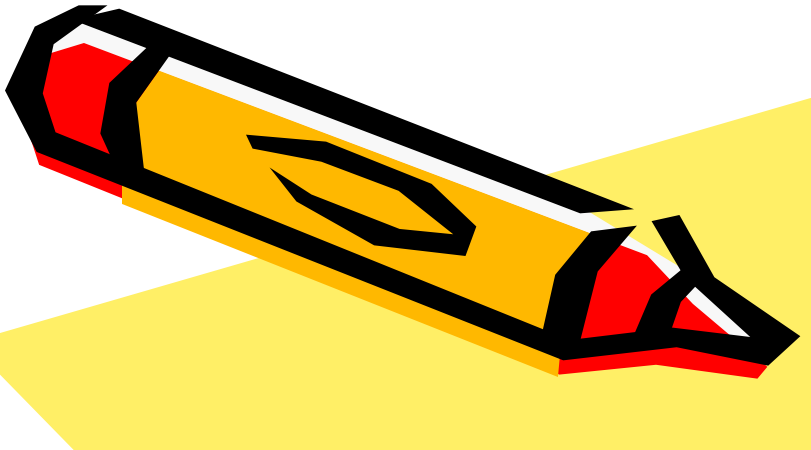
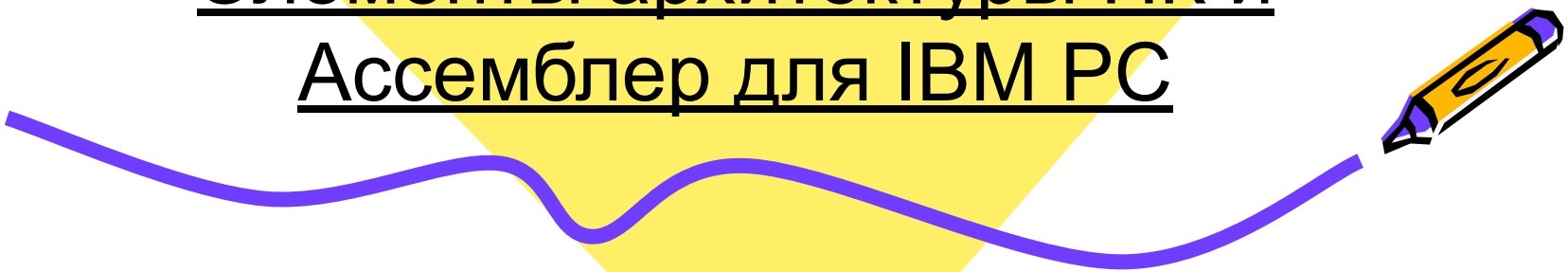


Системное программирование

Элементы архитектуры ПК и Ассемблер для IBM PC



- Основная литература:

- Бройдо В.Л., Ильина О.П. «Архитектура ЭВМ и систем», учеб. для вузов . -2-е изд., М.; СПб. [и др.]: Питер, 2009, 720 с.(52+3) экз

- Калашников О.А. «Ассемблер? Это просто! Учимся программировать»,: СПб.: БХВ-Петербург, 2007 365, +1 эл. опт. диск (CD-ROM) (в медиазале)+ (4+1)экз http://www.kalashnikoff.ru/Assembler

- Федорова А.Г. Электронный учебник «Основы программирования на языке Ассемблер для процессора INTEL» на сервере httpФедорова А.Г. Электронный учебник «Основы программирования на языке Ассемблер для процессора INTEL» на сервере http://Федорова А.Г. Электронный учебник «Основы программирования на языке Ассемблер для процессора INTEL» на сервере http://courseФедорова А.Г. Электронный учебник

Литература

1. В.Н. Пильщиков «программирование на языке Ассемблера»
2. В.И. Пустоваров «Язык Ассемблера в информационных и управляющих системах программирования»
3. О.В. Бурдаев, М.А. Иванов, И.И. Тетерин «Ассемблер в задачах защиты информации»
4. С.В. Зубков «Ассемблер – язык неограниченных возможностей. Программирование под DOS, Windows, Unix»
5. А. Жуков, А. Авдюхин «Ассемблер. Самоучитель»
6. С.К. Фельдман «Системное программирование»

История развития ПК

1948 год – создание транзистора….

1958 год – 1-я микросхема….

1971 год – 1-ый МП, МП, реализованный в виде 1 интегральной микросхемы. Intel 4004 - ….

