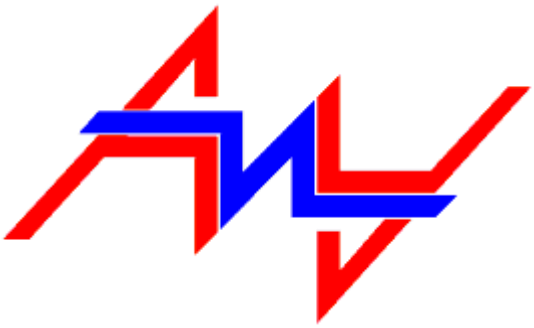
Министерство образования и науки Российской Федерации

Государственное образовательное учреждение
КАЗАНСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. А.Н.ТУПОЛЕВА



Б.А. Старостин

Специальные микропроцессоры и приборы в системах ориентации, стабилизации и навигации, часть 2

Конспект лекций

Казань 2013

Лекции 1-2

Темы:

- Представление данных в языке ассемблера
- Архитектура микропроцессора 8086
- Способы адресации
- Система команд микропроцессора 8086
- Взаимодействие с внешними устройствами
- Архитектура сопроцессора 8087

Литература:

1. Юров В.И. Assembler : Учеб. пособие для вузов / В.И. Юров. -СПб.: Питер, 2003.-624с.

2. Абель П. Язык ассемблера для IBM PC и программирования / Пер. с англ. Ю.В. Сальникова. – М.: Высш. шк. - 1992.

3. Пирогов В.Ю. ASSEMBLER. Учебный курс. – М.: Издательство Нолидж, 2001. - 848 с.

4. Зубков С.В. – Assembler для DOS, Windows и Unix – М: Пресс, 2000 г. - 608 с.

3. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT.

Основные понятия

Процессор – это устройство, которое осуществляет прием, обработку и выдачу информации.

Процессоры могут быть реализованы на различной физической основе: электронной, оптоэлектронной, оптической, биологической и даже на пневматической или гидравлической.

Микропроцессор представляет собой миниатюрное устройство, которое реализуется в виде одной или нескольких интегральных схем, которые содержат полный набор устройств, позволяющих обрабатывать данные в соответствии с заданной последовательностью команд – программой.

Обычно такие МП не имеют постоянного запоминающего устройства (ПЗУ, ROM), позволяющего хранить управляющие команды. Микропроцессоры, включающие в себя ПЗУ, носят название **однокристальных ЭВМ.**

По назначению различают **универсальные** и **специализированные** микропроцессоры.

Универсальные микропроцессоры – это МП процессоры общего назначения, которые предназначены для решения широкого круга задач вычисления, обработки и управления.

Специализированные микропроцессоры – предназначены для решения определенного класса задач, а иногда только для решения одной конкретной задачи. Их существенными особенностями являются простота управления, компактность аппаратурных средств, низкая стоимость и малая мощность потребления.

Универсальные микропроцессоры могут быть применены для решения широкого круга разнообразных задач. При этом их эффективная производительность слабо зависит от проблемной специфики решаемых задач. Специализация МП, т. е. его проблемная ориентация на ускоренное выполнение определенных функций, позволяет резко увеличить эффективную производительность при решении только определенных задач.

Среди *специализированных микропроцессоров* можно выделить различные микроконтроллеры, ориентированные на выполнение сложных последовательностей логических операций; математические МП, предназначенные для повышения производительности при выполнении арифметических операций за счет, например матричных методов их выполнения; МП для обработки данных в различных областях применений и т. д. С помощью специализированных МП можно эффективно решать новые сложные задачи параллельной обработки данных.

Формы представления данных

Положительные целые числа

Целые числа принято записывать с помощью позиционной системы кодирования. В этом случае каждая позиция числа имеет свой вес, зависящий от используе-

мой системы счисления. Общая формула для формирования целых чисел, представленных в позиционном коде, имеет вид:

$$d_n*p^n + d_{n-1}*p^{n-1} + \ldots + d_2*p^2 + d_1*p^1 + d_0*p^0,$$

где

p – основание системы счисления,

d – цифровые значения (от 0 до p-1),

n – номер позиции (отсчитывается справа налево, начиная с 0).

Примеры перевода:

из десятичной системы счисления: $\mathbf{349_{10}} \rightarrow 3*10^2+4*10^1+9*10^0=\boldsymbol{349_{10}}$;

из пятеричной системы счисления: $\mathbf{243_5} \rightarrow 2*5^2+4*5^1+5*5^0=2*25+4*5+3=\boldsymbol{73_{10}}$;

из троичной системы счисления:

$\mathbf{1202_3} \rightarrow 1*3^3+2*3^2+0*3^1+2*3^0=1*27+2*9+0+2=\boldsymbol{47_{10}}$.

в шестеричную систему счисления: $\mathbf{317_{10}} \rightarrow \boldsymbol{1245_6}$

317|6

312|52|6

5|48|8|6

4 |6|**1**

2

в пятеричную систему счисления: $\mathbf{190_{10}} \rightarrow \boldsymbol{1230_5}$

190|5

190|38|5

0|25| 7|5

3 5|**1**

2

Максимально возможное значение для n-разрядного числа вычисляется по формуле $\boldsymbol{Max = p^n-1}$, где p - основание системы счисления.

10 двоичных разрядов – 1023

5 троичных разрядов – 242

3 пятеричных разряда – 124

В вычислительной технике для представления чисел обычно используется двоичная система счисления. В этом случае требуется всего два уровня сигнала – низкий и высокий. Один двоичный разряд носит название **бит**, а 8 двоичных разрядов – **байт**.

Примеры перевода:

10 → 2

1110 → 14

10000001 → 129

11111111 → 255

Диапазоны чисел:

бит – от 0 до 1

байт – от 0 до 255

два байта – от 0 до 65535

Запись чисел в двоичной системе счисления выглядит обычно достаточно громоздко. Поэтому для компактности часто используют **шестнадцатеричную** систему счисления. В шестнадцатеричной системе счисления диапазон возможных цифр – от 0 до 15. Для обозначения цифр больше 9 здесь дополнительно используются латинские буквы от *A* до *F*.

Перевод из двоичной системы в 16-ричную и наоборот производится очень просто: двоичное число разбивается на четверки битов и каждая из этих четверок заменяется соответствующим шестнадцатеричным числом:

1111 1110 → FE

1000 1010 → 8A

Один байт содержит 2 шестнадцатеричные цифры.

Аналогично производится преобразование из двоичной системы в восьмеричную – только биты разбиваются на тетрады (по 3 бита).

Отрицательные целые числа

В ряде приложений вычислительной техники для чисел нет понятия знака. Это справедливо, например, для адресов ячеек памяти, кодов ASCII символов, результатов измерений многих физических величин, кодов управления устройствами, подключаемыми к компьютеру. Для таких чисел естественно использовать весь диапазон чисел, записываемых в ячейку того или иного размера. Если, однако, мы хотим работать как с положительными, так и с отрицательными числами, нам придется половину чисел из их полного диапазона считать положительными, а другую половину – отрицательными. В результате диапазон изменения числа уменьшается в два раза. Кроме того, необходимо предусмотреть систему кодирования, чтобы положительные и отрицательные числа не перекрывались.

В **прямом коде** знак числа определяется его старшим разрядом. В случае положительных чисел в этом разряде содержится 0, в случае отрицательных – 1.

В вычислительной технике принято записывать отрицательные числа в так называемом **дополнительном коде**, который образуется путем замены всех двоичных нулей единицами и наоборот (**обратный код**) и прибавления к полученному числу единицы.

Это справедливо как для байтовых (8-битовых) чисел, так и для чисел размером в слово или в двойное слово:

+1 → 00000001	*-1 → 11111110+1=11111111*
+25 → 00011001	*-25 → 11100110+1=11100111*
+1 → 00000000 00000001	*-1 → 11111111 11111110+1=11111111 11111111*
+25 → 00000000 00011001	*-25 → 11111111 11100110+1=11111111 11100111*

Такой способ образования отрицательных чисел удобен как минимум по трем причинам.

1). Существует только один 0:

+0 → 00000000 *-0 → 11111111+1=00000000*

2). Знак числа можно менять бесконечное число раз без каких-либо изменений и потерь:

+5 → 00000101 *-5 → 11111010+1=11111011*

-(-5) → 00000100+1=00000101

3). Позволяет выполнять над числами арифметические операции по общим правилам с получением правильного результата. При этом операция вычитания сводится к операции сложения с отрицательным числом:

5+(-5)=0 *00000101+11111011=00000000*

3-5=-2 *00000011-00000101=11111110*

3+(-5)=-2 *00000011+11111011=11111110*

Анализируя алгоритм образования отрицательного числа, можно заметить, что для всех отрицательных чисел характерно наличие двоичной единицы в старшем бите. Положительные числа, наоборот, имеют в старшем бите 0. Это справедливо для чисел любого размера.

4). Кроме того, из примеров видно, что для преобразования отрицательного 8-битового числа в 16-битовое достаточно дополнить его слева восемью двоичными единицами. Для преобразования положительного 8-битового числа в слово его надо дополнить восемью двоичными нулями. То же справедливо и для преобразования 16-разрядного числа со знаком в 32-разрядное число со знаком, только добавить придется уже не 8, а 16 единиц или нулей.

Следует подчеркнуть, что знак числа условен. Одно и то же число можно в одном контексте рассматривать, как отрицательное (-5), а в другом – как положительное, или, правильнее, число без знака (FB=251). Знак числа является характеристикой не самого числа, а нашего представления о его смысле.

Двоично-десятичные числа

Поскольку для человека привычно оперировать числами, представленными в десятичной системе счисления, то иногда применяется компромиссное решение – двоично-десятичное представление чисел (BCD-числа). **BCD** – *binary-coded decimal*.

В таком формате выдают данные некоторые измерительные приборы; он же используется часами реального времени компьютеров IBM PC для хранения информации о текущем времени.

Двоично-десятичный формат существует в двух разновидностях: **упакованный** и **распакованный**. В первом случае в байте записывается двухразрядное десятичное число от 00 до 99. Каждая цифра числа занимает половину байта и хранится в двоичной форме:

01 → 0000 0001

25 → 0010 1001

87 → 1000 0111

99 → 1001 1001

В двух байтах можно хранить в двоично-десятичном формате четырехразрядные десятичные числа от 0000 до 9999:

9604 → 1001 1010 0000 0100

8230 → 1000 0010 0011 0000

Распакованный формат отличается от упакованного тем, что в каждом байте записывается лишь одна десятичная цифра (по-прежнему в двоичной форме). В этом случае в слове можно записать десятичные числа от 00 до 99 (см. рис. 2.13)

98 → 00001001 00001000

25 → 00000010 00000101

В случае упакованного и тем более распакованного двоично-десятичного представления чисел предполагается избыточность в хранении данных.

При хранении десятичных чисел в аппаратуре обычно используется более экономный упакованный формат; умножение и деление выполняются только с распакованными числами, операции же сложения и вычитания применимы и к тем, и к другим.

Представление символов

Символы обычно кодируются целыми числами. Один байт позволяет закодировать до 256 символов. Сюда входят некоторые управляющие коды (символ с кодом 0Dh - конец строки), знаки препинания, цифры (символы с кодами 30h - 39h), большие (41h - 5Ah) и маленькие (61h - 7Ah) латинские буквы. Вторая половина символьных кодов используется для алфавитов других языков и псевдографики, набор и порядок символов в ней отличаются в разных странах и даже в пределах одной страны. Для русских букв имеются как минимум две таблицы кодировок: в DOS – кодировка ASCII, в WINDOWS – кодировка ANSY.

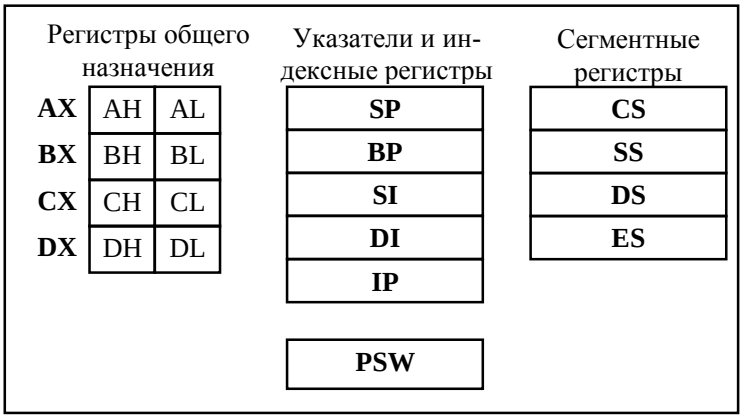
Существует также стандарт, использующий слова для хранения кодов символов, известный как UNICODE или UCS-2, и даже двойные слова (UCS-4). Кодировка UNICODE содержит алфавиты многих языков.

Лекция 3

Архитектура микропроцессора 8086

Регистры процессора разделены на три группы в соответствии с выполняемыми ими функциями. Это

- группа регистров общего назначения;
- указательная группа;
- группа сегментных регистров.



Регистры общего назначения

Микропроцессор имеет четыре 16-разрядных (двухбайтовых) регистра общего назначения: *AX*, *BX*, *CX*, *DX*. Они предназначены для хранения операндов и результатов операций и допускают адресацию не только целых регистров, но и их младшей и старшей половин. Каждый из этих регистров состоит из двух 8-разрядных (однобайтовых регистров):

AX → *AH*, *AL*

BX → *BH*, *BL*

CX → *CH*, *CL*

DX → *DH*, *DL*

Указательные регистры

Индексные регистры – *SI* и *DI*. Это 16-разрядные регистры, которые не делятся на 8-разрядные. Используются в индексных способах адресации.

Базовые регистры – *SP*, *BP*.

SP – указатель стека. Хранит адрес вершины стека.

BP – базовый регистр. Используется в различных способах адресации.

IP – указатель команд. Служит для хранения адреса текущей команды. Прямой доступ к регистру отсутствует. Косвенно на содержимое регистра влияют операторы условных и безусловных переходов.

Сегментные регистры

CS, DS, SS, ES – сегментные регистры кода, данных, стека и вспомогательный регистр. Хранят адреса, с которых начинаются соответствующие области (сегменты) оперативной памяти.

Регистр флагов

PSW – регистр флагов (слово состояния процессора).

Регистр флагов P*SW* микропроцессора 8086 содержит 16 бит (флагов), но семь из них не используются. Флаги микропроцессора разделяются на ***условные*** и ***управляющие***. **Условные** флаги (или флаги условий) отражают результат предыдущей операции (в АЛУ). **Управляющие** (или флаги управления) влияют на выполнение специальных функций.

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

Регистр флагов микропроцессора 8086.

Флаг знака ***SF*****.** Равен старшему биту предыдущего результата. Так как в дополнительном коде старший байт отрицательных чисел содержит 1, а у положительных чисел он равен 0, то флаг SF показывает знак предыдущего результата.

Флаг нуля ***ZF*****.** Устанавливается в 1 при получении нулевого результата и сбрасывается в 0, если результат отличен от нуля.

Флаг паритета ***PF*****.** Устанавливается в 1, если младшие 8 бит результата содержат четное число единиц: в противном случае он сбрасывается в 0.

Флаг переноса ***CF*****.** При сложении (вычитании) устанавливается в 1, если возникает перенос (заем) из старшего бита.

Флаг вспомогательного переноса ***AF*****.** Устанавливается в 1, если при сложении (вычитании) возникает перенос (заем) из бита 3. Флажок предназначен только для двоично-десятичной арифметики.

Флаг переполнения ***OF*****.** Устанавливается в 1, если возникает переполнение, т.е. получение результатов вне допустимого диапазона. При сложении этот флажок устанавливается, если имеется перенос в старший бит и нет переноса из старшего бита или наоборот. При вычитании он устанавливается, когда возникает заем из старшего бита, но заем в старший бит отсутствует, или наоборот.

Флаг направления ***DF*****.** Применяется в командах манипуляции цепочками данных. Если он сброшен, цепочка обрабатывается с первого элемента, имеющего наименьший адрес. В противном случае цепочка обрабатывается от наибольшего адреса к наименьшему.

Флаг разрешения прерываний ***IF*****.** Когда установлен этот флажок, ЦП распознает маскируемые прерывания; в противном случае эти прерывания игнорируются.

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

Основные архитектурные свойства и принципы, используемые при построении вычислительных устройств на базе микропроцессоров семейства 86.

- *Принцип хранимой программы*. Код программы и ее данные находятся в одном адресном пространстве в оперативной памяти.
- *Принцип микропрограммирования*. Для исполнения каждой машинной команды с помощью блока микропрограммного управления генерируется набор микрокоманд.
- *Линейное пространство памяти* – память представляется совокупностью ячеек, которым последовательно присваиваются номера (адреса).
- *Последовательное выполнение программ*. Процессор выбирает команды из памяти строго последовательно в сторону увеличения адреса. Для изменения прямолинейного хода выполнения программы или осуществления ветвления

необходимо использовать специальные команды. Они называются командами условного и безусловного перехода.

- С точки зрения процессора, нет принципиальной разницы между данными и командами. Данные и машинные команды находятся в одном пространстве памяти. Процессор, исполняя содержимое последовательных ячеек памяти, всегда пытается трактовать его как коды машинной команды. Поэтому важно в программе всегда четко разделять пространство данных и команд.
- *Безразличие к целевому назначению данных.* Процессору все равно, какую логическую нагрузку несут обрабатываемые им данные.

Архитектурные особенности микропроцессора Intel

Операционная система MS-DOS, язык ассемблера МП 86 и методы программирования микропроцессоров корпорации Intel разрабатывались применительно к 16-разрядному процессору 8086 и тому режиму, который впоследствии получил название **реального**. Появление процессора 80386 знаменовало собой начало нового этапа в развитии операционных систем и прикладного программирования – этапа многозадачных графических операционных систем **защищенного** режима типа Windows и 32-разрядных прикладных программ. При этом все архитектурные средства 86-го процессора входят в состав любого современного процессора, который, таким образом, можно условно разделить на две части:

- МП 86;

- дополнительные средства, обеспечивающие защищенный режим, 32-разрядную адресацию и другие возможности.

Из этих дополнительных средств можно выделить:

1). Средства, которые обеспечивают работу в защищенном режиме и в реальном режиме не используются (во всяком случае, явным образом; в действительности, современный процессор, даже работая в реальном режиме, использует, по крайней мере, некоторые из этих средств). Сюда, например, относятся регистры таблиц дескрипторов, регистры тестирования и отладки, привилегированные команды защищенного режима, система страничного отображения адресов и др.

