
| 00080h | Векторов прерываний DOS (20H - 3FH) |
| 0180h | Векторов прерываний пользователя (60H - 7FH) |
| 0100h | Зарезервировано векторы прерываний (40H - 5FH) |
| 0400h | Области данных BIOS |
| 0200h | Векторов прерываний Бейсика (80H - FFH) |
| 0600h | 62,5K Области данных, доступной для чтения и записи. |
| 0500h | Области данных Бейсик и DOS |

Как и вызов подпрограммы, прерывания может иметь дистанцию **NEAR** или **FAR**.

Во время выполнения программы обработки прерывания запрещаются маскируемые прерывания и трассировки программы. Поэтому флаги **IF** и **TF** обнуляются. Следовательно, программы обработки прерываний изменяют значение регистра флагов. Поэтому перед началом обработки прерывания в стек засылается регистр флагов **F**. Поэтому, в отличие от вызова подпрограмм, здесь в стеке будет три слова:

| | |
|---|---|
| a | IP |
| a+2 | CS |
| a+4 | F |

Существует три команды прерываний:

**MOV AH, номер_функции**

**<параметры>**

**INT 21H; прерывание DOS**

1. INT;
2. INTO;
3. IRET.

**INT** (interrupt) - команда условного прерывания

Прерывает обработку программы, передает управление в **DOS** или **BIOS** для определенного действия и затем возвращает управление в прерванную программу для продолжения обработки. Наиболее часто прерывание используется для выполнения

операций ввода или вывода. Для выхода из программы на обработку прерывания и для последующего возврата команда **INT** выполняет следующие действия:

1. уменьшает указатель стека на 2 и заносит в вершину стека содержимое флагового регистра;
2. очищает флаги **TF** и **IF**;
3. уменьшает указатель стека на 2 и заносит содержимое регистра **CS** в стек;
4. уменьшает указатель стека на 2 и заносит в стек значение командного указателя;
5. обеспечивает выполнение необходимых действий;
6. восстанавливает из стека значение регистра и возвращает управление в прерванную программу на команду, следующую после **INT**.

Этот процесс выполняется полностью автоматически. Необходимо лишь определить сегмент стека достаточно большим для записи в него значений регистров.

Рассмотрим два типа прерываний: команду **BIOS INT 10H** и команду **DOS INT 21H** для вывода на экран и ввода с клавиатуры. В последующих примерах в зависимости от требований используются как **INT 10H** так и **INT 21H**.

**INTO** (interrupt if overflow) - прерывание при переполнении. Приводит к прерыванию при возникновении переполнения (флаг **OF** установлен в 1) и выполняет команду **IRET 4**. Адрес подпрограммы обработки прерывания (вектор прерывания) находится по адресу 10H.

**IRET** (interrupt return) - команда возвращения после прерывания. Обеспечивает возврат из подпрограммы обработки прерывания. Команда **IRET** выполняет следующее:

1) помещает слово из вершины стека в регистр **IP** и увеличивает значение **SP** на 2;
2) помещает слово из вершины стека в регистр **CS** и увеличивает значение **SP** на 2;
3) помещает слово из вершины стека во флаговый регистр и увеличивает значение **SP** на 2.

**ПРЕРЫВАНИЯ BIOS**

**INT 05H** Печать экрана **-** выполняет вывод содержимого экрана на печатающее устройство. Команда INT 05H выполняет данную операцию из программы, а нажатие клавишей Ctrl/PrtSc - с клавиатуры. Операция запрещает прерывания и сохраняет позицию курсора.