1974 г. – 8-разрядный МП Intel 8080…

1975-1976 – 1-я ПЭВМ, созданная фирмой APPLE…

1978 г. – 16-й МП 8088…

1979 г. – 16-й МП 8086…29000 транзисторов, 3Мкр технология, МП – 33мм^2 площадь кристалла

1981 г. – IBM PC

1983 г. – IBM PC XT (Extended Technology)

1984 г. – IBM PC AT (Advenced Technology) – 2-е поколение

1987 г. – 32-й i386

1990 г. – i486…1,5 млн транзисторов, 1Мкр технология, 5-ти стадийный конвейер для выполнения команд кэш-память на кристалле процессора 8Кбайт

История развития ПК

1993 г. – 64-й МП «Pentium» - 5-е поколение: 3,1 млн. транзисторов, 0,8 Мкм технология

6-е поколение – Pentium Pro, Pentium 2, Pentium 3 с такт частотой 300 – 600 МГц

7-е поколение – «Willamate» 800 – 1200 МГц, кэш до 1 Мбайт 2000 г.

С 2002 г. – P4: 0,13 Мкм, 146мм^2, 55 млн транзисторов

В 2002 году обещали к 2005-у МП: 0,03 Мкм, на 1 см 12 млн транзисторов, размер транзистора в 100000 раз меньше толщины листа папиросной бумаги, такт частота – 10ГГц, более 400 Млн транзисторов и напряжение питания меньше 1в., может питаться от батарейки. Но….

История развития ПК

Еще недавно производительность процессора определяли его тактовой частотой, измеряемой в мегагерцах или гигагерцах. Конечно, тактовая частота процессора является одной из основных характеристик, но далеко не единственной. Процессоры могут отличаться друг от друга такими параметрами, как:

- микроархитектура ядра процессора,
- размер кэша,
- технологический процесс производства,
- поддерживаемая частота системной шины (FSB),
- напряжение питания,
- тепловыделение.

От них во многом зависит производительность процессора и его цена.

История развития ПК

Технологический процесс производства, определяет в первую очередь структурный размер тех элементов, из которых состоит процессор. От технологического процесса производства напрямую зависят размеры транзисторов и их характеристики.

Технологическим процессом производства определяется общее количество транзисторов в процессоре, разгонные возможности, максимальная тактовая частота, энергопотребление и тепловыделение процессора.

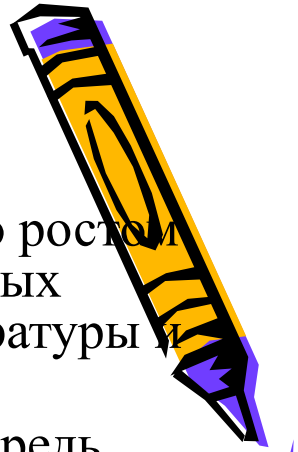
Не так давно процессоры производились по 0,18-микронному технологическому процессу, затем по 0,13-микронному, и 90-нанометровой технологии.

В апреле 2010 года был представлен современный игровой процессор

Intel Core i7-980X Extreme Edition

– шестиядерный, способный обрабатывать 12 потоков команд одновременно, изготовленный на базе 32-нм технологии и предназначенный для топовых систем. Он предлагает высокий уровень производительности для создания цифрового контента, 3D-рендеринга, одновременного запуска большого числа приложений и требовательных к ресурсам видеоигр. Чип обладает 12 МБ кэш-памяти Intel® Smart Cache – на 50% больше в сравнении с существующим флагманским процессором для настольных систем. Сочетание процессора Intel Core i7-980X Extreme Edition, видеокарты ATI 5770 с 1 Gb памяти и 6 Gb оперативной памяти обеспечивает максимальную производительность для самых современных 3D игр, многозадачных приложений, кодирования видео, рендеринга, обработки графики и других ресурсоемких задач»

Введение



Расширение сфер применения компьютерной техники обусловлено ростом производительности и информационной емкости вычислительных систем, что в свою очередь зависит от успехов в развитии аппаратуры и программного обеспечения

Успехи в развитии аппаратуры определяются сегодня в первую очередь степенью интеграции элементной базы, развитием технологий параллельной обработки информации, развитием коллективного использования сетевых распределенных ресурсов.

Успехи в развитии ПО требуют использования всех средств автоматизации программирования для получения максимальной эффективности, скорости выполнения критических участков программ. Для решения этой задачи большую роль играет использование машинно-ориентированных языков. Выделим две сферы их применения:

1) разработка системных программ, включаемых в состав операционных систем (ОС), например, драйверы устройств;

2) решение специализированных задач информационных и управляющих систем, к которым относят программы управления базами данных и языком интерфейса, программы сбора и обработки информации в информационно-измерительных системах и комплексах, в том числе и сетевых,…

При классификации программных средств традиционно их деление на прикладные, или проблемные - программы пользователей и системные программы, поддерживающие работу вычислительных систем, комплексов и сетей в автоматическом режиме.