2). Часть новых свойств современных процессоров, которые можно использовать и в реальном режиме, выполняя программы под управлением MS-DOS. Сюда, прежде всего, относится использование 32-битовых операндов, некоторых новых команд процессора и расширенных возможностей старых команд.

32-разрядные процессоры содержат несколько десятков программно-адресуемых регистров (не считая регистров сопроцессора), из которых шесть являются 16-разрядными, а остальные – 32-разрядными.

Регистры *общего назначения* и *регистры-указатели* отличаются от аналогичных регистров МП 86 тем, что они являются 32-разрядными. Соответственно, к их мнемоническим обозначениям добавлена буква ***E*** (*extended* – расширенный): *EAX*, *EBX*, *ECX*, *EDX*, *ESI*, *EDI*, *EBP* и *ESP*. Для сохранения совместимости с ранними моделями процессоров допускается обращение к младшим половинам всех регистров, которые имеют те же мнемонические обозначения, что и в МП 86 (*AX*, *BX*, *CX*, *DX*, *SI*, *DI*, *BP* и *SP*). Сохранена возможность работы с младшими (*AL*, *BL*, *CL* и *DL*) и старшими (*AH*, *BH*, *CH* и *DH*) половинками регистров МП 86. Однако старшие половины 32-разрядных регистров не имеют мнемонических обозначений и непосредственно недоступны. Для того, чтобы прочитать, например, содержимое старшей половины регистра *EAX* (биты 31...16) придется сдвинуть все содержимое *EAX* на 16 бит вправо (в регистр *AX*) и прочитать затем содержимое *AX*. Все регистры общего назначения и указатели программист может использовать по своему усмотрению для хранения адресов и данных размером от байта до двойного слова.

Все *сегментные* регистры (*CS*, *DS*, *SS* и *ES*), как и в МП 86, являются 16-разрядными. В их состав включено еще два регистра – ***FS*** и ***GS***, которые могут использоваться для хранения сегментных адресов двух дополнительных сегментов данных.

Регистр *указателя команд* является 32-разрядным и обычно при описании процессора его называют ***EIP***. Младшие шестнадцать разрядов этого регистра соответствуют регистру ***IP*** процессора МП 86. Весь регистр *EIP* используется только в 32-разрядных приложениях; в 16-разрядных программах используется младшая половина регистра *EIP*.

Важным элементом архитектуры, появившимся в i486, стал ***конвейер*** — специальное устройство, реализующее такой метод обработки команд внутри микропроцессора, при котором исполнение машинной команды разбивается на несколько элементарных операций. Каждая такая операция обрабатывается параллельно отдельной частью конвейера. Процессор i486 имеет пятиступенчатый конвейер. Соответствующие пять этапов включают:

1) выборку команды из кэш-памяти или оперативной памяти;
2) декодирование команды;
3) генерацию адреса, при которой определяются адреса операндов в памяти;
4) выполнение операции с помощью АЛУ;
5) запись результата (куда будет записан результат, зависит от алгоритма работы конкретной машинной команды).

После *выборки* команда попадает в *блок декодирования*. Таким образом, *блок выборки* освобождается и может выбрать следующую команду. В результате на конвейере могут находиться в различной стадии выполнения пять команд. Скорость вычисления в результате существенно возрастает.

Микропроцессоры, имеющие один конвейер, называются *скалярными,* а два и более — *суперскалярными*.

Микропроцессор *Pentium* имеет два конвейера, т.е. имеет суперскалярную архитектуру, и поэтому может выполнять две команды за один такт машинного времени. Внутренняя структура каждого конвейера такая же, как у *i486*. Микропроцессоры серии *P6* (*Pentium Pro*/II/III) имеют три конвейера с более сложной структурой.

Сопроцессор

Сопроцессор используется для поддержки операций с вещественными числами. Сначала изготавливался в виде отдельной микросхемы.

Процессор	**Сопроцессор**
8086	8087
80286	80287

80386	80387

Начиная с процессоров Intel 486 сопроцессор располагается в одном корпусе с процессором.

Сопроцессор состоит из 8 регистров разрядностью в 80 бит (R0, R1, …,R7).

Лекция 4

Организация оперативной памяти

Микропроцессор аппаратно поддерживает две модели использования оперативной памяти:

- *сегментированную модель.* В этой модели программе выделяются непрерывные области памяти (сегменты), а сама программа может обращаться только к данным, которые находятся в этих сегментах;
- *страничную модель.* Ее можно рассматривать как надстройку над сегментированной моделью. В случае использования этой модели оперативная память рассматривается как совокупность блоков фиксированного размера (4 Кбайт). Основное применение этой модели связано с организацией виртуальной памяти, что позволяет операционной системе использовать для работы программ пространство памяти большее, чем объем физической памяти. Для микропроцессоров *i486* и *Pentium* размер возможной виртуальной памяти может достигать 4 Тбайт.

Особенности использования и реализации этих моделей зависят от режима работы микропроцессора:

Реальный режим. Это режим, в котором работал *i8086*. Наличие его в *i486* и *Pentium* обусловлено тем, что фирма *Intel* старается обеспечить в новых моделях микропроцессоров функционирование программ, разработанных для ранних моделей микропроцессоров.

Защищенный режим. Этот режим позволяет максимально реализовать все архитектурные идеи, заложенные в модели микропроцессоров *Intel*, начиная с *i20286*. Программы, разработанные для *i8086* (реального режима), не могут функционировать в защищенном режиме. Одна из причин этого связана именно с особенностями формирования физического адреса в защищённом режиме.

Режим виртуального 8086. Переход в этот режим возможен, если микропроцессор уже находится в защищённом режиме. Отличительной особенностью этого режима является возможность одновременной работы нескольких программ, разра-

ботанных для *i8086*. Несмотря на то, что микропроцессор находится в защищенном режиме, в режиме виртуального *i8086* возможна работа программ реального режима. Это объясняется тем, что процесс формирования физического адреса для этих программ производится по правилам реального режима.

Режим системного управления – это новый режим работы микропроцессора, впервые появившийся в микропроцессоре *Pentium*. Он обеспечивает операционную систему механизмом для выполнения машинно-зависимых функций, таких как перевод компьютера в режим пониженного энергопотребления или выполнения действий по защите системы.

Сегментация – механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния.

В основе механизма сегментации лежит понятие *сегмента.*

Сегмент – независимый, поддерживаемый на аппаратном уровне блок памяти.

Для микропроцессоров *Intel*, начиная с *i8086*, принят особый подход к управлению памятью. Каждая программа в общем случае может состоять из любого количества сегментов, но непосредственный доступ она имеет только к трем основным сегментам: **кода, данных** и **стека,** – и от одного до трех дополнительных сегментов данных.

1. *Сегмент кода.* Содержит команды программы. Для доступа к этому сегменту служит регистр ***CS*** (*code segment register*) – ***сегментный регистр кода.*** Он содержит адрес сегмента с машинными командами текущей программы;

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

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

этому сегменту служит регистр ***SS*** (*stack segment register*) – ***сегментный регистр стека,*** содержащий адрес сегмента стека;

4. *Дополнительный сегмент данных.* Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре *DS*. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных (для *i8086* – один). 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*). Для *i8086* – в регистре *ES*.

Программы могут иметь несколько сегментов кода и данных, в зависимости от выбранной **модели памяти**:

Модель	*Количество сегментов кода*	*Количество сегментов данных*	*Количество сегментов стека*
Крошечная	Один	Один	Один
Малая	Один	Один	Один
Средняя	Много	Один	Один
Компактная	Один	Много	Один
Большая	Много	Много	Один

В *Крошечной* модели памяти области стека, данных и кода располагаются в одном сегменте памяти, в других моделях памяти эти области располагаются в разных сегментах памяти.

Программа никогда не знает, по каким физическим адресам будут размещены ее сегменты. Этим занимается операционная система. Операционная система размещает сегменты программы в оперативной памяти по определенным физическим адресам, после чего помещает значения этих адресов в определенные места. В ре-

альном режиме эти адреса помещаются непосредственно в соответствующие **сегментные регистры**, а в защищенном режиме они размещаются в элементы специальной системной **дескрипторной таблицы**.

Внутри сегмента программа обращается к адресам относительно начала сегмента линейно, то есть, начиная с 0 и заканчивая адресом, равным размеру сегмента. Этот относительный адрес, или **смещение**, который микропроцессор использует для доступа к данным внутри сегмента, называется **эффективным.**

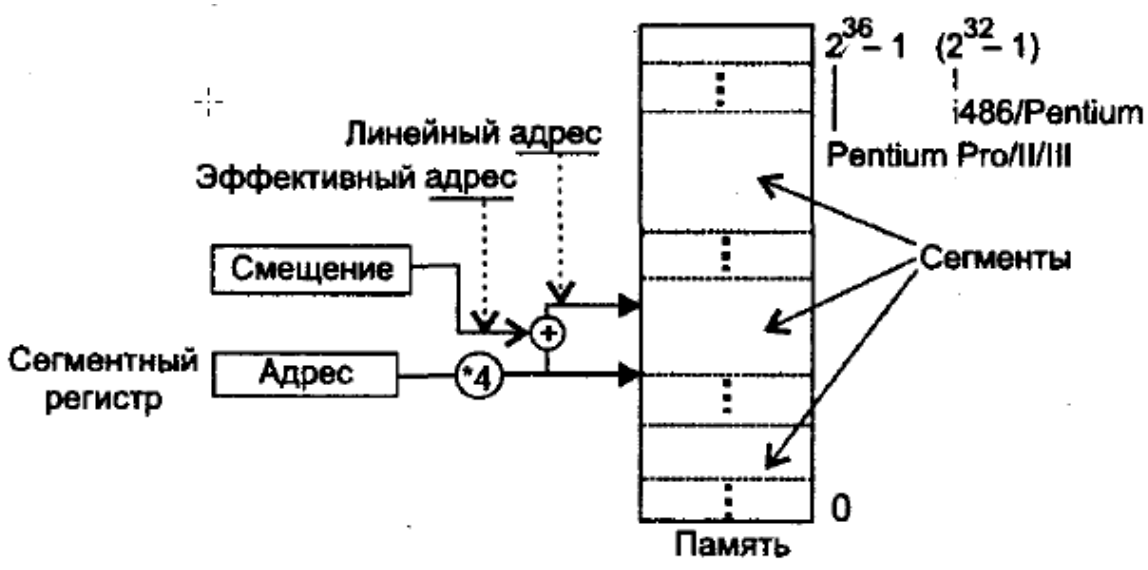
Физический абсолютный адрес

В реальном режиме механизм адресации физической памяти имеет следующие характеристики:

– диапазон изменения физического адреса от 0 до 1 Мбайт. Эта величина определяется тем, что шина адреса *i8086* имела 20 линий (бит);

– максимальный размер сегмента – 64 Кбайт. Это объясняется 16-разрядной архитектурой *i8086*. Нетрудно подсчитать, что максимальное значение, которое могут содержать 16-разрядные регистры, составляет 2^{16} - 1, что применительно к памяти и определяет величину 64 Кбайт;

– для обращения к конкретному физическому адресу оперативной памяти необходимо определить **адрес начала сегмента** (сегментную составляющую) и **смещение внутри сегмента**. Максимально возможное значение при этом соответствует 2^{16}– 1.



Сегментная модель памяти реального режима

Получается, что адрес начала сегмента может быть только в диапазоне 0-64 Кбайт от начала оперативной памяти. Возникает вопрос о том, как адресовать остальную часть оперативной памяти вплоть до 1 Мбайт с учетом того, что размер самого сегмента не превышает 64 Кбайт. Дело в том, что в сегментном регистре содержатся только старшие 16 бит физического адреса начала сегмента. Недостающие младшие четыре бита 20-битного адреса получаются сдвигом значения в сегментном регистре влево на 4 разряда. Эта операция сдвига выполняется аппаратно и для программного обеспечения абсолютно прозрачна. Получившееся 20-битное значение и является настоящим физическим адресом, соответствующим **началу сегмента**. Что касается второго компонента, участвующего в образовании физического адреса некоторого объекта в памяти – **смещения**, – то оно представляет собой 16-битное значение. Это значение может содержаться явно в команде либо косвенно в одном из регистров общего назначения, В микропроцессоре эти две составляющие складываются на аппаратном уровне, в результате чего получается физический адрес памяти размерностью 20 бит. Данный механизм образования физического адреса позволяет сделать программное обеспечение перемещаемым, то есть не зависящим от конкретных адресов загрузки его в оперативной памяти.

Недостатки такой организации памяти:

- сегменты бесконтрольно размещаются с любого адреса, кратного 16 (так как содержимое сегментного регистра аппаратно смещается на 4 разряда). Как следствие, программа может обращаться по любым адресам, в том числе и реально не существующим;
- сегменты имеют максимальный размер 64 Кбайт;
- сегменты могут перекрываться с другими сегментами.

Описание данных

Практически любая программа содержит в себе перечень данных, с которыми она работает. Это могут быть символьные строки, предназначенные для вывода на экран; числа, определяющие ход выполнения программы или участвующие в вы-

числениях; адреса подпрограмм, обработчиков прерываний или просто тех или иных полей программы; специальные коды, например, коды цвета выводимых на экран символов и т.д. Кроме данных, определяемых в тексте программы, в программу часто входят зарезервированные поля, предназначенные для заполнения по ходу выполнения программы, например, результатами вычислений или путем чтения из файла. Все эти данные и зарезервированные поля должны быть определены в составе сегмента данных программы (в принципе они могут быть определены, и часто определяются, не в сегменте данных, а в сегменте команд, но здесь мы не будем касаться этого вопроса). Для определения данных используются, главным образом, три директивы ассемблера: **db** (*define byte*, определить байт) для записи байтов, **dw** (*define word*, определить слово) для записи слов и dd (*define double*, определить двойное слово) для записи двойных слов:

```
db      255
dw      65535
dd      100000000
```

Кроме перечисленных, имеются и другие директивы, например **dq** (*define quad words*, определить четверное слово) или **dt** (*define tera words*, определить 10-байтовую переменную), но они используются значительно реже. Для того чтобы к данным можно было обращаться, они должны иметь имена. Имена данных могут включать латинские буквы, цифры (не в качестве первого знака имени) и некоторые специальные знаки, например, знаки подчеркивания (_), доллара ($) и коммерческого at (@). Длину имени некоторые ассемблеры ограничивают (например, ассемблер MASM - 31 символом), другие – нет, но в любом случае слишком длинные имена затрудняют чтение программы. С другой стороны, имена данных следует выбирать таким образом, чтобы они отражали назначение конкретного данного, например *counter* для счетчика или *filename* для имени файла:

```
counter dw 10000
filename db 'a:\myfile.txt'
```

Значения числовых данных можно записывать в различных системах счисления; чаще других используются десятичная и 16-ричная запись:

```
size    dw 256      ; в ячейку size записывается десятичное число 256
setbit  db 80h      ; в ячейку setbit записывается 16-ричное число 80h
```

Необходимо отметить неточность приведенных выше комментариев. В памяти компьютера могут храниться только двоичные коды. Если мы говорим, что в какой-то ячейке записано десятичное число 128, мы имеем в виду не физическое содержимое ячейки, а лишь форму представления этого числа в исходном тексте программы. В слове с именем *size* фактически будет записан двоичный код 0000000100000000, являющийся двоичным эквивалентом десятичного числа 256. Во втором случае в байте с именем *setbit* будет записан двоичный эквивалент шестнадцатеричного числа 80h, который составляет 10000000 (т.е. байт с установленным битом 7, откуда и получила имя эта ячейка).

Для резервирования места под массивы используется оператор dup (duplicate, дублировать), который позволяет "размножить" байт, слово или двойное слово заданное число раз:

```
rawdata  dw 300 dup (1)  ; Резервируются 300 слов, заполненных числом 1
string   db 80 dup ('*')  ; Резервируются 80 байтов, заполненных знаком '*'
```

Присвоение данным символических имен позволяет обращаться к ним в программных предложениях, не заботясь о фактических адресах этих данных. Однако программист, использующий язык ассемблера, должен иметь отчетливое представление о том, каким образом назначаются адреса ячейкам программы, и уметь работать не только с символическими обозначениями, но и со значениями адресов. Для обсуждения этого вопроса рассмотрим пример сегмента данных, в котором определяются данные различных типов. В левой колонке укажем смещения данных (в шестнадцатеричной форме), вычисляемые относительно начала сегмента.

```
0000        counter     dw  10000
0002        pages       db  'Страница 1'
0012        numbers     db  0, 1, 2, 3, 4
0017        page_addr   dw  pages
```

Все данные размещаются ассемблером друг за другом в порядке их объявления в программе. Сегмент данных начинается с данного по имени *counter*, которое

описано, как слово (2 байт) и содержит число 10000. Очевидно, что его смещение равно 0. Поскольку это данное занимает 2 байт, следующее за ним данное *pages* получило смещение 2. Данное *pages* описывает строку текста длиной 10 символов и занимает в памяти столько же байтов, поэтому следующее данное *numbers* получило относительный адрес 2 + 10 = 12. В поле *numbers* записаны 5 байтовых чисел, поэтому последнее данное сегмента с именем *page_addr* размещается по адресу 12+5=17. Ассемблер, начиная трансляцию сегмента (в данном случае сегмента данных) начинает отсчет его относительных адресов. Этот отсчет ведется в специальной переменной транслятора (не программы!), которая называется счетчиком текущего адреса и имеет символическое обозначение знака доллара ($). По мере обработки полей данных, их символические имена сохраняются в создаваемой ассемблером таблице имен вместе с соответствующими им значениями счетчика текущего адреса. Другими словами, введенные нами символические имена получают значения, равные их смещениям. Таким образом, с точки зрения транслятора *counter* равно 0, *pages* - 2, *numbers* - 12 и т.д. Поэтому предложение

```
page_addr    dw   pages
```

трактуется ассемблером, как

```
page_addr    dw   2
```

и приводит к записи в слово с относительным адресом 17 числа 2 (смещения строки *pages*).

Лекция 5

Язык ассемблера

Задачи, которые лучше всего решать на Ассемблере:

- любые программы, требующие минимального размера и максимального быстродействия,
- драйверы и программы, напрямую работающие с аппаратурой;
- ядра ОС, системные программы, работающие в защищенном режиме;
- программы для защиты информации, взлома этой защиты и защиты от таких взломов.
- резидентные программы, т.е. программы, код которых остается в оперативной памяти после их завершения и активизируется при возникновении тех или иных событий.
- Ассемблерные вставки в языках высокого уровня.

Достоинства Ассемблера

- Максимальная гибкость и максимальный доступ к ресурсам компьютера и операционной системы.
- Компактность выходного кода.
- Высокое быстродействие программ.
- Возможность ручной оптимизации программ, которая ограничивается лишь возможностями процессора.