**INT 10H** Управление дисплеем **-** обеспечивает экранные и клавиатурные операции. Прерывание INT 10H обеспечивает управление всем экраном. В регистре AH устанавливается код, определяющий функцию прерывания. Команда сохраняет содержимое регистров **BX, CX, DX, SI** и **BP.**

**Пример:**

```
                MOV  AH,02          ;Установить положение курсора
                MOV  BH,00          ;Страница 0
                MOV  DH,строка      ;Строка
                MOV  DL,столбец     ;Столбец
                INT  10H            ;Вызвать BIOS
```

или

```
                MOV  AX,0600H  Полная прокрутка вверх всего экрана,  очищая  его
```

пробелами

```
                MOV  AX,0601H  ;Прокрутить на одну строку вверх
                MOV  BH,07     ;Атрибут: нормальный, черно-белый
                MOV  CX,0000   ;Координаты от 00,00
                MOV  DX,184FH  ; до 24,79 (полный экран)
                INT  10H       ;Вызвать BIOS
```

**INT 11H** Запрос списка присоединенного оборудования **-** определяет наличие различных устройств в системе, результирующее значение возвращает в регистре AX. При включении компьютера система выполняет эту операцию и сохраняет содержимое AX в памяти по адресу 410h. Значения битов в регистре AX:

| Бит | Устройство |
|---|---|
| 15,14 | Число подключенных принтеров |
| 13 | Последовательный принтер |
| 12 | Игровой адаптер |
| 11-9 | Число последовательных адаптеров стыка RS232 |
| 7,6 | Число дискетных дисководов, при бите 0=1: 00=1, 01=2, 10=3 и 11=4 |
| 5,4 | Начальный видео режим:<br>00 = не используется,<br>01 = 40x25 плюс цвет,<br>10 = 80x25 плюс цвет,<br>11 = 80x25 черно-белый режим |
| 1 | Значение 1 говорит о наличии сопроцессора |
| 0 | Значение 1 говорит о наличии одного или более дисковых устройств и загрузка операционной системы должна осуществляться с диска |

**INT 12H** Запрос размера физической памяти -возвращает в регистре AX размер памяти в килобайтах, например, шест.200 соответствует памяти в 512 К. Данная операция полезна для выравнивания размера программы в соответствии с доступной памятью.

**INT 13H** Дисковые операции ввода/вывода - обеспечивает операции ввода-вывода для дискет и винчестера.

**INT 14H** Управление коммуникационным адаптером - обеспечивает последовательный ввод-вывод через коммуникационный порт RS232. Регистр DX должен содержать номер (0 или 1) адаптера стыка RS232. Четыре типа операции, определяемые регистром AH, выполняют прием и передачу символов и возвращают в регистре AX байт состояния коммуникационного порта.

**INT 15H** Кассетныеоперациивводa-вывода и специальные функциидлякомпьютеров AT - обеспечивает операции ввода-вывода для кассетного магнитофона, а также расширенные операции для компьютеров AT.

**INT 16H** Ввод склавиатуры - обеспечивает три типа команд ввода с клавиатуры.

**INT 17H** Вывод напринтер - обеспечивает вывод данных на печатающее устройство.

**INT 18H** Обращение к BASIC, встроенному в ROM - вызывает BASIC-интерпретатор, находящийся в постоянной памяти ROM.

**INT 19H** Перезапуск системы - данная операция при доступном диске считывает сектор 1 с дорожки 0 в область начальной загрузки в памяти (сегмент 0, смещение 7C00) и передает управление по этому адресу. Если дисковод не доступен, то операция передает управление через INT 18H в ROM BASIC. Данная операция не очищает экран и не инициализирует данные в ROM BASIC, поэтому ее можно использовать из программы.

**INT 1AH** Запрос и установка текущего времени и даты - считывает и записывает показание часов в соответствии со значением в регистре AH. Для определения продолжительности выполнения программы можно перед началом выполнения установить часы в 0, а после считать текущее время. Отсчет времени идет примерно 18,2 раза в секунду. Значение в регистре AH соответствует следующим операциям:

**AH=00** Запрос времени. В регистре CX устанавливается старшая часть значения, а в регистре DX - младшая. Если после последнего запроса прошло 24 часа, то в регистре AL будет не нулевое значение.

**AH=01** Установка времени. Время устанавливается по регистрам CX (старшая часть значения) и DX (младшая часть значения).

Коды 02 и 06 управляют временем и датой для AT.

**INT 1FH** Адрес таблицы графических символов - в графическом режиме имеется доступ к символам с кодами 128-255 в 1К таблице, содержащей по восемь байт на каждый символ. Прямой доступ в графическом режиме обеспечивается только к первым 128 ASCII-символам (от 0 до 127).