Программные средства пользователей включают в себя комплексы долговременно сохраняемых программ для решения задач из узкой предметной области пользователя.

К классу системных программ относят специальные программы, обеспечивающие автоматизированную разработку программ и выполнение любых программ.

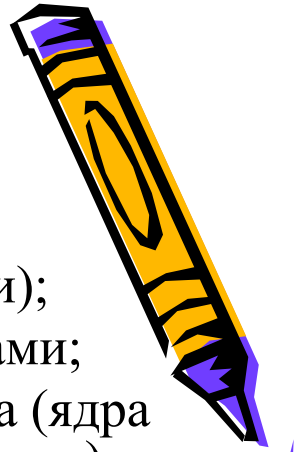
При развитии ВС часто употребляемые функции типовых проблемных программ поднимают на уровень системных программ для использования их в различных приложениях, а в дальнейшем наиболее распространенные и критичные по временным затратам на уровень частичной или полной аппаратной реализации. Такой путь прошли в последние десятилетия средства управления многопрограммным защищенным режимом в процессорах фирмы Intel – от программной до частично аппаратной. А путь от прикладных до системных управляющих прошли, например, средства управления диалоговым взаимодействием с пользователем, реализованных в объектно-ориентированных [illegible] программных оболочках (Windows, например).

Управляющие системные программы, обеспечивающие корректное выполнение всех процессов при решении задач на компьютере и функционирование всех устройств ВС, постоянно находятся в оперативной памяти (ОП) составляют ядро ОС и называются **резидентными** программами. Управляющие программы, которые загружаются в ОП непосредственно перед выполнением, называют **транзитными.**

Обрабатывающие системные программы выполняются как специальные прикладные или приложения ОС, используемые пользователем при создании новых или модификации ранее созданных системных программ. При создании таких программ используются машинно-ориентированные языки и языки высокого уровня. Однако, эффективность программ, созданных на языках высокого уровня в любом случае будет ниже, чем на языках машинно-ориентированных, написанных высоко квалифицированным программистом.

Язык Ассемблер используется везде, где необходима максимальная производительность и эффективность, и будет использоваться до тех пор, пока проводятся исследовательские работы в области развития и создания новых архитектур ЭВМ.

На Ассемблере пишут:



то, что требует максимальной скорости выполнения (основные компоненты компьютерных игр, ядра ОС реального времени);

то, что непосредственно взаимодействует с внешними устройствами;

то, что должно полностью использовать возможности процессора (ядра многозадачных ОС, программы перевода в защищенный режим);

все, что полностью использует возможности ОС (вирусы, антивирусы, программы защиты и взлома защит);

программы, предназначенные для обработки больших объемов информации.

К недостаткам относят:

- трудно выучить…
- трудночитаемы…
- не переносятся на другие процессоры (благодаря этому максимальная эффективность)…
- трудно писать (нет стандартных модулей)…
- зачем использовать, если такие мощные компьютеры….

Архитектура ПК

Понятие «архитектура ЭВМ» включает в себя структурную организацию аппаратных средств (набор блоков, устройств, объединенных в единую вычислительную систему) и функциональную организацию, позволяющую реализовать программное управление этой системой. С точки зрения программиста архитектура ЭВМ - это набор программно-доступных средств.