Недостатки Ассемблера

- Трудоёмкость разработки программ в несколько раз выше чем на языках высокого уровня.
- Трудность понимания исходных текстов программ.
- Непереносимость с одной аппаратной платформы на другую.

Программы, которые осуществляют перевод программ с одного языка программирования на другой, называются **трансляторами.**

Трансляторы бывают двух видов – **компиляторы** и **интерпретаторы.**

Основные этапы создания программ

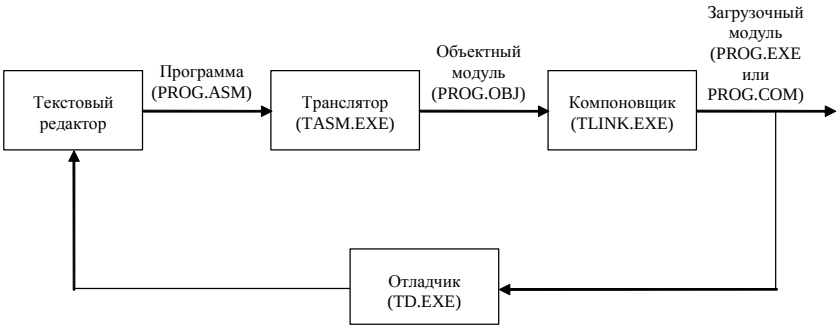
Составление программы на любом языке программирования для решения некоторой задачи начинается с изучения условия задачи и разработки алгоритма. После этого можно приступить к написанию текста программы, используя операторы языка программирования для записи действий, представленных в алгоритме. Набор текста программы производится с помощью текстового редактора.

Следующий этап – перевод программы с языка программирования в двоичные коды машинных команд. Сначала текст программы обрабатывается транслятором, который вырабатывает так называемый объектный модуль программы. Затем полученный объектный модуль обрабатывается компоновщиком и, наконец, сформированный загрузочный модуль выполняется на компьютере. Если при трансляции не обнаружено ошибок в записи операторов программы и после выполнения программы получен требуемый результат, то программа составлена правильно и пригодна для дальнейшего использования. Если нет, то следующий этап - отладка программы. На этом этапе осуществляется обнаружение и исправление ошибок, оставленных на предыдущих этапах.

Таким образом, подготовка новой программы к выполнению - это итерационный процесс, состоящих из четырех основных шагов (без учета этапа составления алгоритма) (Рис.1):

- Работа с **текстовым редактором** для создания или модификации файла исходной программы в коде ASCII.
- Работа с **транслятором** языка для получения объектного модуля программы.
- Работа с **компоновщиком** для преобразования объектного модуля программы в выполнимый в среде MS-DOS загрузочный модуль.
- **Отладка** программы с целью проверки логики и поиска ошибок в программе.

На каждом из этих этапов создания программы используются свои специальные инструментальные средства: текстовый редактор, транслятор, компоновщик и отладчик.



Основные этапы создания рабочей программы

Примером **транслятора** является турбо-ассемблер фирмы Borland (*Borland Turbo Assembler*). Транслятор хранится в файле с названием TASM.EXE. Вызов турбо-ассемблера для трансляции исходного файла осуществляется командой:

TASM [*ключи*] *source* [, *object*] [, *listing*]

где *source* - имя исходного файла;

object - имя объектного файла;

listing - имя файла листинга.

ключи:

/h, /? - получение справочной информации;

/l, /la - генерация листинга (l - обычный листинг, la - расширенный листинг);

/z - вывод вместе с сообщением об ошибке строки исходной программы;

/zi, /zd - помещение в объектный файл отладочной информации (zi - полной, zd - только номера строк).

Если имена объектного файла или файла листинга опущены, то им присваивается то же имя, что и имя исходного файла.

Пример команд для трансляции программы из файла PROG.ASM:

TASM PROG

TASM /zi PROG.ASM

TASM PROG, PROG1

Преобразование объектного файла в исполняемый производится программой, называемой **компоновщиком** или **редактором связей**. В комплект с турбо-ассемблером входит компоновщик, который содержится в файле TLINK.EXE. Вызов компоновщика для обработки объектного файла осуществляется командой:

TLINK [*ключи*] *objfile*[, *exefile*]

где *objfile* - имя входного объектного файла

exefile - имя выходного исполняемого файла

ключи:

/t - создание файла формата COM

/v - включение в исполняемый файл отладочной информации.

Если в командной строке отсутствует имя выходного файла, то ему присваивается то же имя, что и у входного.

По умолчанию TLINK.EXE создает исполняемый файл в формате .EXE. Для создания файла формата COM необходимо в командной строке компоновщика задать ключ /t.

Примеры команд компоновки:

TLINK PROG

TLINK PROG.OBJ, PROG.EXE.

Для проведения отладки используется специальный отладчик, который находится в файле TD.EXE.

Операторы языка ассемблера

Каждый оператор языка ассемблера может содержать до 4 полей следующего вида:

[*Метка*:] *Мнемокод* [*Операнды*] [*Комментарий*]

Здесь поля, не являющиеся обязательными, заключены в квадратные скобки.

Каждый оператор помещается на отдельной строке. Операторы могут располагаться в любом месте строки. Поля обязательно разделяются хотя бы одним пробелом или символом табуляции.

Поле метки

Поле метки служит для присваивания *имени* команде языка ассемблера. По нему на эту команду могут ссылаться другие команды программы. Метка команды должна заканчиваться двоеточием (:). В состав метки могут входить:

- буквы от **A** до **Z** или от **a** до **z** (Ассемблер не различает строчные и прописные буквы);
- цифры от **0** до **9**;
- специальные знаки: **? @ _ $.**

Метку можно начать любым символом, кроме цифры. Нельзя использовать в качестве меток обозначения регистров, имена команд и специальные слова. В метку нельзя вставлять пробел, его можно заменить символом подчеркивания _. Например: GET_COUNT.

Рекомендуется использовать в качестве меток имена покороче, но достаточно понятные, использовать для этого разумные сокращения. Выбирать такие имена, чтобы их легче было набирать без ошибок. Не использовать метки, которые легко спутать между собой (XXX и X4X, O0 и OO).

Поле мнемокода

Поле мнемокода содержит имя команды микропроцессора. Имена состоят из от **двух** до **шести** букв (чаще всего три). Например, *MOV* – имя команды пересылки данных (*move* – переместить), а *ADD* - имя команды сложения (*add* – сложить). При трансляции каждый мнемокод переводится в соответствующую машинную команду.

Во многих командах кроме мнемокода надо указывать один или два аргумента (операнда). Ассемблер по мнемокоду узнает, сколько должно быть операндов и какого типа, а затем обрабатывает поле операнда.

Поле операнда

В поле операндов микропроцессору сообщается, где найти данные, подлежащие обработке. Например, в команде *пересылки*

MOV CX, DX

указано, что надо скопировать содержимое регистра *DX* в регистр *CX*.

Наличие и отсутствие операндов, а также их количество зависит от мнемокода оператора. Если это поле присутствует, то в нем должно быть один-два операнда, отделенных от мнемокода по крайней мере одним пробелом или символом табуляции. Если операндов два, то между собой они разделяются запятой.

В командах с двумя операндами первый из них представляет собой *приемник,* а второй - *источник.* Операнд-источник определяет значение, которое берется микропроцессором для выполнения соответствующей операции (присвоения, сложения, вычитания, сравнения) со значением операнда-приемника или просто для загрузки в операнд-приемник. Операнды обязательно **должны быть одного типа.**

В качестве операндов могут использоваться:

- константы (только в качестве операнда-источника);
- регистры;
- переменные или адреса ячеек памяти;
- метки (только в операторах условных и безусловных переходов).

В вышеприведенном примере оператор *MOV* означает, что значение операнда-источника *DX* надо запомнить в операнде-приемнике *CX*. Поэтому при исполнении команды операнд-источник никогда не изменяется, в то время как операнд-приемник изменяется почти всегда.

Поле комментария

Это необязательное поле позволяет описывать назначение операторов исходной программы для облегчения ее понимания. Перед комментарием обязательно указывается точка с запятой (;), которая должна быть отделена от предыдущего поля, по крайней мере, одним пробелом или символом табуляции. Ассемблер игнорирует комментарии при трансляции, но сохраняет их в листинге программы.

В комментариях рекомендуется описывать не столько действие отдельной команды, сколько ее роль в программе.

Для описания программы или какой-нибудь ее части можно вводить *самостоятельные* комментарии, располагаемые на отдельной строке. Для этого достаточно начать строку точкой с запятой (;). Встретив точку с запятой в начале строки с комментарием, ассемблер игнорирует остальную часть этой строки.

Методы адресации

Делятся на семь групп:

1. Регистровая адресация
2. Непосредственная
3. Прямая
4. Косвенная регистровая
5. Базовая
6. Прямая индексная
7. Базовая индексная

Самыми быстрыми являются регистровая и непосредственная адресации, так как не требуют предварительного вычисления адреса.

Регистровая адресация

При регистровой адресации микропроцессор извлекает операнд из регистра (или загружает его в регистр). Регистры могут быть 8 или 16-разрядные.

MOV AX, CX – копирует 16 битовое содержимое регистра счетчика *CX* в аккумулятор *AX*. Содержимое регистра *CX* не изменяется.

Непосредственная адресация

Непосредственная адресация позволяет указывать 8 или 16 битовое значение константы в качестве операнда-источника. Эта константа содержится непосредственно в команде

```
MOV CX, 500      ; 500 → CX
MOV CL, 30       ; 30 → CL
```

Примеры:

```
MOV AL, CL
MOV AL, 300
MOV AX, CL
MOV AX, CX
MOV AX, 1000
MOV CL, AX
```

MOV 100, BX

Прямая адресация

При прямой адресации эффективный адрес ячейки памяти, содержащей данные, является составной частью команды. Микропроцессор использует этот адрес как смещение для формирования вместе с сегментным регистром данных *DS* 20-битового физического адреса операнда.

Обычно прямая адресация применяется, если операндом служит переменная.

Например: MOV AX, TABLE ; 3412h → AX

MOV AX, TABLE+2 ; 7856h → AX

Сегмент данных

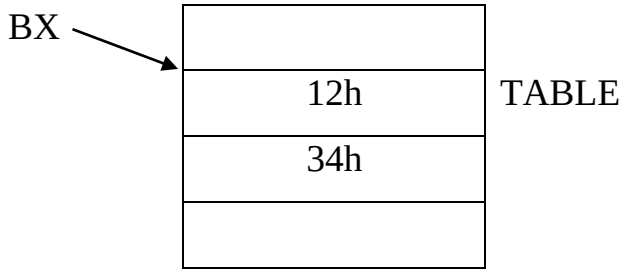
12h	TABLE
34h	
56h	TABLE+2
78h	

Косвенная регистровая

При косвенной регистровой адресации эффективный адрес ячейки памяти с данными содержится в базовом регистре *BX*, регистре указателя базы *BP* или индексном регистре (*SI* или *DI*). Косвенные регистровые операнды заключаются в квадратные скобки [ ], чтобы отличить их от регистровых операндов. По умолчанию сегментный адрес для регистров *BX*, *SI*, *DI* берется из *DS*, для регистра *BP* – из *SS*.

Например: MOV AX, [BX] ; 3412h → AX

Сегмент данных



Для размещения эффективного адреса (смещения) в регистр BX:

- применяется оператор OFFSET(смещение) к адресу ячейки памяти

MOV BX, offset TABLE

- используется оператор LEA

LEA BX, TABLE

Последовательность команд

LEA BX, TABLE

MOV AX, [BX]

эквивалентна оператору MOV AX, TABLE.

Если нужен доступ лишь к одной ячейке памяти (в данном случае TABLE), то разумнее воспользоваться одной командой. Однако для доступа к нескольким ячейкам, начиная с данного базового адреса, гораздо лучше иметь исполнительный адрес в регистре. В этом случае содержимым регистра можно манипулировать, не извлекая каждый раз новый адрес.

Существует возможность явного указания сегментного регистра:

MOV AX, DS:[BP]

Базовая адресация

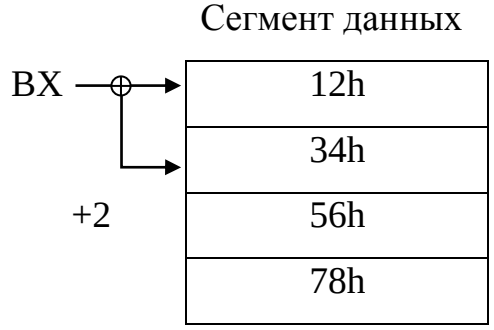
Исполнительный адрес определяется с помощью сложения значения смещения с содержимым регистров *BX* или *BP*.

Регистр BX удобно использовать при доступе к структурированным записям данных, расположенных в разных областях памяти.

Например: MOV AX, [BX]+2 ; 7856h → AX

MOV AX, [BX+2]

MOV AX, 2[BX]



Прямая индексная адресация

При прямой индексной адресации эффективный адрес вычисляется как сумма значений смещения и индексного регистра (*DI* или *SI*). Этот тип адресации удобен для доступа к элементам таблицы, когда смещение указывает на начало таблицы, а индексный регистр – на ее элемент.

Например, если *TABLE* – таблица байтов, то последовательность команд

MOV DI, 2

MOV AL, TABLE[DI]

загрузит третий элемент таблицы в регистр *AL*.

В таблице слов соседние элементы отстоят друг от друга на два байта, поэтому при работе с ней надо удваивать номер элемента при вычислении значения индекса. Если *TABLE* – таблица слов, то для загрузки в регистр AX его третьего элемента надо использовать последовательность команд

MOV DI, 4

MOV AX, TABLE[DI]

Базовая индексная адресация

При базовой индексной адресации эффективный адрес вычисляется как сума значений базового регистра (*BX* или *BP*), индексного регистра (*SI* или *DI*) и, возможно, смещения.

Пример:

MOVE AX, TABLE[BX][DI];

MOVE AX, [BX+2+DI];

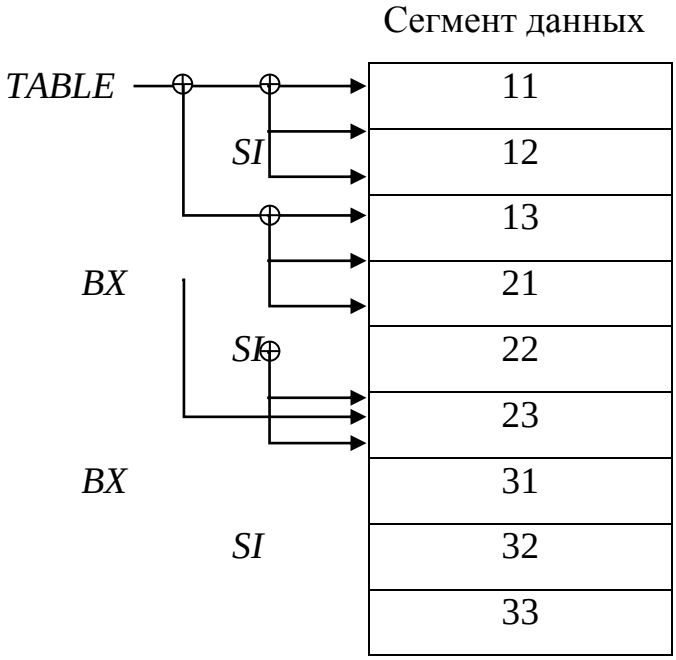
MOVE AX, [DI+BX+2];

MOVE AX, [BX+2][DI];

MOVE AX, [BX][DI+2];

Так как в этом режиме адресации складывается два отдельных смещения, то он удобен при адресации двумерных массивов, когда базовый регистр содержит начальный адрес массива, а значения сдвига и индексного регистра суть смещения по строке и столбцу.

Например, для массива *TABLE* размерностью 3x3:



MOV AL, TABLE[BX][SI]

Для перемещения по элементам этого массива *BX* должен изменяться с шагом от 0 до 6 с шагом 3 (перемещение по строкам), а *SI* – от 0 до 2 с шагом 1 (перемещение по столбцам)

Режим адресации	**Операнд**	**Сегментный регистр**
Регистровая	Регистр	Не используется
Непосредственная	Константа	Не используется
Прямая	Имя переменной + смещение	DS
Косвенная регистровая	[BP] [BX] [DI] [SI]	SS DS DS DS
Базовая	[BP] + смещение [BX] + смещение	SS DS
Прямая индексная	[DI]+ смещение [SI] + смещение	DS DS
Базовая индексная	[BP] [SI] + смещение	SS

	[BP] [DI] + смещение	SS
	[BX] [SI] + смещение	DS
	[BX] [DI] + смещение	DS

Лекция 6

Псевдооператоры (директивы)

В отличие от операторов, которые с помощью транслятора переводятся в машинные коды микропроцессора, псевдооператоры (директивы) служат для управления работой транслятора и машинные коды не генерируют. С помощью псевдооператоров можно ***определять сегменты*** и ***процедуры***, ***давать имена командам*** и ***элементам данных***, ***резервировать рабочие области памяти*** и выполнять множество других задач.

Иидентификатор EQU

Структура:

имя EQU *текст*

Подставляет значение *текст* вместо идентификатора *имя*.

Пример:

```
Message1EQU     'Сообщение$'
Truth     EQU     1
var2      EQU     4[SI]

Msg  db      Message1
     mov BL, Truth
     mov AX, var2
```

Внешние ссылки PUBLIC и EXTRN

PUBLIC *идентификатор* [, …]

Делает определенный в данном модуле идентификатор доступным другим модулям программы, которые впоследствии должны быть присоединены к этому модулю (на этапе компоновки).

EXTRN *имя*: *тип* [, …]

Указывает, что *имя* определено в другом модуле программы.

Пример:

Доступ к переменной TOTAL из другого модуля

Модуль 1

```
PUBLIC TOTAL
TOTAL DW 0:          ; там где определена TOTAL
```

Модуль 2

```
EXTRN TOTAL: WORD          ; где используется TOTAL
```

Если *имя* – идентификатор, определенный в сегментных данных или дополнительном сегменте, то тип может быть BYTE, WORD или DWORD.

Если *имя* – метка процедуры, то тип может быть NEAR или FAR.

Если *имя* – константа, то тип должен быть ABS.