**ПРЕРЫВАНИЯ DOS**

Во время своей работы BIOS использует два модуля DOS: IBMBIO.COM и IBMDOS.COM. Так как модули DOS обеспечивают большое количество разных дополнительных проверок, то операция DOS проще в использовании и менее машиннозависимы, чем их BIOS аналоги.

Модуль IBMBIO.COM обеспечивает интерфейс с BIOS низкого уровня. Эта программа выполняет управление вводом-выводом при чтении данных из внешних устройств в память и записи из памяти на внешние устройства.

Модуль IBMDOS.COM содержит средства управления файлами и ряд сервисных функций, таких как блокирование и деблокирование записей. Когда пользовательская программа выдает запрос INT 21H, то в программу IBMDOS через регистры передается определенная информация. Затем программа IBMDOS транслирует эту информацию в один или несколько вызовов IBMBIO, которая в свою очередь вызывает BIOS. Указанные связи приведены на следующей схеме:

| **Пользовательский уровень** | **Высший уровень** | **Низший уровень** | **ROM** | **Внешний уровень** |
|---|---|---|---|---|
| Программный запрос в/в | DOS IBMDOS.COM | DOS IBMBIO.COM | BIOS | Устройство |

Прерывания от 020h до 062h зарезервированы для операций DOS. Ниже приведены наиболее основные из них:

**INT 20H** Завершение программы **-** запрос завершает выполнение программы и передает управление в DOS. Данный запрос обычно находится в основной процедуре.

**INT 21H** Запрос функций DOS **-** Основная операция DOS, вызывающая определенную функцию в соответствии с кодом в регистре AH.

**INT 22H** Адрес подпрограммы обработки завершения задачи **(см.INT 24H).**

**INT 23H** Адрес подпрограммы реакции на Ctrl/Break **(см.INT 24H).**

**INT 24H** Адрес подпрограммы реакции на фатальную ошибку **-** в этом элементе и в двух предыдущих содержатся адреса, которые инициализируются системой в префиксе программного сегмента и, которые можно изменить для своих целей.

**INT 25H** Абсолютное чтение с диска

**INT 26H** Абсолютная запись надиск

**INT 27H** Завершение программы,оставляющее ее резидентной **-** позволяет сохранить COM-программу в памяти.

**ФУНКЦИИ ПРЕРЫВАНИЯ DOS INT 21H**

Ниже приведены базовые функции для прерывания **DOS INT 21H**. Код функции устанавливается в регистре **AH:**

**00** Завершение программы (аналогично INT 20H).
**01** Ввод символа с клавиатуры с эхом на экран.
**02** Вывод символа на экран.
**03** Ввод символа из асинхронного коммуникационного канала.
**04** Вывод символа на асинхронный коммуникационный канал.
**05** Вывод символа на печать.
**06** Прямой ввод с клавиатуры и вывод на экран.
**07** Ввод с клавиатуры без эха и без проверки Ctrl/Break.
**08** Ввод с клавиатуры без эха с проверкой Ctrl/Break.
**09** Вывод строки символов на экран.
**0A** Ввод с клавиатуры с буферизацией.
**0B** Проверка наличия ввода с клавиатуры.
**0C** Очистка буфера ввода с клавиатуры и запрос на ввод.
**0D** Сброс диска (гл.16).
**0E** Установка текущего дисковода.
**0F** Открытие файла через FCB.
**10** Закрытие файла через FCB.

**11** Начальный поиск файла по шаблону.
**12** Поиск следующего файла по шаблону.
**13** Удаление файла с диска.
**14** Последовательное чтение файла.
**15** Последовательная запись файла.
**16** Создание файла.
**17** Переименование файла.
**18** Внутренняя операция DOS.
**19** Определение текущего дисковода.
**1A** Установка области передачи данных (DTA.
**1B** Получение таблицы FAT для текущего дисковода.
**1C** Получение FAT для любого дисковода.
**21** Чтение с диска с прямым доступом.
**22** Запись на диск с прямым доступом.
**23** Определение размера файла.
**24** Установка номера записи для прямого доступа.
**25** Установка вектора прерывания.
**26** Создание программного сегмента.
**27** Чтение блока записей с прямым доступом.
**28** Запись блока с прямым доступом.
**29** Преобразование имени файла во внутренние параметры.
**2A** Получение даты (CX-год,DH-месяц,DL-день).
**2B** Установка даты.
**2C** Получение времени (CH-час,CL-мин,DH-с,DL-1/100с).
**2D** Установка времени.
**2E** Установка/отмена верификации записи на диск.