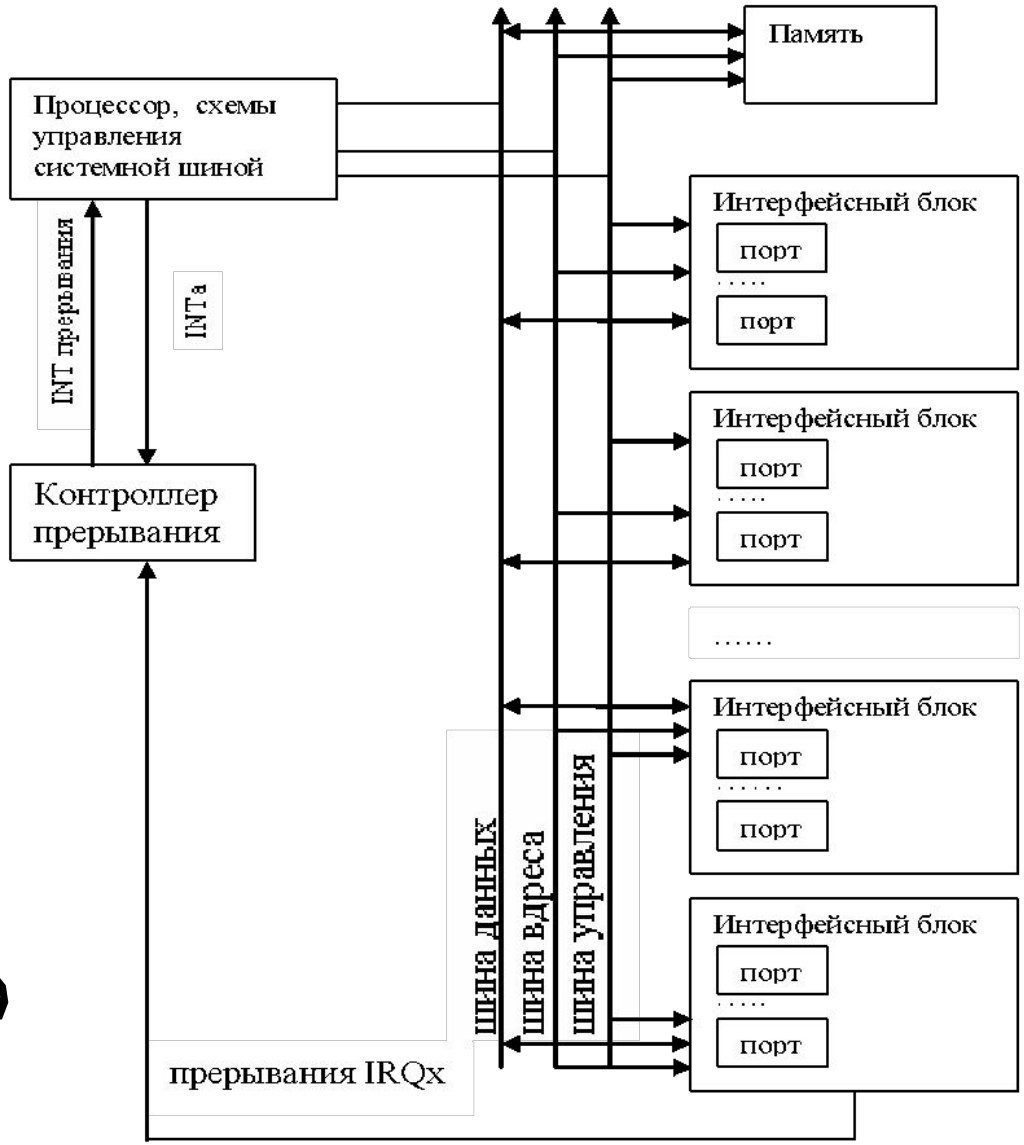
В современных ПК реализован магистрально-модульный принцип построения. Все устройства (модули) подключены к центральной магистрали, системной шине, которая включает в себя адресную шину, шину данных и шину управления.

Шина – это набор линий связи, по которым передается информация от одного из источников к одному или нескольким приемникам. Адресная шина однонаправленная, адреса передаются от процессора. Шина данных двунаправленная, данные передаются как от процессора, так и к процессору. В шину управления входят линии связи и однонаправленные и двунаправленные.

Внешние устройства работают значительно медленнее процессора, поэтому для организации параллельной работы процессора и внешних устройств в архитектуру компьютера входит система прямого доступа к памяти (ПДМА) и интерфейсные блоки, включающие в себя устройства управления внешними устройствами (контроллеры, адаптеры)…

Архитектура ПК

ПК



послед. порт

парал. порт

Для устройств ввода

Для устройств вывода

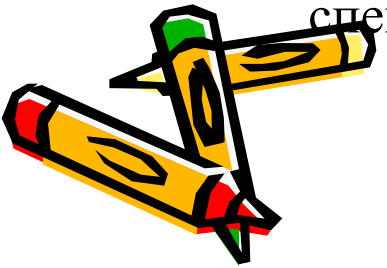
Архитектура микропроцессора ix86.

Процессор ix86 после включения питания устанавливается в реальный режим адресации памяти и работы процессора.