INCLUDE

Вставляет содержимое указанного файла в текущий файл исходной программы

INCLUDE *имя_файла*

SEGMENT

Определяет границы сегмента программы

имя_сегмента SEGMENT [*выравнивание*] [*тип*] [*разрядность*] [*'класс'*]

выравнивание – указывает, с какого адреса может начинаться сегмент:

BYTE – с любого адреса;

WORD – с четного адреса;

DWORD – с адреса, кратного 4;

PARA (по умолчанию) – с адреса, кратного 16 (граница параграфа);

PAGE – с адреса, кратного 256.

тип – выбирает один из возможных типов комбинирования сегментов:

PUBLIC – все сегменты с одинаковым именем и разными классами будут объединены в один, располагаясь в нем последовательно;

COMMON – все сегменты с одинаковым именем и разными классами будут объединены в один, располагаясь в нем по одному и тому же адресу;

AT – сегмент должен располагаться по фиксированному абсолютному адресу (указываемый адрес измеряется в параграфах);

PRIVATE (по умолчанию) – сегмент не объединяется с другими.

STACK – сегмент будет использоваться в качестве стека.

разрядность – разрядность программы:

USE16 (по умолчанию) – для 16-разрядных программ (размер до 64 Кб)

USE32 – для 32-разрядных программ (размер до 4 Гб)

класс – любая метка, заключенная в одинарные кавычки. Все сегменты с одинаковым классом, даже сегменты типа PRIVATE, будут расположены в исполняемом файле непосредственно друг за другом.

ASSUME

Сообщает ассемблеру, с каким сегментом или группой сегментов связан тот или иной сегментный регистр.

ASSUME *сегм_регистр*: *имя_сегм*[, …]

или

ASSUME *сегм_регист*: NOTHING[, …]

Оператор ASSUME NOTHING отменяет действие всех предыдущих операторов ASSUME для данного регистра.

Модель памяти MODEL

Описание модели памяти.

.MODEL *имя_модели*

имя_модели – наименование модели памяти:

TINY – крохотная;

SMALL – малая;

COMPACT – компактная (код – в 1 сегменте, данные – в нескольких);

MEDIUM – средняя (данные – в 1 сегменте, код – в нескольких);

LARGE – большая.

Упрощенные директивы описания сегментов

.STACK *размер* ; Описание сегмента стека (*число* – размер стека в байтах)

.DATA ; Описание сегмента данных

.CODE ; Описание сегмента стека

Описание начала процедуры (подпрограммы) PROC

Присваивает имя процедуры.

имя PROC [NEAR]

или

имя PROC FAR

Конец процедуры ENDP

Завершает процедуру

имя ENDP

Конец программы END

Последний транслируемый оператор. Содержимое файла после оператора END транслятором игнорируется.

END [*метка*]

Метка – точка входа. Метка оператора, с которого начинается исполнение программы.

Директива управления программным счетчиком ORG

ORG *число*

Устанавливает значение программного счетчика в заданное значение (*число*).

Программный счетчик - внутренняя переменная ассемблера, равная смещению текущей команды или данных относительно начала сегмента. Для преобразования меток в адреса используется именно значение этого счетчика.

Управление листингом

PAGE [*число_строк*] [, *число_столбцов*]

Длина и ширина печатаемой страницы.

TITLE *текст*

Указывает заголовок, который должен быть напечатан на второй строке каждой страницы.

SUBTTL *текст*

Указывает подзаголовок, который должен быть напечатан на третьей строке каждой страницы.

Задание набора допустимых команд

.8086 – разрешены только команды процессора 8086 (по умолчанию);

.286 – разрешен набор команд процессора 80286

.386 – разрешен набор команд процессора 80386

.486 – разрешен набор команд процессора 80486

.586 – разрешен набор команд процессора Pentium;

.686 – разрешен набор команд процессора Pentium II, Pentium Pro.

Лекция 7

Структура программы

В MS DOS используются программы двух форматов – EXE и COM. Формат EXE является родным для этой операционной системы, а формат COM является устаревшим и оставлен для совместимости с программами, предназначенными для предыдущих операционных систем. Структура программы зависит от используемого формата.

Каждая программа состоит из описания входящих в нее сегментов памяти: стека, данных и кода. Описание сегментов производится с помощью директивы SEGMENT.

Программа формата EXE:

Описание сегмента стека:

```
STACKSEG SEGMENT STACK 'STACK'
    DB    64 DUP (?)
STACKSEG ENDS
```

Описание сегмента данных:

```
DATASEG SEGMENT PUBLIC 'DATA'
    A       DB 10
    B       DB ?
    … Описание переменных …
DATASEG ENDS
```

Описание сегмента кода:

```
CODESEG SEGMENT PUBLIC 'CODE'
    ASSUME CS: CODESEG, DS: DATASEG, SS: STACKSEG
    mov AX, DATASEG
    mov DS, AX
    MOV AH, A
    MOV B, AH
```

```
… Операторы программы …
CODESEG ENDS
END
```

Следует обратить внимание на то, что необходимо явным образом загрузить адрес начала сегмента данных в регистр DS. Командой ASSUME этого сделать нельзя.

Программа формата COM

Особенности программ формата COM.

Размер программы. EXE-программа может иметь любой размер, в то время как COM-файл ограничен размером одного сегмента и не превышает 64К. COM-файл всегда меньше, чем соответствующий EXE-файл.

Сегмент стека. В EXE-программе определяется сегмент стека, в то время как COM-программа генерирует стек автоматически. Таким образом, при создании ассемблерной программы формата COM, стек должен быть опущен.

Сегмент данных. В EXE программе обычно определяется сегмент данных, а регистр DS инициализируется адресом этого сегмента. В COM-программе все данные должны быть определены в сегменте кода.

Инициализация. В COM-программе все сегментные регистры содержат адрес префикса программного сегмента (PSP), – 256-байтового (100h) блока, который резервируется операционной системой DOS непосредственно перед COM или EXE программой в памяти при ее запуске и содержит служебную информацию о программе. Поэтому в COM-программе адресация начинается со смещения 100h от начала PSP (в программе после оператора SEGMENT кодируется директива ORG 100h).

Процедура создания программ. Для программ в EXE и COM форматах выполняется ассемблирование для получения OBJ-файла. На этапе компоновки для получения COM-файла необходимо дополнительно указать ключ *-t*.

```
CODESG SEGMENT 'Code'
ASSUME CS:CODESG, DS:CODESG, SS:CODESG, ES:CODESG
ORG 100H  ; Смещение для PSP

BEGIN:    JMP MAIN  ; Обход через данные
        A     DB 10
        B     DB ?
        … Описание переменных …

MAIN:
        MOV AH, A
        MOV B, AH
        … Операторы программы …
CODESG ENDS
        END BEGIN
```

Замечания.

- Стек и сегмент данных отсутствует.
- Оператор ASSUME указывает ассемблеру установить относительные адреса с начала сегмента кодов. Регистр CS также содержит этот адрес, являющийся к тому же адресом префикса программного сегмента (PSP).
- Директива ORG служит для резервирования 100h байт от начального адреса под PSP.
- Команда JMP используется для обхода данных, определенных в программе.

Упрощенные директивы описания сегментов

Программа формата EXE:

```
.MODEL SMALL
.STACK 100
```

```
        .DATA
        … Описание переменных …
        .CODE
Start:
        mov AX, @DATA
        mov DS, AX
        … Операторы программы …
        END Start
```

Программа формата COM:

```
.MODEL TINY
        .CODE
        ORG 100h

Start:      JMP Main
FLAG        DB    0
        … Описание переменных …
Main:
        … Операторы программы …
        END Start
```

Описание процедур

```
имя   PROC
        … Операторы процедуры …
        RET
имя   ENDP
```

Лекция 8

Система команд микропроцессора 8086

Систему команд можно разбить на семь функциональных групп:

1. *Команды пересылки данных* – обмен информацией между регистрами, ячейками данных и портами ввода-вывода.

2. *Арифметические команды* – арифметические операции

3. *Команды манипулирования битами* – выполняют сдвиг, циклический сдвиг и логические операции со значениями регистров и ячеек памяти.

4. *Команды передачи управления* – управляют последовательностью исполнения команд программы.

5. *Команды обработки строк* – перемещают, сравнивают и сканируют строки данных;

6. *Команды прерывания* – отвлекают микропроцессор на обработку некоторых специфических ситуаций.

7. *Прочие команды.*

Команды пересылки данных

Команды пересылки данных общего назначения

MOV	*Приемник*, *Источник*	Присвоить *Приемнику* значение *Источника*
PUSH	*Источник*	Сохранить содержимое *Источника* в стеке
POP	*Приемник*	Извлечь значение из стека и поместить его в *Приемник*
XCHG	*Приемник*, *Источник*	Обменять содержимое *Источника* и *Приемника*

Примеры:

```
MOV AX, BX
MOV BX, 10
MOV PNAME, BX
MOV PNAME[DI], AX
```

```
MOV PNAME[DI+1], 0
PUSH AX
POP BX
XCHG AX, BX
```

Команды загрузки адреса

LEA	*Рег.16, Пам.*	Загрузить в *Рег.16* эффективный адрес ячейки памяти *Пам.*
LDS	*Рег.16, Пам.*	Загрузить в *Рег.16* эффективный адрес ячейки памяти *Пам.*, а в регистр DS – адрес сегмента
LES	Рег.16, *Пам*.	Загрузить в *Рег.16* эффективный адрес ячейки памяти *Пам.*, а в регистр ES – адрес сегмента

Примеры:

```
LEA DX, Stroka
MOV DX, offset Stroka
LDS SI, Stroka
LES DI, Stroka
```

Команды пересылки флагов

LAHF – загрузить в регистр AH содержимое младшего байта регистра флагов;
SAHF – загрузить в младший байт регистра флагов содержимое регистра AH;
PUSHF – сохранить в стеке содержимое регистра флагов;
POPF – извлечь значение из стека и поместить его в регистр флагов.

Арифметические команды

Команды сложения

ADD *Приемник*, *Источник* → *Приемник = Приемник + Источник*

Команда ADD выполняет сложение указанных операндов, представленных в двоичном дополнительном коде. Может иметь длину 1-2 байта, со знаком и без знака. Микропроцессор помещает результат на место первого операнда после того, как

сложит оба операнда. Второй операнд не изменяется. Операнды обязательно должны иметь одинаковый тип. Автоматического преобразования типов не производится. Команда корректирует регистр флагов в соответствии с результатом сложения: был ли результат нулевым, отрицательным, имел ли четность, перенос или переполнение.

Например, команда ADD AX, BX складывает содержимое регистра BX с содержимым регистра AX и оставляет результат в регистре AX.

Примеры:

```
ADD AX, 1
ADD AL, MEM[BX]
ADD MEM[BX][SI], 10
```

Команда сложения с переносом.

ADC *Приемник*, *Источник* → *Приемник* = *Приемник* + *Источник* + *CF*

Команда ADC делает то же самое, что и команда ADD, но при сложении также использует флаг переноса *CF*. Перенос при сложении чисел возникает тогда, когда сумма не помещается в операнде-приемнике.

Пример:

```
      1111 1010 → (250)
     +0000 1010 → (10)
CF ← (1) 0000 0100 → (260)
```

Совместно операторы ADD и ADC позволяют складывать целые числа любой разрядности:

```
ADD AX, CX
ADC BX, DX      ; сложение (BX,AX) = (BX,AX) + (DX,CX)
```

Инкрементирование

INC *Приемник* → *Приемник* = *Приемник* + 1

Примеры:

```
INC AL
```

INC MEM

Команды вычитания

SUB *Приемник*, *Источник* → *Приемник = Приемник – Источник*

Команда выполняет вычитание содержимого *Источника* из содержимого *Приемника* и размещает результат в *Приемнике*. Содержимое *Источника* при этом не изменяется. Может иметь длину 1-2 байта.

Например, команда SUB AX, BX вычитает содержимое регистра BX из содержимого регистра AX и размещает результат в регистре AX.

Примеры:

SUB AX, 1

SUB AL, MEM[BX]

SUB MEM[BX][SI], 10

Команда вычитания с переносом.

SBB *Приемник*, *Источник* → *Приемник = Приемник – Источник – CF*

Команда SBB делает то же самое, что и команда SUB, но при вычитании также использует флаг переноса *CF*. Перенос при вычитании чисел возникает тогда, когда требуется заем.

Совместно операторы SUB и SBB позволяют вычитать целые числа любой разрядности:

SUB AX, CX

SBB BX, DX ; сложение (BX,AX) = (BX,AX) – (DX,CX)

Декрементирование

DEC *Приемник* → *Приемник = Приемник* – 1

Примеры:

DEC AL

DEC MEM

Лекция 9

Команда обращения знака NEG

NEG *Приемник*

Команда NEG вычитает содержимое *Приемника* из нулевого значения.

Примеры:

NEG AL

NEG BX

NEG MEM[DI]

Команды умножения

Команда умножения целых чисел без знака

MUL *Источник*

- Если *Источник* имеет байтовый тип (8 разрядов), то
  (AH, AL) = AL * *Источник*.
- Если *Источник* имеет тип слово (16 разрядов), то
  (DX, AX) = AX * *Источник*.

Источник может быть регистром или содержимым ячейки памяти, но не может быть константой.

Примеры:

MUL DL

MUL BX

MUL MEM

Команда умножения целых чисел со знаком

IMUL *Источник*

Исполняется точно так же, как MUL, но с учетом знака.

Команды деления

Команда деления целых чисел без знака

DIV *Источник*

- Если *Источник* имеет байтовый тип (8 разрядов), то
  (AL) = (AH, AL) / *Источник*, а остаток от деления помещается в регистр AH.
- Если *Источник* имеет тип слово (16 разрядов), то
  (AX) = (DX, AX) / *Источник*, а остаток от деления помещается в регистр DX.

Источник может быть регистром или содержимым ячейки памяти, но не может быть константой.

Примеры:

DIV DL

DIV BX

DIV MEM

Команда деления целых чисел с знаком

IDIV *Источник*

Исполняется точно так же, как DIV, но с учетом знака.

Команда сравнения CMP

CMP *Приемник*, *Источник*

Подобно команде SUB команда CMP вычитает операнд *Источник* из операнда *Приемника* и в зависимости от результата устанавливает или обнуляет флаги регистра словосостояния *PSW*. Но, в отличие от команды SUB, *команда CMP* ***не сохраняет результат вычитания***.

Команда CMP не изменяет операнды. Она целиком предназначена для установки значений флагов регистра словосостояния, на основании которых команды условного перехода будут "принимать решение" о передаче управления.

Условие	*OF*	*SF*	*ZF*	*CF*
Операнды без знака				
Приемник > Источник	H	H	0	0
Приемник = Источник	H	H	1	0
Приемник < Источник	H	H	0	1
Операнды со знаком				
Приемник > Источник	0/1	0	0	H
Приемник = Источник	0	0	1	H
Приемник < Источник	0/1	1	0	H

Здесь **H** – не имеет значения, **0/1** – значения флага может быть 0 или 1 в зависимости от знаков операндов.

Команды манипулирования битами

Логические операции

Под **логическими** понимаются операции, в основе которых лежат правила *формальной логики.* Формальная логика работает на уровне утверждений *истинно* и *ложно.* Для микропроцессора это, как правило, означает 1 и 0, соответственно. Для компьютера язык нулей и единиц является родным, но минимальной единицей данных, с которой работают машинные команды, является байт. Однако на системном уровне часто необходимо иметь возможность работать на предельно низком уровне – на уровне бит.

Основными логическими командами являются NOT (не), AND (и), OR (или), XOR (исключающее или), TEST.

Оператор NOT имеет один операнд:

NOT *op*

Он устанавливает обратное значение бит в операнде: нули становятся единицами, а единицы – нулями.

Исходный бит A	**NOT** A
0	1
1	0

Команды AND, OR и XOR имеют два операнда:

AND *op*1, *op*2

OR *op*1, *op*2

XOR *op*1, *op*2

Они осуществляют над операндами побитные операции согласно правилам, приведенным в таблице.

Исходный бит A	Исходный бит B	A **AND** B	A **OR** B	A **XOR** B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Команда **AND** устанавливает каждый бит результата (операнда-приемника *op*1) в 1 только в том случае, если оба соответствующих бита операндов *op*1 и *op*2 равны 1. Команда **OR** устанавливает каждый бит результата (операнда-приемника *op*1) в значение 1, если любой из соответствующих битов операнда-источника равен 1. Операция исключающее или **XOR** в точности соответствует сложению, у которого игнорируется перенос.

С помощью логических команд можно выделить отдельные биты в байте или слове так, что они могут быть установлены, сброшены, проверены. Для выделения битов эти команды используют маску. Роль маски играет второй операнд *op*2. Значение маски используется побитно. С помощью битов маски выделяются нужные для конкретной операции биты первого операнда.

Для установки какого-либо одного бита операнда в 1 нужно использовать оператор OR. В этом случае все значения маски – нули, кроме единицы на месте устанавливаемого бита. Команда OR над маской и другим операндом устанавливает 1 в выбранном бите, а другие биты результата оставляет неизменными. Аналогично, оператор AND может сбросить отдельные биты в 0. В этом случае в маске все разряды единичные, кроме разрядов, соответствующих сбрасываемым битам, который имеет значение 0. Этот бит сбросится в 0, а остальные останутся без изменений.

Оператор **TEST** работает с двумя операндами:

TEST *op*1, *op*2

Он действует аналогично оператору AND, но устанавливает только флаги в регистре флагов, а значения операндов не изменяет.

Примеры логических команд:

```
NOT   BX
AND   AL, 10110101b
OR    CX, 0F0h
XOR   DX, DX
TEST  BL, 11110000b
```

Команды сдвига

Команды сдвига осуществляют сдвиг содержимого регистра или ячейки памяти влево или вправо на один или несколько битов. Команды содержат два операнда. Первый операнд может быть регистром или ячейкой памяти и является источником данных, над которыми выполняется сдвиг. В нем же сохраняется результат. Второй операнд является счетчиком сдвига и определяет количество сдвигаемых бит. Второй операнд может быть константой или регистром.

Все команды сдвига устанавливают флаг переноса *CF*. По мере сдвига битов за пределы операнда они сначала попадают во флаг переноса, устанавливая его равным значению очередного бита, оказавшегося за пределами операнда. Куда этот бит попадет дальше, зависит от типа команды сдвига и алгоритма программы.

По принципу действия команды сдвига можно разделить на два типа:

- команды линейного сдвига;
- команды циклического сдвига.