# Лекция 15

# Раздел 4. Структуры и записи

## *4.2. Определение структур и объединений*

Рассмотренные нами ранее массивы представляют собой совокупность однотипных элементов. Но часто в приложениях возникает необходимость рассматривать некоторую совокупность данных разного типа как некоторый единый тип. Это очень актуально, например, для программ баз данных, где с одним объектом может ассоциироваться совокупность данных разного типа.

Структуры и объединеия - это специальные конструкции ассемблера, позволяющие смешивать и объединять данные различных типов. Структура представляет тип данных, в котором содержится один или более элементов данных, называемых членом структуры.

Структура отличается от записи тем, что члены структуры всегда представлены определенным количеством байтов. Размер структуры определяется суммарным размером всех элементов данных, входящих в нее.

Объединения подобны структурам, за исключением того, что все члены объединения занимают один и тот же участок памяти. Размер объединеия таким образом определяется размером самого большого его члена. Объединеия применяются в тех случаях, когда блок памяти должен представлять один из нескольких возможных типов данных,

Ассемблер позволяет применять вложенные структуры и объединения, хотя злоупотребление этим значительно усложняет восприятие программы.

С целью повысить удобство использования языка ассемблера в него также был введен такой тип данных.

По определению **структура** — это тип данных, состоящий из фиксированного числа элементов разного типа.

Для использования структур в программе необходимо выполнить три действия.

1. Задать шаблон структуры. По смыслу это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
2. Определить экземпляр структуры. Этот этап подразумевает инициализацию конкретной переменной с заранее определенной (с помощью шаблона) структурой.
3. Организовать обращение к элементам структуры.

Очень важно хорошо понимать разницу между описанием структуры в программе и ее определением. Описание структуры в программе означает лишь указание компилятору ее схемы, или шаблона; память при этом не выделяется. Компилятор извлекает из этого шаблона информацию о расположении полей структуры и их значениях по умолчанию. Определение структуры означает указание транслятору на выделение памяти и присвоение этой области памяти символического имени. Описать структуру в программе можно только один раз, а определить — любое количество раз. После того как структура определена, то есть ее имя связано с именем переменной, возможно обращение к полям структуры по их именам.

### *Описание шаблона структуры*

Описание шаблона структуры имеет следующий синтаксис:

**имя_структуры STRUC**
**<описание полей>**
**имя_структуры ENDS**

Здесь <описание полей> представляет собой последовательность директив описания данных **DB, DW, DD, DQ** и **DT**. Их операнды определяют размер полей и при

необходимости — начальные значения. Этими значениями будут, возможно, инициализироваться соответствующие поля при определении структуры.

Как мы уже отметили, при описании шаблона память не выделяется, так как это всего лишь информация для транслятора. Местоположение шаблона в программе может быть произвольным, но, следуя логике работы однопроходного транслятора, шаблон должен быть описан раньше, чем определяется переменная с типом данных структуры. То есть при описании в сегменте данных переменной с типом некоторой структуры ее шаблон необходимо описать в начале сегмента данных либо перед ним.

Рассмотрим работу со структурами на примере моделирования базы данных о сотрудниках некоторого отдела. Для простоты, чтобы уйти от проблем преобразования информации при вводе, условимся, что все поля символьные.