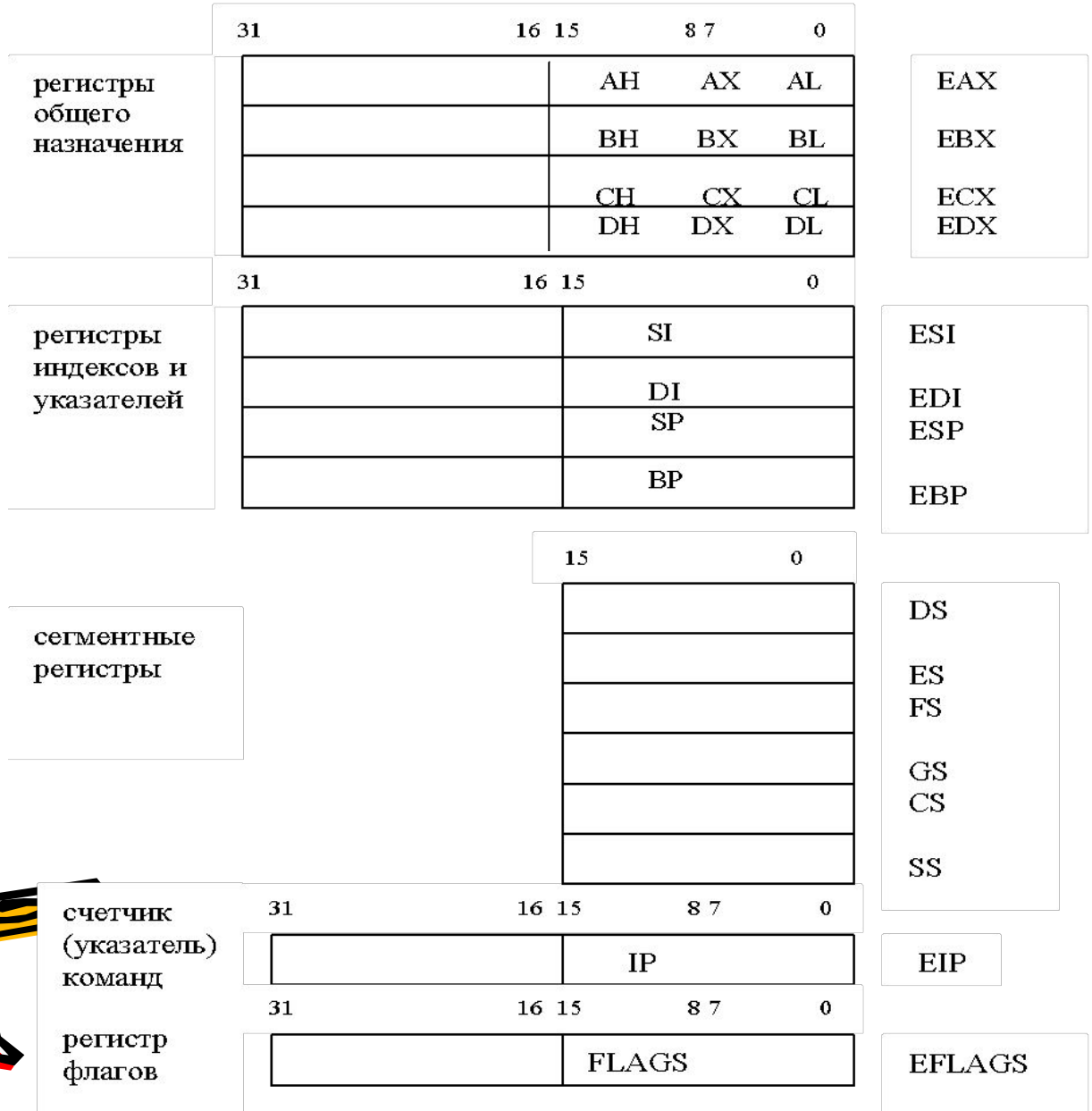
Большинство ОС сразу переводит его в защищенный режим, обеспечивает многозадачность, распределение памяти, ресурсов и других дополнительных возможностей. Программы пользователей в таких ОС могут работать в еще одном режиме, режиме виртуальных машин...

Совокупность программно-доступных средств процессора называется архитектурой процессора, с точки зрения программиста.

Начиная с 386 процессора программисту доступны 16 основных регистров, 11 регистров для работы с сопроцессором и мультимедийными приложениями, и в реальном режиме доступны некоторые регистры управления и некоторые специальные регистры.



Регистр – это набор из n устройств, способных хранить n-разрядное двоичное число.



Регистры общего назначения

32-х разрядные регистры общего назначения без ограничения могут использоваться для временного хранения команд, адресов и данных. Обращение к ним осуществляется по именам **EAX, EBX, ECX, EDX** при работе с 32-х разрядными данными, по именам **AX, BX, CX, DX**, при работе со словами - 16-ти разрядными данными, и при работе с байтами могут использоваться восемь 8-разрядных регистров: **AL, AH, BL, BH, CL, CH, DL, DH**.

Эти регистры имеют собственные имена, которые говорят о том, как они обычно используются. AX - аккумулятор…, DX – регистр данных. BX – регистр базы используется для организации специальной адресации операндов по базе.

CX - счетчик используется автоматически для организации циклов и при работе со строками.