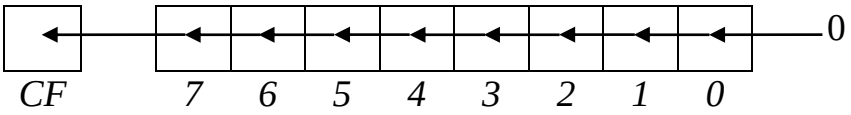
Линейный сдвиг

Команды линейного сдвига делятся на два вида:

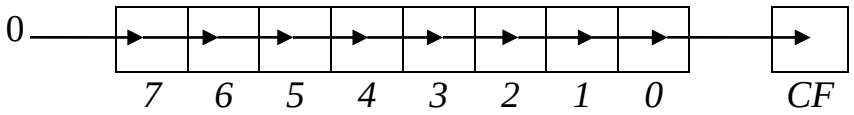
- команды логического линейного сдвига;
- команды арифметического линейного сдвига.

К командам **логического линейного сдвига** относятся следующие:

SHL (*Shift Logical Left*) – логический сдвиг влево. Содержимое первого операнда сдвигается влево на количество битов, определяемое вторым операндом. Справа (в позицию младшего бита) вписываются нули;

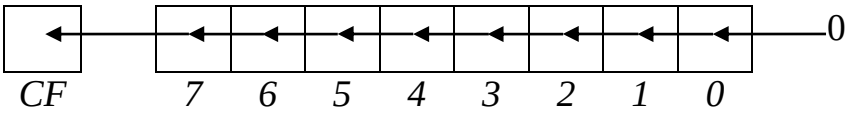


SHR (*Shift Logical Right*) – логический сдвиг вправо. Содержимое первого операнда сдвигается вправо на количество битов, определяемое вторым операндом. Слева (в позицию старшего, знакового бита) вписываются нули.

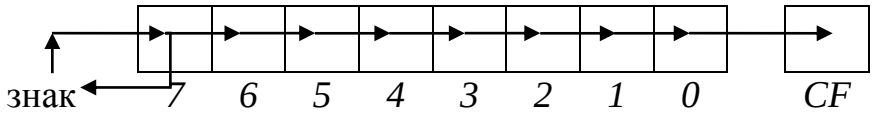


Команды **арифметического линейного сдвига** отличаются от команд логического сдвига тем, что они особым образом работают со знаковым разрядом операнда:

SAL (*Shift Arithmetic Left*) – арифметический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое вторым операндом. Справа (в позицию младшего бита) вписываются нули. Команда SAL не сохраняет знака, но устанавливает в 1 флаг переполнения *OF* при смене знака очередным выдвигаемым битом. В остальном команда SAL полностью аналогична команде SHL;



SAR (*Shift Arithmetic Right*) – арифметический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое вторым операндом. Слева в операнд вписываются нули. Команда SAR сохраняет знак, восстанавливая его после сдвига каждого очередного бита.



Команды арифметического сдвига позволяют выполнять умножение и деление операнда на степени двойки.

Примеры операторов линейного сдвига:

```
SHL   AX, 1
SHR   X, CL
SAL   BL, 1
SAR   Y, CL
```

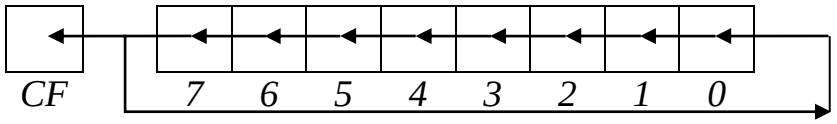
Циклический сдвиг

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига:

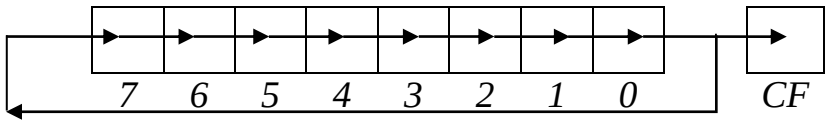
- команды простого циклического сдвига;
- команды циклического сдвига через флаг переноса *CF*.

К командам простого циклического сдвига относятся:

ROL (*Rotate Left*) – циклический сдвиг влево. Содержимое операнда сдвигается влево на количество бит, определяемое вторым операндом. Сдвигаемые влево биты записываются в тот же операнд справа;



ROR (*Rotate Right*) – циклический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое вторым операндом. Сдвигаемые вправо биты записываются в тот же операнд слева.

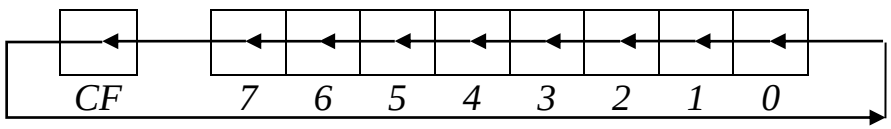


В процессе исполнения команд простого циклического сдвига крайний сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становится значением флага *CF*.

Команды циклического сдвига через флаг переноса *CF* отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а записывается сначала во флаг переноса *CF*. Лишь следующее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита в другой конец операнда.

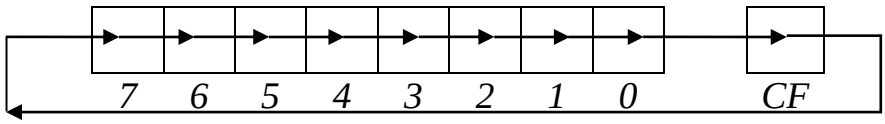
К командам циклического сдвига через флаг переноса *CF* относятся следующие:

RCL (*Rotate through Carry Left*) – циклический сдвиг влево через перенос. Содержимое операнда сдвигается влево на количество бит, определяемое второго операнда. Сдвигаемые биты поочередно становятся значением флага переноса *CF*;



RCR (*Rotate through Carry Right*) – циклический сдвиг вправо через перенос. Содержимое операнда сдвигается вправо на количество бит, определяемое значени-

ем второго операнда. Сдвигаемые биты поочередно становятся значением флага переноса *CF*.



Примеры операторов циклического сдвига:

```
ROL   AX, 1
ROR   X, CL
RCL   BL, 1
RCR   Y, CL
```

Лекция 10

Команды передачи управления

Процессор обычно исполняет операторы друг за другом, в том порядке, как они были описаны в исходном тексте программы. Команды передачи управления позволяют нарушить этот порядок, заставив процессор обойти некоторый участок программы, перейти на выполнение другой ветви или передать управление подпрограмме. Все эти операции осуществляются с помощью команд переходов.

Переходы разделяются на:

- **безусловные** – передача управления в другую точку программы осуществляется всегда, независимо ни от каких обстоятельств;
- **условные** – осуществляются или нет, в зависимости от тех или иных условий.

Безусловные переходы подразделяются на **собственно переходы** (без возврата в точку перехода) и **вызовы подпрограмм** (с возвратом после завершения подпрограммы).

Безусловные переходы

Команды безусловного перехода могут быть **прямые** или **косвенные.**

Оператор **прямого** безусловного перехода имеет вид:

JMP [*модификатор*] *Метка*

здесь *Метка* – метка оператора, которому будет передано управление;

модификатор:

short – короткий переход на метку внутри текущего сегмента кода. В команде перехода используется смещение на точку перехода размером в 1 байт. Точка перехода должна располагаться не дальше, чем в -128 до 127 байтах от данного оператора.

near ptr – ближний переход на метку внутри текущего сегмента кода. В команде перехода используется смещение на точку перехода размером в 2 байта.

far ptr – дальний переход на метку в другом сегменте кода. В команде указывается адрес сегмента и смещение на точку перехода.

Примеры:

```
jmp    short met1
jmp    near ptr met2
jmp    far ptr met3
```

В операторах **косвенного** безусловного перехода указывается не сам адрес перехода, а место, откуда его можно получить:

JMP [*модификатор*] *Адрес_перехода*

модификатор:

word ptr – косвенный переход внутри текущего сегмента кода;

dword ptr – косвенный переход на в другой сегмент кода.

Примеры:

```
.data
addr   dw m1
dw m2
offs   dw 1040h
segm  dw 60A0h

.code
. . .
jmp    dword ptr offs
. . .
mov si, 0
jmp addr[si]
. . .
mov   si, 2
jmp    addr[si]
. . .
m1:   . . .
. . .
```

m2: . . .

Вызовы подпрограмм

В любой программе, независимо от ее содержания, встречаются участки, которые требуется выполнять несколько раз по ходу программы. Такие повторяющиеся участки целесообразно выделить из общей программы, оформить в виде подпрограмм и обращаться к ним каждый раз, когда в основной программе возникает необходимость их выполнения.

Подпрограмма может быть оформлена в виде процедуры, и тогда имя этой процедуры **PName** будет служить точкой входа в подпрограмму:

```
PName PROC      ; Подпрограмма-процедура
. . .           ; Тело подпрограммы
RET             ; Команда возврата в вызывающую программу
PName ENDP
```

Подпрограмма должна завершаться командой **RET**, служащей для возврата управления в ту точку, откуда подпрограмма была вызвана.

Вызов подпрограммы может быть прямым или косвенным и имеет вид:

CALL [*млодификатор*] *Имя_процедуры*

где *модификатор* может принимать значения **near ptr** или **far ptr**.

CALL [*млодификатор*] *Адрес_процедуры*

где *модификатор* может принимать значения **word ptr** или **dword ptr**.

Смысл модификаторов такой же, как и в случае оператора JMP.

Примеры:

```
CALL PName
CALL word ptr [si]
CALL bx
```

При исполнении оператора CALL в стек помещается адрес следующего после CALL оператора (сначала адрес сегмента, затем – смещение). Затем производится переход на указанную процедуру.

При исполнении оператора RET из стека извлекается адрес точки возврата из процедуры (сначала смещение, затем – адрес сегмента) и производится переход по указанному адресу.

Условные переходы

В системе команд микропроцессора 8086 имеется свыше трех десятков команд условных переходов, позволяющих осуществлять переходы при наличии разнообразных условий: равенства, неравенства, положительности или отрицательности результата и т.д. Все операторы условных переходов имеют вид:

Jx Метка

При выполнении этих операторов процессор анализирует содержимое регистра флагов и осуществляет переход на указанную метку в зависимости от состояния отдельных флагов или их комбинаций, удовлетворяющих заданному условию. Если заданное условие не выполняется, то перехода на указанную в операторе метку не происходит, а выполняется следующий по тексту программы оператор. Так, например, оператор JZ представляет команду условного перехода, которая, осуществляет передачу управления только в том случае, если в регистре флагов флаг нуля ZF равен 1. В противном случае выполняется оператор, непосредственно следующий за оператором JZ.

При исполнении команд условного перехода метка перехода должна находиться не далее -127 или +128 байтов от команды условной передачи управления.

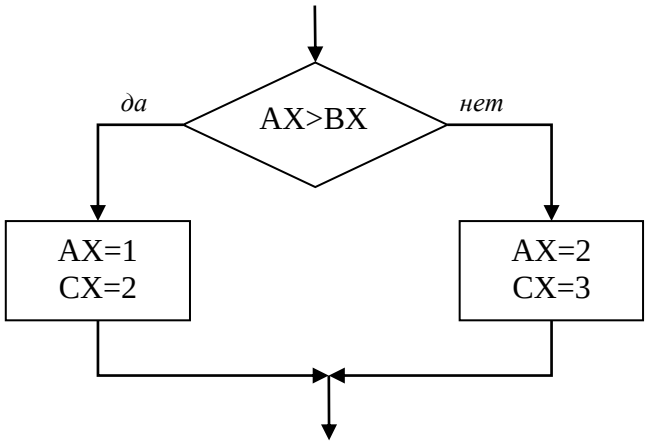
Набор команд процессора 8086 предусматривает большое разнообразие инструкций условных переходов, что позволяет осуществлять переход почти по любому флагу или их комбинации. Можно осуществлять условный переход по состоянию нуля, переноса, по знаку, четности или флагу переполнения и по комбинации флагов, показывающих результаты операций над числами.

Командам условной передачи управления могут предшествовать любые команды, изменяющие состояния флагов, но обычно они используются совместно с командой сравнения CMP (от *compare* – сравнить). Оператор сравнения имеет вид:

CMP *op*1, *op*2

В качестве операнда *op*1 здесь можно использовать регистр или переменную, в качестве операнда *op*2 – регистр, переменную или константу.

Оператор CMP вычитает содержимое операнда *op*2 из содержимого операнда *op*1 и в зависимости от полученного результата устанавливает или обнуляет флаги в регистре флагов процессора. Сам результат вычитания нигде *не сохраняется.* Другими словами, команда CMP не изменяет операнды. Она целиком предназначена для установки значений флагов, на основании которых команды условного перехода будут "принимать решение" о передаче управления.



В качестве примера рассмотрим следующую задачу. Если содержимое регистра *AX* больше содержимого регистра *BX*, то присвоить регистрам *AX* и *CX* соответственно значения 1 и 2. В противном случае присвоить этим регистрам соответственно значения 2 и 3.

Блок-схема алгоритма решения данной задачи представлена на рисунке. Соответствующая программа на языке ассемблера будет иметь вид:

```
        CMP   AX, BX      ; сравниваем содержимое регистров AX и BX
        JG    M1          ; если условие выполняется, то перейти на метку M1
        MOV   AX, 2       ; в противном случае присвоить регистрам AX и CX
        MOV   CX, 3       ; значения 2 и 3 (правая ветвь блок-схемы)
        JMP   M2          ; перейти на метку M2
M1:     MOV   AX, 1       ; присвоить регистрам AX и CX
        MOV   CX, 2       ; значения 1 и 2 (левая ветвь блок-схемы)
M2:     . . .
```

Если метка перехода находится далее чем –127 или +128 байтов от команды условной передачи управления, то необходимо использовать оператор условного перехода в комбинации с оператором безусловного перехода, у которого нет таких ограничений, например:

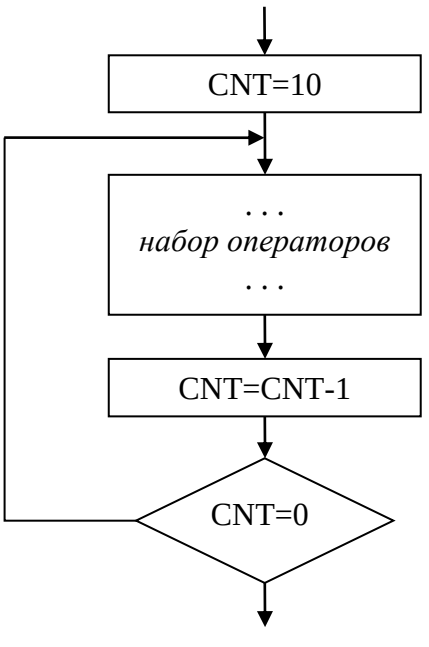
```
        CMP  AX, 5
        JNZ  M1
        JMP  M2
M1:     . . .
        . . .
M2:     . . .
```

Поскольку на состояние регистра флагов влияют многие команды процессора, командами условных переходов можно пользоваться не только после команд сравнения или анализа, но и после многих других команд, если внимательно изучить влияние этих команд на флаги процессора.

Организация циклов

Система команд процессора 8086 предполагает два способа организации циклов: с помощью команд условных переходов или с помощью специальных операторов организации цикла.

Пусть в программе необходимо выполнить некоторый набор операторов ровно 10 раз. Для решения данной задачи обычно вводится переменная или назначается регистр, используемые в качестве счетчика циклов. Первоначально счетчику циклов присваивается значение, равное количеству исполняемых циклов (в нашем случае 10). Затем при каждом очередном выполнении цикла из текущего значения счетчика вычитается единица. Это повторяется до тех пор, пока значение счетчика не станет равным нулю. После этого выполнение цикла прекращается. Блок схема данного алгоритма представлена на рисунке.



Программа имеет следующий вид:

```
        MOV   CNT, 10
CYCLE:  . . .
        набор операторов
        . . .
```

```
        DEC   CNT
        CMP   CNT, 0
        JNZ   CYCLE
```

В системе команд процессора 8086 существует специальная команда организации цикла. В ассемблере эта команда представлена оператором:

LOOP *Метка*

В качестве счетчика циклов данный оператор использует регистр CX. Оператор LOOP вычитает из текущего содержимого регистра CX единицу и сохраняет результат в том же регистре. Если полученный результат не равен 0, то осуществляется переход на указанную метку. Если результат равен нулю, то переход на метку не производится, а выполняется следующий по тексту программы оператор. Фактически оператор LOOP производит те же действия, что и три последних оператора из предыдущего примера при условии, что в качестве счетчика циклов выступает регистр CX. Организация цикла с помощью оператора LOOP будет иметь вид:

```
        MOV   CX, 10
CYCLE:  . . .
        набор операторов
        . . .
        LOOP  CYCLE
```

Оператор организации цикла LOOP реализует только короткие переходы (от –128 до +127 байт). Для работы с длинными циклами необходимо использовать операторы условных и безусловных переходов.

Лекция 11

Команды обработки строк (цепочечные команды)

Это команды, которые позволяют производить действия над блоками памяти, представляющими собой последовательности элементов типа байт, слово или двойное слово.

Для всех команд обработки строк используются следующие соглашения.

Адрес ячейки, являющейся *источником* данных, всегда размещается в паре регистров *DS:SI* **([DS:SI] → источник данных**).

Адрес ячейки, являющейся *приёмником* данных, всегда размещается в паре регистров *ES:DI* **([ES:DI] → приёмник данных**).

Направление обработки данных (от меньших адресов к большим или наоборот) определяется состоянием флага направления DF регистра флагов.

а) если флаг направления *DF*=0, то значения индексных регистров увеличивается (направление обработки – в сторону *увеличения* адресов);

б) если флаг направления *DF*=1, то значения индексных регистров уменьшается (направление обработки – в сторону *уменьшения* адресов).