Определим структуру записи этой базы данных следующим шаблоном:

```
worker struc ;информация о сотруднике
 nam db 30 dup (" ")      ;фамилия, имя, отчество
 sex db " "               ;пол
 position db 30 dup (" ") ;должность
 age db 2 dup (" ")       ;возраст
 standing db 2 dup (" ")  ;стаж
 salary db 4 dup (" ")    ;оклад в рублях
 birthdate db 8 dup (" ") ;дата рождения
worker ends
```

***Создание экземпляра структуры***

Для использования описанной с помощью шаблона структуры в программе необходимо определить переменную с типом данных структуры. Для этого используется следующая синтаксическая конструкция:

**[имя переменной] имя_структуры <[список значений]>**

Здесь:

**имя переменной** — идентификатор переменной данного структурного типа. Задание имени переменной необязательно. Если его не указать, будет просто выделена область памяти размером в сумму длин всех элементов структуры;

**список значений** — заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми, но его задание необязательно..

Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона. Допускается инициализация отдельных полей, но в этом случае пропущенные поля, которые будут инициализированы значениями из шаблона структуры, должны отделяться запятыми. Если при определении новой переменной с типом данных структуры мы ***согласны*** со всеми ***значениями*** полей в ее шаблоне (то есть заданными по умолчанию), то нужно просто написать ***угловые скобки***.

Пример:

```
victor worker <>
```

Для примера определим несколько переменных с типом описанной ранее структуры:

```
data segment
sotrl worker <"Гурко Андрей Вячеславович",,'художник', '33' , '15' , '1800' , '26.01.64 >
sotr2 worker <"Михайлова Наталья Геннадьевна",'ж','программист', '30','10'.'1680'.'27.10.58>
sotr3 worker <"Степанов Юрий Лонгинович",,'художник', '38' , '20','1750','01.01.58'>
sotr4 worker <"Юрова Елена Александровна",'ж','связист', '32' , '2' , , '09.01.66>
sotrS worker <> ;здесь все значения по умолчанию
data ends
```

### *Методы работы со структурами*

Смысл введения структурного типа данных в любой язык программирования состоит в объединении разнотипных переменных в один объект. В языке должны быть средства доступа к этим переменным внутри конкретного экземпляра структуры. Для того чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор. (точка):

**адресное_выражение . имя_поля_структуры**

Здесь:

- **адресное_выражение** — идентификатор переменной некоторого структурного типа или выражение в скобках в соответствии с указанными ранее синтаксическими правилами (рис.);
- **имя_поля_структуры** — имя поля из шаблона структуры (это на самом деле тоже адрес, а точнее, смещение поля от начала структуры).

Оператор «.» (точка) вычисляет выражение (**адресное_выражение**) + (**имя_поля_структуры**).

Продемонстрируем с помощью определенной нами структуры worker некоторые приемы работы со структурами. К примеру, требуется извлечь в регистр **AX** значения поля с возрастом. Так как вряд ли возраст трудоспособного человека может быть больше 99 лет, то после помещения содержимого этого символьного поля в регистр **AX** его будет удобно преобразовать в двоичное представление командой **AAD**. Будьте внимательны, так как из-за принципа хранения данных «младший байт по младшему адресу» старшая цифра возраста будет помещена в **AL**, а младшая — в **AH**. Для корректировки достаточно использовать команду ***xchg ah,al***:

```
mov ax,word ptr sotrl.age ;в al возраст sotrl
xchg ah, al
;а можно и так:
lea bx,sotrl
mov ax,word ptr [bx].age
xchg ah, al
aad ; преобразование BCD в двоичное представление
```

Давайте представим, что сотрудников не четверо, а намного больше, и к тому же их число и информация о них постоянно меняются. В этом случае теряется смысл явного определения переменных с типом worker для конкретных личностей.

Ассемблер разрешает определять не только отдельную переменную с типом структуры, но и массив структур.

К примеру, определим массив из 10 структур типа worker:

```
mas_sotr worker 10 dup (<>)
```

Дальнейшая работа с массивом структур производится так же, как и с одномерным массивом. Здесь возникает несколько вопросов. Как быть с размером и как организовать индексацию элементов массива? Аналогично другим идентификаторам, определенным в программе, транслятор назначает имени типа структуры и имени переменной с типом структуры атрибут типа. Значением этого атрибута является размер в байтах, занимаемый полями структуры. Извлечь это значение можно с помощью оператора:

**TYPE имя_типа_структуры**

После того как становится известным размер экземпляра структуры, организация индексации в массиве структур не представляет особой сложности.