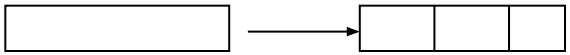
Регистры указателей и индексов имеют специальные назначения. Регистры индексов используются для организации сложных способов адресации операндов, а регистры указателей - для организации работы с сегментом стека.

Рассматриваемый процессор может работать с оперативной памятью как с непрерывным массивом байтов (модель памяти flat), так и с разделенной на много массивов - сегментов.

Во втором случае физический адрес байта состоит из 2-х частей: адрес начала сегмента и смещение внутри сегмента.

Для получения адреса начала сегмента используются сегментные регистры **DS,ES, FS, GS, CS и SS**, называемые селекторами. Операционные системы могут размещать сегменты в различных областях оперативной памяти и даже временно записывать на винчестер, если ОП не хватает. С каждым селектором связан программно-недоступный дескриптор, в котором содержится адрес сегмента, размер сегмента и некоторые его атрибуты. Это для защищенного режима работы. В реальном режиме размер сегмента фиксирован и составляет 64 Кбайта. Адрес сегмента кратен 16 и в 16-ой системе счисления может быть записан в виде $XXXX0_{16}$ и четыре старшие цифры адреса сегмента содержатся в сегментном регистре. В защитном режиме размер сегмента может изменяться до 4Гбайт.

селектор



дескриптор

DS, ES, FS, GS - 16-ти разрядные сегментные регистры, используемые для определения начала сегментов данных. CS - сегментный регистр кодового сегмента. SS - сегментный регистр для определения сегмента стека.

Сегментных регистров всего 6, но в любой момент пользователь может изменить содержимое этих регистров. Например,…..

Специальным образом реализуется и используется сегмент стека….

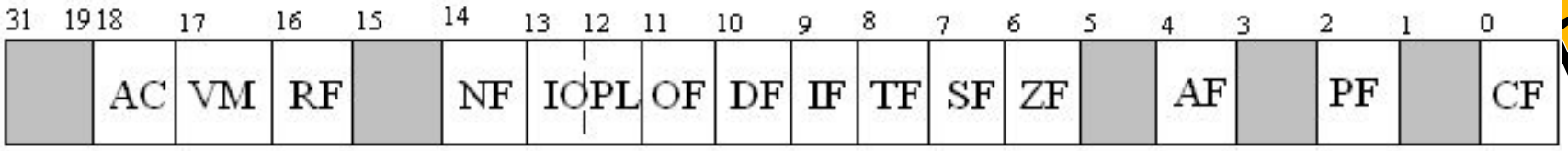
Адрес начала сегмента стека определяется автоматически ОС с помощью регистра SS, а указатель на вершину стека – это регистр указателей SP (ESP). Стек организован таким образом, что при добавлении элементов в стек, содержимое указателя стека уменьшается. Стек растет вниз от максимального значения, хранящегося в SS (растет вниз головой). При добавлении в стек адреса уменьшаются. Такая организация необходима при использовании модели памяти flat. В этом случае программа размещается, начиная с младших адресов, а стек размещается в старших

Стек используется для временного хранения данных, для организации работы с подпрограммами, в том числе и рекурсивными, для передачи параметров подпрограммам, размещения локальных параметров и т.д.

Для того, чтобы стек можно было использовать для хранения и фактических и локальных параметров, после передачи фактических параметров значение указателя на вершину стека можно сохранить в регистре BP и тогда к глобальным параметрам можно обращаться, используя конструкцию

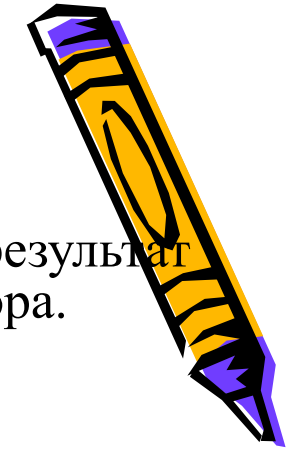
BP - k, а к локальным - BP + n, где k, и n - определяются количеством параметров и их размером.

Регистр флагов. Регистр FLAGS или EFLAGS определяет состояние процессора и программы в каждый текущий момент времени.



- CF - перенос
- PF - четность
- AF - полуперенос
- ZF - флаг нуля
- SF - флаг знака
- TF - флаг трассировки
- IF - флаг прерывания
- DF - флаг направления
- OF - флаг переполнения
  - AC - флаг выравнивания операндов
  - VM - флаг виртуальных машин
  - RF - флаг маскирования прерывания
  - NT - флаг вложенной задачи
  - IOPL - уровень привилегий ввода/вывода.

Регистр флагов



Биты 1, 3, 5, 15, 19 - 31 - не используются, зарезервированы.