Для сброса или установки флага направления *DF* регистра флагов используются команды:

cld	$DF \leftarrow 0$;
lstd	$DF \leftarrow 1$;

1. Загрузка элемента из строки

lods ***адрес_источника***

lodsb	$AL \leftarrow [DS:SI]$; $SI \leftarrow SI \pm 1$
lodsw	$AX \leftarrow [DS:SI]$; $SI \leftarrow SI \pm 2$

2. Сохранение элемента в строке

stos ***адрес_приемника***

stosb	[*ES:DI*] ← *AL*; *DI* ← *DI* ± 1
stosw	[*ES:DI*] ← *AX*; *DI* ← *DI* ± 2

3. Пересылка строк

movs ***адрес_приемника, адрес_источника***

movsb	[*ES:DI*] ← [*DS:SI*]; *SI* ← *SI* ± 1; *DI* ← *DI* ± 1
movsw	[*ES:DI*] ← [*DS:SI*]; *SI* ← *SI* ± 2; *DI* ← *DI* ± 2
movsd	[*ES:DI*] ← [*DS:SI*]; *SI* ← *SI* ± 4; *DI* ← *DI* ± 4

4. Сравнение строк

cmps ***адрес_приемника, адрес_источника***

cmpsb	*регистр флагов* ← [*DS:SI*] – [*ES:DI*]; *SI* ← *SI* ± 1; *DI* ← *DI* ± 1
cmpsw	*регистр флагов* ← [*DS:SI*] – [*ES:DI*]; *SI* ← *SI* ± 2; *DI* ← *DI* ± 2

5. Сканирование строк

scas ***адрес_приемника***

scasb	*регистр флагов* ← *AL* – [*ES:DI*]; *DI* ← *DI* ± 1
scasw	*регистр флагов* ← *AX* – [*ES:DI*]; *DI* ← *DI* ± 2

Префиксы повторения

rep – заставляет команду повторятся до тех пор, пока содержимое регистра *CX* не станет равным 0. После каждого повторения содержимое *CX* уменьшается на 1;

repe или **repz** – повторяют команду, пока *CX* не равен 0 и флаг нуля регистра флагов *ZF*=1. После каждого повторения содержимое *CX* уменьшается на 1;

repne или **repnz** – повторяют команду, пока *CX* не равен 0 и флаг нуля регистра флагов *ZF*=0. После каждого повторения содержимое *CX* уменьшается на 1.

Примеры.

1. Копирование строк

```
.MODEL SMALL
.STACK 100
.DATA
Source   db  'Строка символов'
Dest     db  15 DUO (?)

.CODE
mov      AX, @DATA
mov      DS, AX
mov      ES, AX
lea      SI, Source
lea      DI, Dest
mov      CX, 15
```

M: mov AL, [SI] mov {DI], AL inc SI inc DI loop M	cld M: movsb loop M	cld rep movsb

2. Сравнение строк

```
.MODEL SMALL
.STACK 100
.DATA
String1  db  'Строка1'
String2  db  'Строка2'

.CODE
mov      AX, @DATA
mov      DS, AX
lea      SI, String1
lea      DI, String2
mov      CX, 7
```

M1:	mov	AL, DS:[SI]	cld	
	cmp	AL, ES:[DI]	repe	cmpsb
	jnz	M2	cmp	CX, 0
	loop	M1	jz	M2

```
        – – –;        если строки не совпадают
        – – –;
M2:     – – –;        если строки совпадают
```

3. Поиск символа в строке

```
.MODEL SMALL
.STACK 100
.DATA
String    db  'Строка'

.CODE
mov      AX, @DATA
mov      DS, AX
```

```
        lea     DI, String
        mov     AL, 'a'
        mov     CX, 6
        repne   scasb
        cmp     CX, 0
        je      FND
        – – –;      если символ не найден
        – – –;
FND:    – – –;      если символ найден
```

Лекция 12

Прерывания

В архитектуре процессоров 80x86 предусмотрены особые случаи, когда процессор прекращает (прерывает) выполнение текущей программы и немедленно передает управление программе-обработчику, специально написанной для обработки подобной ситуации. Такие особые ситуации называются прерываниями. Прерывания делятся на два типа: аппаратные (внешние) и программные (внутренние), в зависимости от того, вызвало ли эту ситуацию какое-нибудь внешнее устройство или выполняемая процессором команда.

Необходимость прерываний обусловлено двумя основными причинами: преднамеренный запрос таких действий, как операции ввода-вывода на различные устройства и непредвиденные программные ошибки (например, переполнение при делении).

Обработка прерываний (как внешних, так и внутренних) в реальном режиме микропроцессора 8086 производится в три этапа:

1. Прекращение выполнения текущей программы.
2. Переход к выполнению и выполнение программы обработки прерываний.
3. Возврат управления прерванной программе.

Подобно вызову процедуры прерывание заставляет микропроцессор 8088 сохранить в стеке информацию для последующего возврата, а затем перейти к группе команд, находящейся в некоторой ячейке памяти. Но при вызове процедуры микропроцессор 8088 исполняет процедуру, а при прерывании – *программу обработки прерывания.*

Прерывание всегда вызывает косвенный переход к своей программе обработки за счет получения ее адреса из *вектора прерывания* – четырехбайтовой ячейки памяти. Вектор прерывания содержит физический адрес программы обработки прерывания (в первых двух байтах находится смещение, а в последних двух байтах – адрес сегмента). Вектор прерывания хранится в специальной области памяти, которая располагается в самом начале оперативной памяти и называется *таблицей векторов прерываний*. Она содержит адреса 256 обработчиков прерываний. Так как

каждый из них имеет длину 4 байта, то все они занимают первые 1К байтов, то таблица векторов прерываний занимает абсолютные адреса от 0 до 3FFh. Кроме того, вызовы процедуры сохраняют в стеке только адрес, а прерывания сохраняют еще и флаги (так же, как команда PUSHF).

Команда INT (*interrupt* – прерывать) имеет формат

INT *тип_прерываиия*

где *тип_ прерывания* – номер, идентифицирующий один из 256 различных векторов, находящихся в памяти.

При исполнении команды INT микропроцессор 8088 производит следующие действия:

1. Помещает в стек регистр флагов.

2. Обнуляет флаг трассировки TF и флаг включения-выключения прерываний IF для исключения пошагового режима исполнения команд и блокировки других маскируемых прерываний.

3. Помещает в стек значение регистра CS.

4. Вычисляет адрес вектора прерывания, умножая *тип_прерывания* на 4.

5. Загружает второе слово вектора прерывания в регистр CS.

6. Помещает в стек значение указателя команд IP.

7. Загружает в указатель команд IP первое слово вектора прерывания.

Итак, после исполнения команды INT в стеке окажутся значения регистра флагов и регистров CS и IP, флаги TF и IF будут равны 0, а пара регистров CS:IP будет указывать на начальный адрес программы обработки прерывания. Затем микропроцессор 8088 начнет исполнять эту программу.

Например, команда

INT 1Ah

заставит микропроцессор 8088 вычислить адрес вектора 68H (4*1 AH). Следовательно, он получит 16-битовые значения регистров IP и CS, отвечающие программе обработки прерывания, из ячеек 68h и 6Ah соответственно.

Для управления аппаратными прерываниями во всех типах IBM PC используется микросхема программируемого контроллера прерываний. Поскольку в каждый момент времени может поступить не один запрос, микросхема имеет схему приоритетов. Имеется 16 уровней приоритетов и обращения к соответствующим уровням обозначаются сокращениями от IRQ0 до IRQ15, что означает запрос на прерывание. Максимальный приоритет соответствует уровню 0. Запросы на прерывание 0-7 соответствуют векторам прерываний от 8H до 0FH; запросы на прерывания 8-15 обслуживаются векторами от 70H до 77H. Ниже приведены назначения этих прерываний:

Аппаратные прерывания в порядке приоритета.

IRQ	*№ преры-вания*	*Наименование*
0	8	таймер
1	9	клавиатура
2	0AH	канал ввода/вывода
8	70H	часы реального времени (только AT)
9	71H	программно переводятся в IRQ2 (только AT)
10	72H	резерв
11	73H	резерв
12	74H	резерв
13	75H	мат. сопроцессор (только AT)
14	76H	контроллер жесткого диска (только AT)
15	77H	резерв
3	0BH	COM1 (COM2 для AT)
4	0CH	COM2 (модем для PCjr, COM1 для AT)
5	0DH	жесткий диск (LPT2 для AT)
6	0EH	контроллер дискет
7	0FH	LPT1

Прерыванию времени суток дан максимальный приоритет, поскольку если оно будет постоянно теряться, то будут неверными показания системных часов. Прерывание от клавиатуры вызывается при нажатии или отпускании клавиши; оно вызывает цепь событий, которая обычно заканчивается тем, что код клавиши помещается в буфер клавиатуры (откуда он затем может быть получен программными прерываниями).

Выполнение прерываний зависит от значения флага прерывания (бит 9) в регистре флагов. Когда этот бит равен 0, то все прерывания разрешены. Когда он равен 1, то все аппаратные прерывания запрещены. Чтобы запретить прерывания, установив этот флаг в 1, используется инструкция CLI. Для очистки этого флага и восстановления прерываний – инструкция STI. Избегайте отключения прерываний на длительный период. Прерывание времени суток происходит 18.2 раза в секунду и если к этому прерыванию был более чем один запрос в то время, когда аппаратные прерывания были запрещены, то лишние запросы будут отброшены и системное время будет определяться неправильно.

Компьютер автоматически запрещает аппаратные прерывания при вызове программных прерываний и автоматически разрешает их при возврате.

Возврат из прерывания

Осуществляется с помощью оператора

IRET

По отношению к прерываниям команда IRET (*interrupt return* – возврат после обработки прерывания) играет ту же роль, что и команда RET для вызовов процедур. Она возвращает управление основной программе и позволяет продолжить ее исполнение с того места, где она была прервана. По этой причине команда IRET должна быть последней при исполнении микропроцессором 8088 программы обработки прерывания.

Команда IRET извлекает из стека три 16-битовых значения и загружает их в указатель команд IP, регистр сегмента команд CS и регистр флагов соответствен-

но. Содержимое других регистров может быть уничтожено, если в программе обработки прерывания не предусмотрено их сохранение.

Операционная система MS-DOS

Операционная система представляет собой набор управляющих программ, которые руководят и управляют работой компьютера. Вообще операционная система обеспечивает:

- управление памятью;
- управление вычислительным процессом;
- безопасность;
- взаимодействие пользователя с компьютером.

Дисковая операционная система фирмы Microsoft - MS-DOS, является традиционной микрокомпьютерной операционной системой, которая состоит из пяти больших компонент:

- Загрузчик операционной системы
- Ядро MS-DOS
- BIOS MS-DOS
- Командный процессор
- Вспомогательные обслуживающие программы

Загрузчик операционной системы

Загрузчик операционной системы устанавливает операционную систему с диска в оперативную память.

Полный процесс загрузки, называемый начальной загрузкой (дословно – *bootstrapping* – шнурование ботинок), часто сложен и может вовлекать в “работу” несколько загрузчиков. (Термин bootstrapping вошел в обиход, потому что каждый уже загруженный уровень системы “подтягивает” следующую часть системы, все равно как зашнуровывание ботинка). Например, в большинстве микрокомпьютерных разработок, основанных на стандартах MS-DOS, загрузчик, записанный в ПЗУ, являющийся первой исполняемой микрокомпьютером программой во время вклю-

чения компьютера или при перезагрузке, считывает дисковый начальный загрузчик из первого (*boot*) сектора диска первоначальной загрузки и исполняет его. Дисковый начальный загрузчик, в свою очередь, считывает основную порцию MS-DOS – MSDOS.SYS и IO.SYS – с последовательных дисковых файлов в память.

Ядро MS-DOS

Ядро MS-DOS – это сердце MS-DOS, которое обеспечивает функционирование всех функций, характерных для традиционных операционных систем. Ядро содержится в единственном файле – MSDOS.SYS, поставляемом корпорацией Microsoft. Функции, предоставляемые ядром прикладным программам, – называемые системными функциями – аппаратно независимы. Сервисные возможности, обеспечиваемые ядром MS-DOS для прикладных программ, включают:

Управление процессом

Управление процессом (или задачей) включает: загрузку программы, выполнение задачи, завершение задачи, планирование задачи, межзадачную связь.

Управление памятью

Так как количество памяти, требуемое программе, изменяется от программы к программе, традиционные операционные системы обычно обеспечивают функции управления памятью. Требования на размер памяти могут также меняться в процессе выполнения программы. Кроме того управление памятью особенно необходимо когда две или более программ находятся в памяти в одно и тоже время.

Обслуживание внешних устройств

Операционная система обеспечивает программам связь с внешними устройствами посредством набора обращений к операционной системе, которые транслируются операционной системой в вызовы соответствующего драйвера устройства.

Потребность прикладной программы в периферии не должна касаться подробностей периферийных устройств или каких-либо специальных свойств, которые имеет периферийное устройство.

Файловая система

Файл – это именованная область памяти на каком-либо физическом носителе (обычно на диске), в которой может храниться исходный текст программы, какое-либо из ее промежуточных представлений, программа в машинном коде, готовая к исполнению, или данные для ее работы. В файлах могут содержаться любые текстовые документы и числовые данные, закодированная табличная, графическая и любая другая информация.

Файловая система является одной из наибольших частей операционной системы. Файловая система является надстройкой над носителем данных блочного устройства (обычно это дисковод гибких магнитных дисков или винчестер), которая преобразовывает структуру каталога и файлы в физическую единицу памяти. Файловая система на диске содержит, как минимум, информацию о размещении файлов, каталог и объем файлов.

В соответствии с характером хранимой информации файлу обычно приписывают тип (расширение). Задание типа осуществляет либо сам пользователь, либо программа, порождающая файл. Имя и тип используются совместно для идентификации файла. По типу файла можно судить о его содержимом. Имя может состоять из 1-8 букв, цифр и знаков, (“минус”, “подчерк”), расположенных в произвольном порядке. Тип может состоять из 1-3 букв или цифр (а также некоторых других символов) или отсутствовать вообще.

Полное имя файла образуется из двух слов - имени и типа, разделяемых знаком “точка” (при отсутствии типа точка необязательна); поэтому тип иногда называют расширением имени. Примеры полных имен файлов: COMMAND.COM, START.BAT, PROG1.ASM, 123.DOC, MYFILE.EXE. При создании файла или изменении его содержимого автоматически регистрируются дата и время, снятые с текущих показаний календаря и часов системы.

BIOS операционной системы MS-DOS

BIOS (*Base Input-Output System* – базовая система ввода-вывода), является уровнем операционной системы, который находится между ядром операционной системы и аппаратными средствами. Прикладные программы осуществляют ввод и

вывод путем выдачи требований к ядру операционной системы, которая, в свою очередь, вызывает подпрограммы BIOS операционной системы MS-DOS, которые непосредственно осуществляют доступ к аппаратным средствам. Такое разделение функций позволяет писать прикладные программы в аппаратно-независимой манере.

BIOS операционной системы MS-DOS состоит из некоторой программы инициализации и набора **драйверов** устройств (*драйвер* устройства – это специализированная программа, которая обеспечивает поддержку специфического устройства, такого как дисплей или последовательный порт). Драйверы устройств отвечают за доступ к аппаратным средствам и за обслуживание прерываний, что позволяет соответствующим устройствам сигнализировать микропроцессору о том, что они нуждаются в обслуживании.

Командный процессор

Пользовательский интерфейс для операционной системы, также называемый командным процессором, является, в основном, обычной программой, которая позволяет пользователю взаимодействовать с операционной системой. Используемый в MS-DOS, по умолчанию, пользовательский интерфейс является заменяемой управляющей программой и называется COMMAND.COM.

Одной из основных задач командного процессора является загрузка по требованию программы в память и передача управления от системы программе, с целью выполнения этой программы. Когда выполнение программы завершается управление передается командному процессору, который предлагает пользователю следующую команду. Кроме того, командный процессор обычно включает функции поддержки и вывода на экран каталогов файлов.

Команды COMMAND.COM делятся на три категории:

- Внутренние команды
- Командные файлы
- Внешние команды

Внутренние команды являются программами, содержащимися в COMMAND.COM и включают операции типа COPY или ERASE. Внутренние ко-

манды состоят из ключевого слова, сопровождаемого иногда списком параметров, уточняющих действие команды.

Командные файлы являются обычными текстовыми файлами, которые содержат внутренние команды, внешние команды, директивы командных файлов и невыполняемые комментарии.

Внешние команды, которые фактически являются выполняемыми программами, хранятся в отдельных файлах с расширениями .COM или .EXE и включаются в дистрибутивные диски MS-DOS. Эти программы вызываются по имени файла без расширения.

Вспомогательные программы

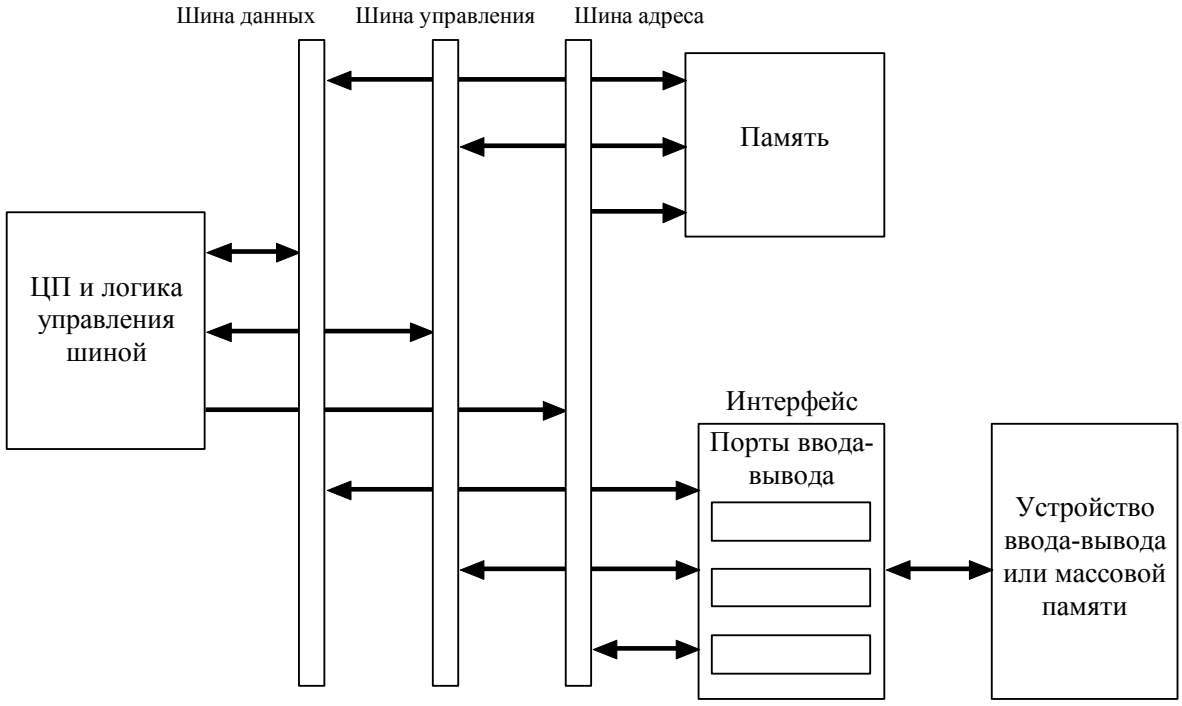
Программное обеспечение MS-DOS включает вспомогательные программы, которые обеспечивают доступ к средствам операционной системы и которые не являются резидентными командами командного процессора, встроенными в COMMAND.COM. Так как такие программы хранятся на диске как исполняемые файлы, то они являются, в сущности, такими же, что и прикладные программы и MS-DOS загружает и выполняет их как любую прикладную программу.