Пример:

```
worker struc
……..
worker ends
……
mas_sotr worker 10 dup (<>)
……..
mov bx, type worker ;bx=77=4Dh
lea di,mas_sotr
;извлечь и вывести на экран пол всех сотрудников:
mov cx,10
cycl:
mov dl,[di].sex
... ;вывод на экран содержимого
;поля sex структуры worker
add di.bx ;к следующей структуре в массиве mas_sort
loop cycl
```

Как выполнить копирование поля из одной структуры в соответствующее поле другой структуры? Или как выполнить копирование всей структуры?

Пример: выполним копирование поля ***nam*** третьего сотрудника в поле ***nam*** пятого сотрудника:

```
worker struc
 ……..
worker ends
 …….
mas_sotr worker 10 dup (<>)
……..
mov bx,offset mas_sotr
mov si,(type worker)*2 ;si=77*2
add si,bx
mov di,(type worker)*4 ;si=77*4
add di,bx
mov cx,30
rep movsb ; пересылка элементов последовательности (ds:esi/si) в (es:edi/di) памяти
```

# Объединения

Представим ситуацию, когда мы используем некоторую область памяти для размещения того или иного объекта программы (переменной, массива или структуры). Вдруг после некоторого этапа работы у нас отпадает надобность в этих данных. В обычном случае память остается занятой до конца работы программы.

Конечно, ее можно было бы задействовать для хранения других переменных, но без принятия специальных мер нельзя изменить тип и имя данных/Неплохо было бы иметь возможность переопределить эту область памяти для объекта с другими типом и именем. Ассемблер предоставляет такую возможность в виде специального типа данных, называемого объединением.

***Объединение*** — тип данных, позволяющий трактовать одну и ту же область памяти как данные, имеющие разные типы и имена.

Описание объединений в программе напоминает описание структур, то есть сначала указывается шаблон, в котором с помощью директив описания данных перечисляются имена и типы полей:

**имя_объединения UNION**
**<описание полей>**
**имя_объединения ENDS**

Отличие объединений от структур состоит, в частности, в том, что при определении переменной типа объединения память выделяется в соответствии с размером максимального элемента. Обращение к элементам объединения происходит по их именам, но при этом нужно, конечно, помнить, что все поля в объединении накладываются друг на друга. Одновременная работа с элементами объединения исключена. В качестве элементов объединения можно использовать и структуры.

Пример, который мы сейчас рассмотрим, примечателен тем, что кроме демонстрации собственно типа данных объединение, в нем показывается возможность взаимного вложения структур и объединений. Постарайтесь внимательно отнестись к анализу этой программы. Основная идея здесь в том, что указатель на память, формируемый программой, может быть представлен в виде:

- 16-разрядного смещения;
- 32-разрядного смещения;
- пары из 16-разрядного смещения и 16-разрядной сегментной составляющей адреса;
- пары из 32-разрядного смещения и 16-разрядного селектора.

Какие из этих указателей можно применять в конкретной ситуации, зависит от режима адресации (use16 или use32) и режима работы процессора. Шаблон объединения, описанный в примере, позволяет упростить формирование и использование указателей различных типов.

### *Вложенные объединения*

Ассемблер позволяет определять вложенные директивы **UNION** и **ENDS** внутри открытого описания данных типа объединения.

В структуре каждый элемент данных начинается сразу за окончанием предыдущего. В объединении каждый элемент начинается по тому же смещению, что и предыдущий. Позволив одному элементу данных содержать целое объединение, можно получить большие возможности гибкого управления данными.