В реальном режиме используют 9 флагов, из них 6 реагируют на результат выполнения команды, 3 определяют режим работы процессора.

В защищенном режиме используются 5 дополнительных флагов, определяющих режим работы процессора.

CF устанавливается в 1, если при выполнении команды сложения осуществляется перенос за разрядную сетку, а при вычитании требуется заем. 0FFFFh + 1 = 0000h и CF = 1 при работе со словами

PF = 1, если в младшем байте результата содержится четное количество единиц.

AF = 1, если в результате выполнения команды сложения (вычитания) осуществлялся перенос (заем) из 3-го разряда байта в 4-й (из 4-го в 3-й).

ZF = 1, если результатом выполнения операции является 0 во всех разрядах результата.

SF всегда равен знаковому разряду результата.

TF = 1 прерывает работу процессора после каждой выполненной команды.

Регистр флагов

DF определяет направление обработки строк данных, если DF= 0 – обработка строк идет в сторону увеличения адресов, 1 - в сторону уменьшения, (автоматическое увеличение или уменьшение содержимого регистров индексов SI и DI).

OF = 1, если результат команды превышает максимально допустимый для данной разрядной сетки.

IOPL = 1, если уровень привилегии текущей программы меньше значения этого флажка, то выполнение команды ввод/вывод для этой программы запрещен.

NT - определяет режим работы вложенных задач.

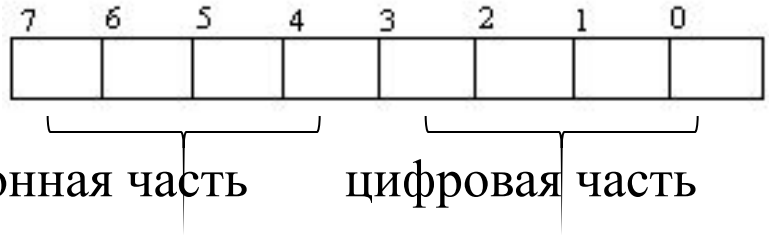
RF позволяет маскировать некоторые прерывания процессора.

VM - позволяет перейти из защищенного режима в режим виртуальных машин.

AC =1 приведет к сообщению об ошибке, если адреса операндов длиной в слово или двойное слово не будут кратны двум и четырем соответственно.

Оперативная память

Оперативная память состоит из байтов, каждый байт состоит из 8 информационных битов.



32-х разрядный процессор может работать с ОП до 4Гбайт и, следовательно, адреса байтов изменяются от 0 до 2^{32}-1

(00000000_{16} – $FFFFFFFF_{16}$).

Байты памяти могут объединяться в поля фиксированной и переменной длины.

Фиксированная длина – слово (2 байта), двойное слово (4 байта). Поля переменной длины могут содержать произвольное количество байтов.

Адресом поля является адрес младшего входящего в поле байта. Адрес поля может быть любым.

ОП может использоваться как непрерывная последовательность байтов, так и сегментированная.

Оперативная память

Физический адрес (ФА) байта записывается как:

<сегмент> : <смещение>, т.е.

он может быть получен по формуле ФА = АС + ИА,

где АС – адрес сегмента, ИА – исполняемый адрес,

т.е. ИА - <смещение> формируется в команде различными способами в зависимости от способа адресации операндов.

В защищенном режиме программа может определить до 16383 сегментов размером до 4 Гбайт, и таким образом может работать с 64 Тбайтами виртуальной памяти.

Для реального режима АС определяется сегментным регистром и для получения двадцатиразрядного двоичного адреса байта необходимо к содержимому сегментного регистра, смещенного на 4 разряда влево, прибавить шестнадцатиразрядное смещение - ИА.

Например, адрес следующей исполняемой команды:

$$ФА = (CS) + (IP)$$

$$(CS) = 7A15_{16} = 0111101000010101000_{2},$$

$$(IP) = C7D9_{16} = 1100011111011001_{2}.$$

$$ФА = 86929_{16} = 10000110100100101001_{2}$$

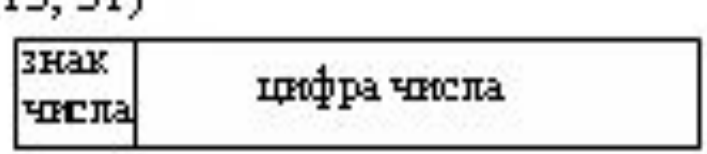
Форматы данных

Процессор ix86 вместе с сопроцессором могут обрабатывать большой набор различных типов данных: целые числа без знака, целые числа со знаком, действительные числа с плавающей точкой, двоично-десятичные числа, символы, строки, указатели.