Вспомогательные программы, обеспечиваемые MS-DOS, часто рассматриваемые как внешние команды, включают дисковые утилиты, такие как FORMAT и CHKDSK и более общие вспомогательные программы, такие как EDIT (текстовый редактор) и PRINT (утилита, позволяющая распечатывать файлы).

Лекция 13

Взаимодействие с внешними устройствами

Рассмотрим способы передачи информации между внешними (периферийными) устройствами и процессором или памятью. На рисунке показана базовая архитектура вычислительной системы. Все периферийные устройства и внешняя память подключаются к системной шине через интерфейсы. Каждый интерфейс имеет набор регистров, называемых портами ввода-вывода, через которые процессор и память взаимодействуют с внешним устройством. Одни порты предназначены для буферизации данных, другие – для хранения информации о состоянии устройства и интерфейса, третьи – для восприятия приказов от процессора, управляющих действиями интерфейса и устройства. Все взаимодействие с внешним миром осуществляется через порты ввода-вывода в интерфейсе. Следовательно, в процессоре должны быть средства для передачи информации в(из) порты(ов), а также в(из) память(и).



Архитектура вычислительной системы

В некоторых компьютерах адреса памяти и портов включены в единое адресное пространство и все команды с обращением к памяти могут обращаться и к портам. В других компьютерах, например в системах на базе микропроцессора 8086,

допускается организация двух адресных пространств: памяти и ввода-вывода. Для этого в шине управления предусматриваются управляющие линии, показывающие, к какому пространству относится адрес на шине адреса. Чтобы выдать правильные сигналы на управляющие линии, система с раздельными пространствами памяти и ввода-вывода должна иметь специальные команды для взаимодействия с портами ввода-вывода.

Ввод из процессора в управляющий или буферный порт осуществляется выдачей адреса на *шину адреса* и соответствующих сигналов на *шину управления* с последующей выдачей данных на *шину данных*. Ввод из входного порта реализуется выдачей адреса и управляющих сигналов на соответствующие шины и ожиданием реакции интерфейса путем выдачи содержимого адресного порта на шину данных. Следует подчеркнуть, что адреса ассоциируются с портами, а не с интерфейсами. Если интерфейс имеет четыре порта, он должен воспринимать четыре адреса.

Существуют следующие способы организации ввода-вывода:

- программный ввод-вывод;
- ввод-вывод по прерываниям;
- блоковые передачи (прямой доступ к памяти).

Передачу данных в порт или из порта можно осуществить двумя способами. Первый способ – выполнить команду, которая передает один байт один байт или одно слово, а второй - выполнить последовательность команд, которая заставляет специальную компоненту, связанную с интерфейсом, передать совокупность байт или слов в(из) указанный(ого) блок(а) ячеек памяти. Второй способ называется блоковой передачей или прямым доступом к памяти (ПДП). Для его поддержки необходима специальная компонента, называемая контроллером прямого доступа к памяти. Байт или слово передаются между портом и процессором, а блоковые передачи осуществляются непосредственно в память или из памяти.

При программном вводе-выводе и вводе-выводе по прерываниям данные от памяти к портам и наоборот передаются через регистры центрального процессора. Например, если необходимо ввести слово из порта в ячейку памяти, его приходится вначале ввести в регистр процессора, а затем переслать в ячейку памяти. Для пере-

дачи блока байт или слов из периферийного устройства в память программа должна последовательно вводить байты или слова в процессор, а потом помещать их в смежные ячейки памяти. Эти операции реализуются программным циклом. Блоковым вводом обладает интерфейс и контроллер ПДП. В этом случае каждые байт и или слово по мере их появления передаются из порта непосредственно в память. Для инициирования передачи программа должна только выдать необходимые приказы в интерфейс и контроллер. Аналогичным образом интерфейс и его контроллер ПДП могут воспринимать последовательные байты или слова из памяти и передавать их во внешнее устройство.

Как система узнает, когда интерфейс имеет данные или готов принимать данные? При программном вводе-выводе программа определяет требующие обслуживания интерфейсы, проверяя биты "готовность" в их регистрах состояния. Программная проверка разрядов или сигналов готовности называется опросом. При вводе-выводе по прерываниям интерфейс посылает в процессор внешнее прерывание, когда он имеет данные для ввода или готов воспринимать их, а сама операция ввода-вывода реализуется процедурой прерывания. В операции ПДП интерфейс запрашивает использование шины, выдавая сигнал по линии управления, и осуществляет необходимую передачу без помощи процессора.

В микропроцессоре 8086 программное взаимодействие с портами ввода-вывода осуществляется командами ввода IN и вывода OUT.

Название	*Мнемоника и формат*	*Описание*
Ввести:		
длинная форма, байт	IN AL, PORT	(AL) ← (PORT)
длинная форма, слово	IN AX, PORT	(AX) ← (PORT+1 : PORT)
короткая форма, байт	IN AL, DX	(AL) ← ((DX))
короткая форма, слово	IN AX, DX	(AX) ← ((DX)+1 : (DX))
Вывести:		

длинная форма, байт длинная форма, слово короткая форма, байт короткая форма, слово	OUT PORT, AL OUT PORT, AX OUT DX, AL OUT DX, AX	(PORT) ← (AL) (PORT+1 : PORT) ← (AX) ((DX)) ← (AL) ((DX)+1 : (DX)) ← (AX)

Обе команды передают байт или слово и имеют длинный и короткий форматы. Первым операндом в команде IN, т.е. операндом-получателем, должен быть регистр AL (передается байт) или регистр AX (передается слово). Как и при обращении к памяти, слово передается из двух смежных адресов, причем младший байт передается в AX из порта с младшим адресом. Если второй операнд в команде IN оказывается константой, она служит адресом порта, содержимое которого вводится в регистр. Когда вторым операндом указан регистр DX, в качестве адреса порта используется содержимое DX. В длинном формате команд IN и OUT адрес порта должен находиться в диапазоне 0000h...00FFh, а в коротком формате – в диапазоне 0000h...FFFFh.

Примеры команд ввода-вывода:

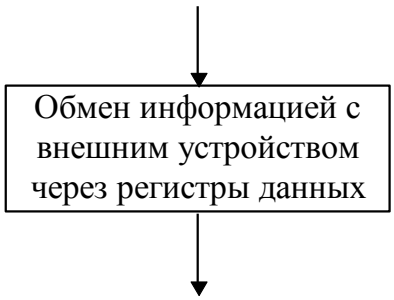
```
IN    AX, 28h
IN    AL, 27h
OUT   26h, AX
OUT   25h,  AL
```

Программно-управляемый ввод-вывод

Существует два типа программно-управляемого ввода-вывода: синхронный обмен и асинхронный обмен.

Синхронный обмен представляет собой наиболее простой способ обмена информации с внешними устройствами. Он обеспечивает передачу информации за одну машинную команду. Схематическое изображение этого способа представлено на рисунке. Центральный процессор, выполняя команду, связанную с передачей ин-

формации, является инициатором обмена, как при вводе, так и при выводе. Внешнее устройство играет пассивную роль, передавая или принимая информацию по командам процессора. Этот способ обмена предполагает близость значений скорости обмена, определяемых программой и быстродействием процессора, и скорости, с которой может производить обмен внешнее устройство. Однако, как правило, эти скорости существенно различаются, а синхронный способ не располагает средствами их синхронизации.

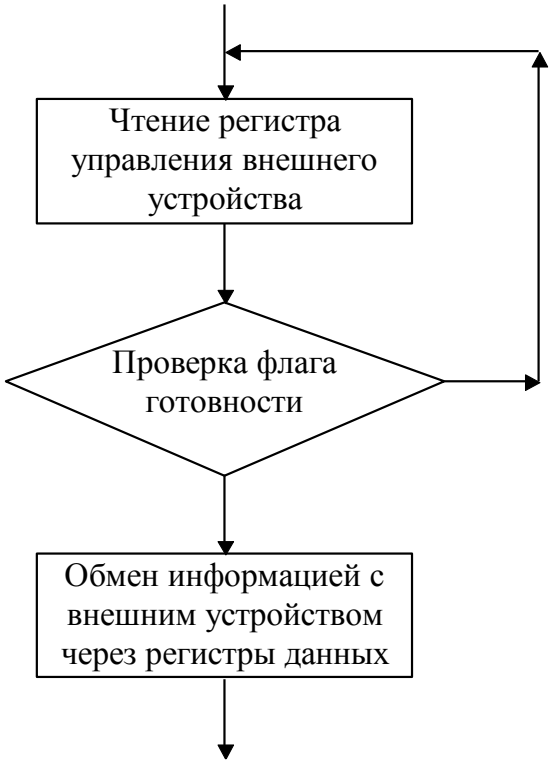


Синхронный обмен

Синхронный способ обмена программируется с помощью стандартных команд ввода-вывода: IN и OUT. Рассмотрим, к чему приводит синхронный способ обмена при использовании медленных внешних устройств, например, принтера. При выводе информации из ЭВМ на принтер большая часть ее не будет напечатана из-за низкой скорости печати в сравнении с высокой скоростью вывода информации из центрального процессора.

Асинхронный обмен позволяет программно синхронизировать обмен между ЭВМ и низкоскоростным внешним устройством. Признаком готовности внешнего устройства к обмену служит определённое состояние (например 1) специального разряда в регистре управления интерфейса внешнего устройства, который называется *битом* или *флагом готовности.* Процессор циклически производит опрос регистра состояния интерфейса внешнего устройства и проверяет состояние флага готовности до тех пор, пока он не установится в заданное состояние, например 1. Этот процесс называется циклом ожиданием готовности. Установка флага готовности осуществляется внешним устройством, когда оно готово выдать или принять информацию. В этом случае производится обмен данными с внешним устройством.

При использовании программного ввода-вывода для нескольких устройств придется опрашивать по очереди биты готовности каждого из этих устройств.



Асинхронный ввод-вывод

В случае асинхронного обмена внешнее устройство играет пассивную роль, предоставляя для чтения или записи регистр данных и принимая или передавая данные для обмена. Обмен производится по командам центрального процессора, внешнее устройство не может быть инициатором обмена. Другим существенным недостатком этого способа является непроизводительная трата времени, особенно при взаимодействии с низкоскоростными внешними устройствами, поскольку большую часть времени процессор тратит на ожидание готовности их к обмену.

Написание собственного прерывания.

Потребность в написании собственного прерывания может возникнуть при необходимости полной или частичной замены одной из процедур операционной системы своей собственной, либо для обеспечения обмена данными с внешним устройством.

Функция 25H прерывания 21H устанавливает вектор прерывания на указанный адрес. Адреса имеют размер два слова. Старшее слово содержит

значение сегмента (CS), младшее содержит смещение (IP). Чтобы установить вектор, указывающим на одну из Ваших процедур, нужно поместить сегмент процедуры в DS, а смещение в DX (следуя порядку нижеприведенного примера). Затем поместите номер прерывания в AL и вызовите функцию. Любая процедура прерывания должна завершаться не обычной инструкцией RET, а IRET. IRET выталкивает из стека три слова, включая регистр флагов, в то время как RET помещает на стек только два. Если вы попытаетесь тестировать такую процедуру как обычную процедуру, но кончающуюся IRET, то Вы исчерпаете стек. Отметим, что функция 25H автоматически запрещает аппаратные прерывания в процессе изменения вектора, поэтому не существует опасности, что посреди дороги произойдет аппаратное прерывание, использующее данный вектор.

```
;---установка прерывания
PUSH    DS              ; сохраняем DS
LEA     DX, ROUT        ; смещение для процедуры в DX
MOV     AX, SEG ROUT    ; сегмент процедуры
MOV     DS, AX          ; помещаем в DS
MOV     AH, 25H         ; функция установки вектора
MOV     AL, 60H         ; номер вектора
INT     21H             ; меняем прерывание
POP     DS              ; восстанавливаем DS
```

Когда программа завершается, должны быть восстановлены оригинальные вектора прерываний. В противном случае последующая программа может вызвать данное прерывание и передать управление на то место в памяти, в котором Вашей процедуры уже нет. Функция 35 прерывания 21H возвращает текущее значение вектора прерывания, помещая значение сегмента в ES, а смещение в BX. Перед установкой своего прерывания получите текущее значение вектора, используя эту функцию, сохраните эти значения, и затем восстановите их с помощью функции 35H (как выше) перед завершением своей программы.

Например:

```
;---в сегменте данных:
KEEP_OFFS DW 0        ; хранит смещение прерывания
KEEP_SEGM             DW 0 ; хранит сегмент заменяемого прерывания
;---в начале программы
MOV     AH, 25H           ; функция получения вектора
MOV     AL, 1CH           ; номер вектора прерывания
INT     21H               ; теперь сегмент в ES, смещение в BX
MOV     KEEP_OFFS, BX             ; запоминаем смещение
MOV     KEEP_SEGM, ES             ; запоминаем сегмент
```

Пример программы с обработчиком прерывания от клавиатуры:

```
.MODEL  SMALL
ESC_CODE  equ   1

.STACK      100
.DATA
INT9_OFFS       DW      ?
INT9_SEGM       DW      ?
FLAG            DB      0

.CODE
MAIN    PROC
mov     ax, 3509h
int     21h
mov     INT9_OFFS, bx
mov     INT9_SEGM, es
push    ds
mov     dx, offset KEYB
mov   ax, cs
```

```
mov   ds, ax
mov     ax, 2509h
int     21h
pop     ds
GET_KEY:
cmp     FLAG, 1
jnz     GET_KEY

mov     dx, INT9_OFFS
mov     ds, INT9_SEGM
mov     ax, 2509h
int     21h
mov     ax, 4C00h
int     21h
MAIN    ENDP

KEYB    PROC
in      al, 60h
cmp     al, ESC_CODE
jnz     MET
mov     FLAG, 1
MET:    jmp     dword ptr INT9_OFFS
KEYB    ENDP

END     MAIN
```

Обращение к функциям BIOS и ядра MS DOS

Для вызова функций BIOS и ядра MS DOS используются программные прерывания. В BIOS за каждым из стандартных внешних устройств компьютера закреплен свой номер прерывания, *например*: 10h – дисплей, 13h – гибкий и жесткий диски, 16h – клавиатура, 15h – последовательный порт.

Так как работа каждого внешнего устройства поддерживается не одной, а множеством функций, то перед вызовом соответствующего прерывания необходимо в регистре AH микропроцессора указать номер используемой функции. Некоторые функции требуют указания дополнительных аргументов. Для этого обычно используются регистры микропроцессора. Результат работы функций также возвращается через регистры микропроцессора. Номера функций прерываний BIOS стандартизированы, их описание можно узнать из справочников.

Большинство функций ядра MS DOS вызываются через единое прерывание с номером 21h. Каждой из этих функций также присвоен индивидуальный номер, который должен быть указан в регистре AH перед вызовом прерывания 21h. При необходимости дополнительные входные и выходные аргументы функций передаются через регистры микропроцессора.

Ниже приведены некоторые базовые функции для прерывания INT 21h.

00h Завершение программы.

01h Ввод символа с клавиатуры с эхом на экран.

02h Вывод символа на экран.

09h Вывод строки символов на экран.

0Ah Ввод с клавиатуры с буферизацией.

0Bh Проверка наличия ввода с клавиатуры.

13h Удаление файла с диска.

14h Последовательное чтение файла.

15h Последовательная запись файла.

16h Создание файла.

2Ah Получение даты.

2Bh Установка даты.

2Ch Получение времени.

2Dh Установка времени.

Подробное описание функций можно найти в справочниках.

Резидентные программы

Программы, остающиеся в памяти после того, как управление возвращаете в DOS, называются **резидентными.** Превратить программу в резидентную просто – достаточно вызвать специальную системную функцию DOS.

Функция DOS 31h – оставить программу резидентной

Вход: AH=31h; AL = код возврата; DX = размер резидентной части в 16-байтных параграфах.

Кроме того, существует и иногда используется предыдущая версия этой функции – прерывание 27h:

Прерывание INT 27h – оставить программу резидентной.

Вход: AH = 27h; DX = адрес последнего байта программы (считая от начала PSP) + 1

Эта функция не может оставлять резидентными программы размером больше 64 Кб, но многие программы, написанные на ассемблере, соответствуют этому условию. Так как резидентные программы уменьшают объем основной памяти, их всегда пишут на ассемблере и оптимизируют для достижения минимального размера. Никогда не известно, по каким адресам в памяти оказываются загруженные в разное время резидентные программы, поэтому единственным несложным способом получения управления является механизм программных и аппаратных пре рыканий. Резидентные программы принято разделять на активные и пассивные в зависимости от того, перехватывают ли они прерывания от внешних устройств или получают управление, только если программа специально вызовет команду INT с нужным номером прерывания и параметрами.

Лекция 14

Архитектура сопроцессора 8087

В архитектуре семейства микропроцессоров Intel 80x86 устройство для обработки чисел с плавающей точкой появилось в составе компьютера на базе микропроцессора *i*8086 и получило название *математический сопроцессор* (далее просто *сопроцессор).* Выбор такого названия был обусловлен тем, что это устройство было предназначено для расширения вычислительных возможностей основного процессора, кроме того оно было реализовано в виде отдельной микросхемы, т.е. его присутствие было необязательным. Микросхема сопроцессора для микропроцессора *i*8086 имела название *i*8087. Как отдельные устройства, сопроцессоры сохранялись вплоть до модели микропроцессора *i*386 и имели название *i*287 и *i*387 соответственно. Начиная с модели *i*486, сопроцессор исполняется в одном корпусе с основным микропроцессором.

Возможности сопроцессоров:

- Полная поддержка существующих стандартов на арифметику с плавающей точкой;
- Обработка целых десятичных чисел с точностью до 18 разрядов, что позволяет сопроцессору выполнять арифметические операции без округления над целыми десятичными числами со значениями до 10^{18};
- Обработка вещественных чисел из диапазона от 3.37×10^{-4932} до $1.18 \times 10^{+4932}$.
- Вычисление значений тригонометрических функций, логарифмов, различных математических функций.

С точки зрения программиста, сопроцессор представляет собой совокупность регистров, каждый из которых имеет свое функциональное назначение.

В программной модели сопроцессора можно выделить три группы регистров.

1. Восемь регистров данных **R0**, **R1**, ..., **R7**, организованных в виде стека и образующих основу сопроцессора – *стек сопроцессора*. Каждый из регистров имеет разрядность 80 битов.