Пример:

```
CUNION   STRUC
 CTYPE    DB ?
  UNION     ;Начало объединения
   STRUC
      CT0PAR1      DW 1
      CT0PAR2      DB 2
    ENDS
   STRUC
    CT1PAR1      DB 3
    CT1PAR2      DD 4
   ENDS
  ENDS
ENDS
```

Члены объединений этого примера имеют следующие параметры:

| Имя | Тип | Смещение | Значение по умолчанию |
|---|---|---|---|
| CTYPE | BYTE | 0 | ? |
| CT0PAR1 | WORD | 1 | 1 |
| CT0PAR2 | BYTE | 3 | 2 |
| CT1PAR1 | BYTE | 1 | 3 |
| CT1PAR2 | DWORD | 2 | 4 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| CTYPE | CT0PAR1<br>CT1PAR1 | CT0PAR1<br>CT1PAR2 | CT0PAR2<br>CT1PAR2 | CT1PAR2 | CT1PAR2 |

Длина этой структуры составляет **6** байт.

Объединения отличаются от структур тем, что их члены перекрывают друг друга. При инициализации данных типа объединения необходимо проявлять осторожность, так как Ассемблер позволяет только одному члену экземпляра объединения иметь инициализирующее значение.

Например:

```
BUNION {}
```

является допустимым, так как все три члена объединения в объявлении данных типа объединения были неинициализированы. Этот оператор эквивалентен

```
DB 4 DUP(?)
```

В данном примере резервируется 4 байта, так как размер объединения определяется размером максимального его члена (в данном случае - двойное слово).

Например:

```
BUNION {z=1}
```

эквивалентно

```
DB 1
DB 3 DUP(?)
```

Конструкция

```
BUNION {x=1,z=2}
```

***Недопустима***

Альтернативным методом инициализации экземпляра объединения является использование инициализатора с угловыми скобками (<>). Значения в инициализаторе такого вида не имеют имен, поэтому они должны обязательно следовать в том же порядке, в каком описаны соответствующие члены объединения в определении структур или объединений. Ключевое слово **?** указывает, что данный член структуры будет неинициализированным. Например:

```
ASTRUC <'abc',,?>
```

является эквивалентом

```
DB 'abc'
DW 1
DD ?
```

При указании в инициализаторе экземпляра объединения меньшего количества значений инициализации Ассемблер использует для всех оставшихся членов экземпляра объединения те значения, которые были указаны при определении типа этой объединения. Таким образом, вместо ASTRUC <'abc',,> можно указать ASTRUC <'abc'>.

Если применяется инициализатор с угловыми скобками, то при инициализации вложенных объединений используются специальные соглашения. Для каждого вложенного уровня необходимо указать дополнительную пару угловых скобок. Например, для типа данных CUNION:

```
CUNION <0,<<2,>,?>>
```

эквивалентно

```
DB 0
DW 2
DB 2
DB 2 DUP(?) ;чтобы сохранить правильный размер  объединения
```

Пример: использование объединения

```
masm
model small
stack 256
.586P
pnt struc       ;структура pnt, содержащая вложенное объединение
        union           ;описание вложенного в структуру объединения
                offs_16         dw      ?
                offs_32         dd      ?
        ends            ;конец описания объединения
        segm    dw      ?
ends            ;конец описания структуры
.data
point   union   ;определение объединения, содержащего вложенную структуру
        off_16 dw       ?
        off_32 dd       ?
        point_16        pnt     <>
        point_32        pnt     <>
point   ends
tst     db      "Строка для тестирования"
adr_data        point   <>      ;определение экземпляра объединения
.code
main:
        mov     ax,@data
        mov     ds,ax
        mov     ax,seg tst
;записать адрес сегмента строки tst в поле структуры adr_data
        mov     adr_data.point_16.segm,ax
;когда понадобится можно извлечь значение из этого поля обратно, в регистр bx:
        mov     bx,adr_data.point_16.segm
;формируем смещение в поле структуры adr_data
        mov     ax,offset tst     ;смещение строки в ax
        mov     adr_data.point_16.offs_16,ax
;аналогично, когда понадобится, можно извлечь значение из этого поля:
        mov     bx,adr_data.point_16.offs_16
        exit:
        mov     ax,4c00h
        int     21h
end     main
```

Какой будет выглядеть строка, чтоб **segm=18** ?

Когда вы будете работать в защищенном режиме процессора и использовать 32-разрядные адреса, то аналогичным способом можете заполнить и задействовать описанное ранее объединение.