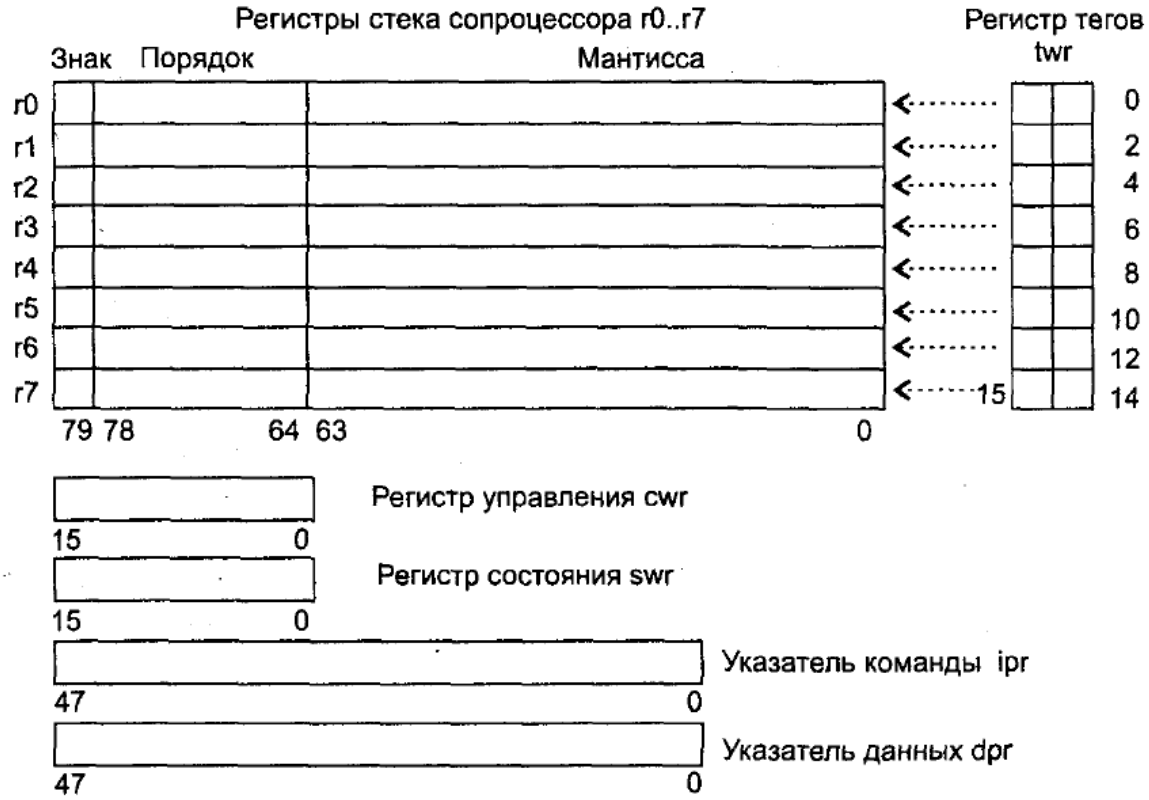
2. Три служебных регистра:

регистр состояния сопроцессора **SWR** (*Status Word Register* – регистр слова состояния, 16 разр.) – отражает информацию о текущем состоянии сопроцессора;

управляющий регистр сопроцессора **CWR** (*Control Word Register* – регистр слова управления, 16 разр.) – управляет режимами работы сопроцессора. С помощью полей в этом регистре можно регулировать точность выполнения численных вычислений, управлять округлением, разрешать или запрещать (маскировать) исключения;

регистр слова тегов **TWR** (*Tags Word Register* – слово тегов, 16 разр.) – используется для контроля за состоянием каждого из регистров R0, ..., R7. Содержит 8 пар битов, определяющих для каждого регистра 4 возможных состояния: регистр пуст, регистр содержит число, регистр содержит нулевое значение, регистр содержит значение, не являющееся числом (например, бесконечность).

3. Два регистра указателей – *данных* **DPR** (*Data Point Register*, 48 разр.) и *команд* **IPR** (*Instruction Point Register*, 48 разр.). Эти указатели используются при обработке исключительных ситуаций. Они предназначены для запоминания информации об *адресе команды*, вызвавшей исключительную ситуацию и *адресе ее операнда*.

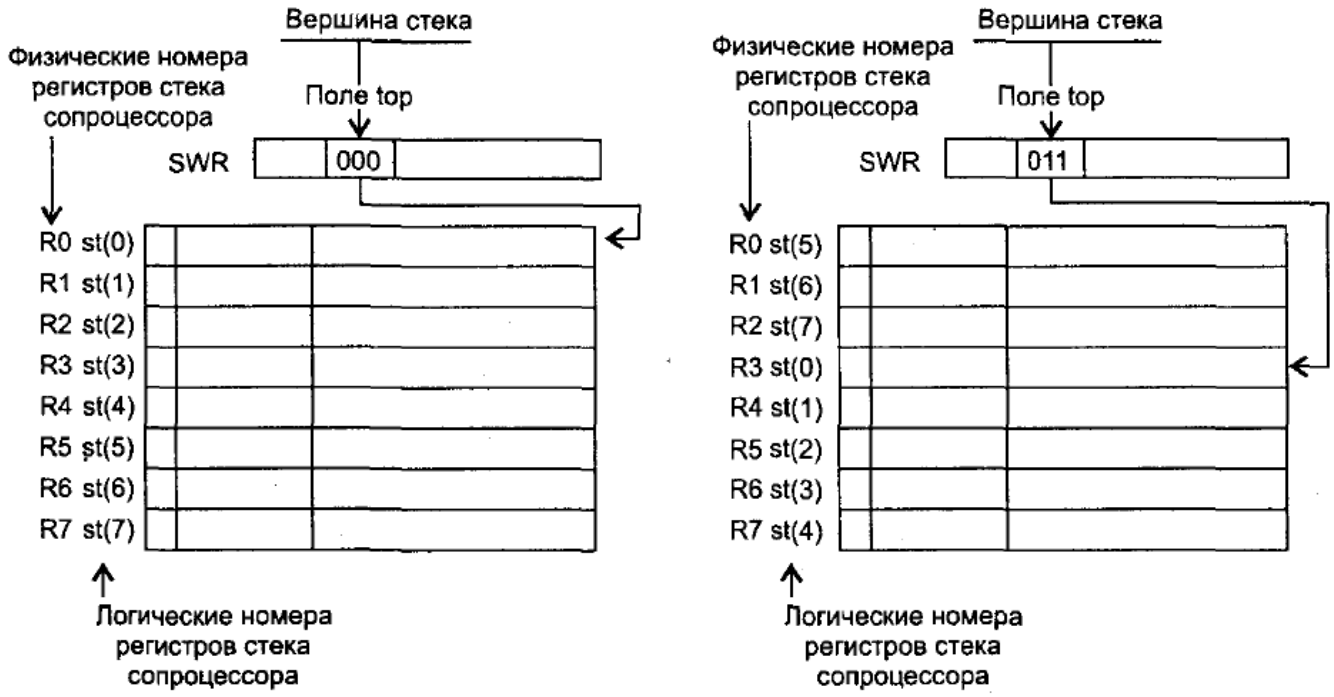


Программная модель сопроцессора

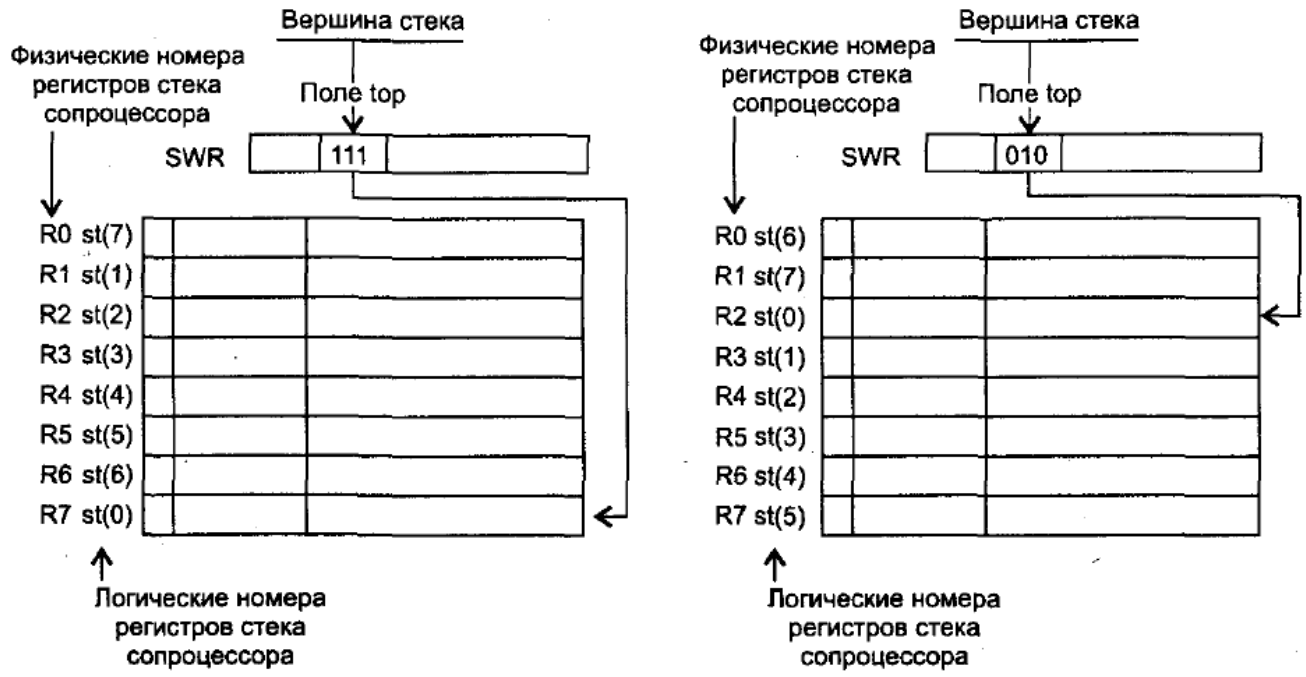
Все регистры являются программно доступными. Однако к одним из них доступ получить легко, так как для этого в системе команд сопроцессора существуют специальные команды. Для доступа к другим регистрам необходимо выполнить дополнительные действия, так как специальных команд для этого нет.

Регистровый стек сопроцессора организован по принципу кольца. Это означает, что среди всех регистров, составляющих стек, нет такого, который является вершиной стека. Все регистры стека с функциональной точки зрения абсолютно одинаковы и равноправны. Контроль текущей вершины осуществляется аппаратно с помощью трехбитового поля **TOP** (11, 12, 13 биты) регистра **SWR**. В поле TOP фиксируется номер регистра стека 0...7 (R0, ..., R7), являющегося в данный момент текущей вершиной стека.

Команды сопроцессора не оперируют физическими номерами регистров стека R0, ..., R7. Вместо этого они используют логические номера этих регистров ST(0), ST(1), …, ST(7). С помощью логических номеров реализуется относительная адресация регистров стека сопроцессора. Например, если текущей вершиной до записи в стек является физический регистр стека R3, то после записи в стек текущей вершиной становится физический регистр стека R2. То есть, по мере записи в стек, указатель его вершины движется по направлению к младшим номерам физических регистров (уменьшается на единицу). Если текущей вершиной является R0, то после записи очередного значения в стек сопроцессора его текущей вершиной станет физический регистр R7. Что касается логических номеров регистров стека ST(0), ..., ST(7), то они смещаются вместе с изменением текущей вершины стека. Таким образом, реализуется принцип кольца.



Состояние стека сопроцессора до выполнения записи данных



Состояние стека сопроцессора после выполнения записи данных

Как процессор, так и сопроцессор подключены к общей системной шине и имеют доступ к одинаковой информации. Инициирует процесс выборки всегда основной процессор. После выборки команда попадает одновременно в оба процессора и исполняется тем из них, для которого она предназначена (это зависит от типа

команды). В общем случае сопроцессор можно воспринимать как набор дополнительных регистров, для работы с которыми предназначены специальные команды.

Система команд сопроцессора

Система команд сопроцессора включает в себя около 80 машинных команд. Имеются следующие группы команд:

1. Команды передачи данных (между стеком сопроцессора и оперативной памятью компьютера)
   - операции с вещественными данными;
   - операции с целочисленными данными;
   - операции с десятичными данными;
   - загрузка констант;
   - обмен данными.
2. Команды сравнения данных
   - сравнение вещественных данных;
   - сравнение целочисленных данных;
   - анализ регистра ST(0);
   - сравнение с нулем;
   - команды условного сравнения.
3. Арифметические команды
   - арифметические операции (+, -, *, /) над вещественными данными;
   - арифметические операции над целочисленными данными;
   - вспомогательные арифметические команды (*fsqrt, fabs*).
4. Транцендентные команды
   - вычисление тригонометрических функций (*sin, cos, tg, arctg*);
   - вычисление логарифмов и степеней.
5. Команды управления
   - инициализация сопроцессора;
   - работа со средой;
   - работа со стеком;

- переключение режимов.

Все команды сопроцессора начинаются с символа ***f***. Здесь:

finit – инициализация сопроцессора (регистры сопроцессора инициализируются начальными значениями);

fld *источник* – загрузка вещественного числа из области памяти *источник* в вершину стека сопроцессора;

fst *приемник* – сохранение вещественного числа из вершины стека сопроцессора в память *приемник*;

fabs – вычисление модуля значения, находящегося в вершине стека сопроцессора ST(0). Результат вычисления помещается в регистр ST(0);

fsqrt – вычисление квадратного корня из значения, находящегося в вершине стека сопроцессора ST(0). Результат вычисления помещается в регистр ST(0);

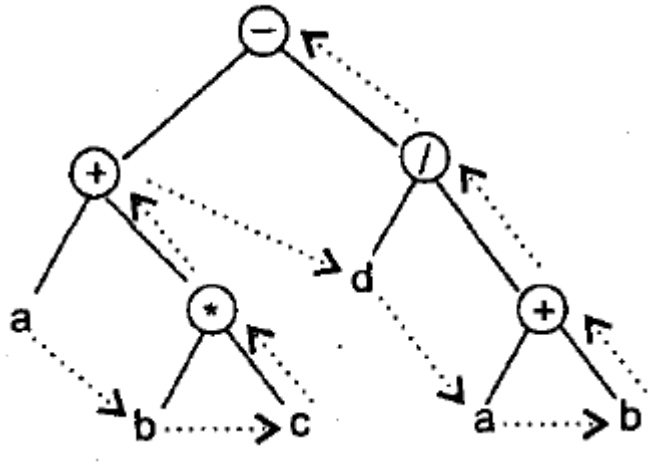
fsub *источник*– команда вычитает значение ячейки памяти с адресом *источник* из значения в ST(0). Результат вычитания запоминается в регистре ST(0);

fmul – умножает содержимое ST(0) на содержимое ST(l). Результат умножения запоминается в регистре ST(0).

Методика написания программ для сопроцессора имеет свои особенности. Главная причина – в стековой организации сопроцессора. Для того чтобы написать программу для вычисления некоторого выражения, его необходимо предварительно преобразовать в удобный для программирования сопроцессора вид. Процесс преобразования основан на использовании обратной польской записи (другое название – постфиксная запись, так как знак записывается после операндов, участвующих в операции).

Пример. Вычислить выражение **a+b*c–d/(a+b)**.

Графически это выражение представляется в виде дерева



Листья ветвей дерева соответствуют операндам, а узлы – операциям. Пусть началом обхода будет лист самой левой ветви дерева.

Для получения обратной польской записи выражения необходимо совершить обход по дереву слева направо, при этом узлы должны просматриваться только после обхода всех исходящих из них ветвей. В результате выражение будет выглядеть так:

a b c * + d a b + / –

Алгоритм вычисления выражений, представленных в обратной польской записи, имеет следующий вид:

1. Выбрать очередной символ выражения.

2. Если очередной выбранный символ – операнд, то поместить его в стек, после чего перейти к шагу 1.

3. Если очередной выбранный символ – знак операции, то выполнить ее над одним или двумя операндами в вершине стека. Результат операции необходимо поместить обратно в вершину стека.

4. Если в исходной записи выражения еще остались символы, то перейти к шагу 1, иначе в вершине стека находится результат вычисления выражения.

Примеры программ

Вычислить выражение **a+b*c–d/(a+b)**

```
.model small
.stack 100
.data
A dt 0.1
B dt 0.2
C dt 0.3
D dt 0.4
X dq ?
.code
start: mov ax, @Data
mov ds, ax
fld   A
fld   B
fld   C
fmul
fadd
fld   D
fld   A
fld   B
fadd
fdiv
fsub
fst   X
mov   ah, 4Ch
int   21h
end   start
```

Вычислить выражение **$z=(sqrt(|x|)-y)^2$**

```
model small
.stack 100
.data
```

```
X       dq      -29e-4
У       dq      4.6
Z       dq      0
.code
mov   AX, @Data
mov   DS, AX
finit                 ; приведение сопроцессора в начальное состояние
fld     X             ; ST(0)=x
fabs                  ; ST(0)=|x|
fsqrt
fsub    Y             ; ST(0)=sqrt|x|-y
fst     ST(1)         ; квадрат вычислим через умножение
fmul
fst     Z
mov   AH, 4Ch
int     21h
end
```

Другие технологии

MMX. В процессоры *Intel Pentium MMX* и *AMD Athlon K6* была включена новая технология **MMX** (*MultiMedia eXtension* – мультимединое расширение). Формально, в процессорах появилось 8 новых 64-битных регистров, названных MM0-MM7. На самом деле, это младшие части регистров FPU. Команды технологии MMX позволяют одновременно обрабатывать нескольких целочисленных значений.

SSE. Начиная с *Intel Pentium* 3, процессоры снабдили новой технологией **SSE** (*Stream SIMD Extension* – потоковое SIMD-расширение) для одновременной обработки чисел с плавающей точкой.

SSE – это 8 новых 128-битных регистров: *XMM0-XMM7*, позволяющие вместить в себя по четыре 32-битных числа или по два 64-битных числа с плавающей точкой для их одновременной обработки.

Затем в Pentium 4 появилась технология **SSE2**, содержащая новые инструкции. Далее последовали SSE3 и SSSE3 (SSE4).

3DNow. Фирма Advanced MicroDevices (AMD) не торопилась встраивать в свои процессоры SSE. Ещё на K6-2 она реализовала новую технологию 3DNow, улучшив её в Athlon (Enhanced 3DNow!). 3DNow использует те же регистры, что и MMX, но позволяет помещать туда два 32-битных числа с плавающей точкой. Причём процессор выполнял 2 инструкции 3DNow за такт, обрабатывая 4 числа с плавающей точкой одновременно.