Целые числа без знака могут занимать байт, слово, двойное слово и принимать значения из диапазонов:

0 - 255, 0 - 65535, 0 – 4294967295 соответственно.

Целые числа со знаком могут занимать также байт, слово, двойное слово. Они хранятся в дополнительном коде и имеют следующий вид.



Форматы данных

Дополнительный код положительного числа равен самому числу.

Дополнительный код отрицательного числа в любой системе счисления может быть получен по формуле :

$X = 10^n - |X|$, где n – разрядность числа.

Например, представим в слове отрицательное 16-ричное число $-AC7_{16}$

$10^4 - AC7 = F539$.

Дополнительный код двоичного числа может быть получен инверсией разрядов и прибавлением 1 к младшему разряду.

Например, - 12 в байте:

1) $12 = 00001100_2$,
2) инверсия – 11110011_2,
3) дополнительный код – 11110100_2.

Рассмотрим выполнение операции вычитания в машине: дополнительный код вычитаемого прибавляется к уменьшаемому.

Например: $65 – 42 = 23$.

1) $65 = 01000001_2$,
2) $42 = 00101010_2$,
3) $-42 = 11010110_2$,
4) $65 - 42 = 00010111_2 = 2^0 + 2^1 + 2^2 + 2^4 = 1+2+4+16=23$.

Форматы данных

Числа с плавающей точкой могут занимать 32 бита или 64 бита или 80 бит и называются короткое вещественное, длинное вещественное, рабочее вещественное. Формат числа с плавающей точкой состоит из трех полей: <знак числа>, <машинной порядок>, <мантисса>.

короткое вещественное 1+ 8 + 23 $- 10^{\pm 32}$ $- + 10^{\pm 32}$

длинное вещественное 1+ 11 + 52 $- 10^{\pm 308}$ $- + 10^{\pm 308}$

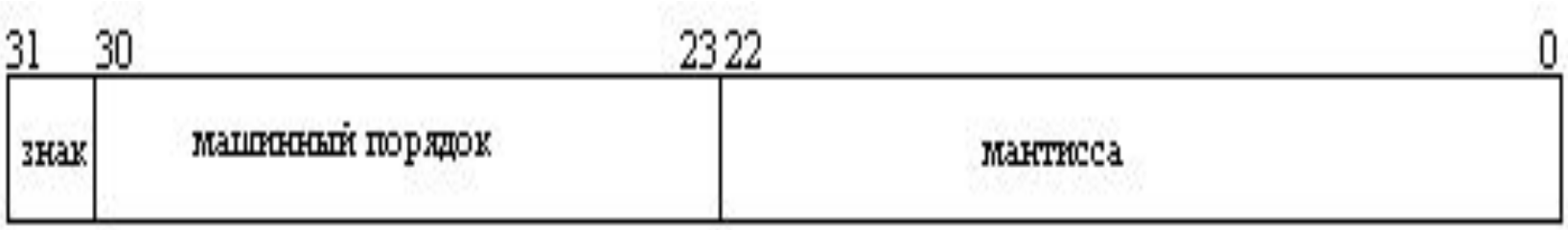
рабочее вещественное 1+ 15 + 64 $- 10^{\pm 4932}$ $- + 10^{\pm 4932}$.

Машинный порядок (Пм) включает в себя неявным образом знак порядка и связан с истинным порядком (Пи) формулой:

$$Пм = Пи + 127_{10} \ (1023_{10}, 16383_{10}).$$

Предполагается, что мантисса нормализована и старший единичный разряд мантиссы не помещается в разрядную сетку.

Например, для короткого вещественного:



Форматы данных

Пример, 3060_{10} представить в виде числа с плавающей точкой, занимающего 4 байта.

1) $3060_{10} = BF4_{16}$

$$\frac{1}{\text{основание сист. счисления}}$$

2) нормализуем число $0.\ BF4*10^{3}_{16}$

3) получим машинный порядок $Пм = 3_{16} + 7F_{16} = 82_{16}$

4) запишем в разрядную сетку в 2-ичной системе счисления:

$0\ 1000\ 0010\ 011\ 1111\ 0100\ 0000\ 0000\ 0000_2$

Или в 16-ричном виде: $413F4000_{16}$.

$0100\ 0001\ 0\ 011\ 1111\ 0100\ 0000\ 0000\ 0000_2$

Двоично-десятичные данные - процессором могут обрабатываться 8-ми разрядные в упакованном и неупакованном формате, и сопроцессором могут обрабатываться 80-ти разрядные данные в упакованном формате. Упакованный формат предполагает хранение двух цифр в байте, а неупакованный – хранит одну цифру в цифровой части байта.

Форматы данных

Символьные данные - символы в коде ASCII. Для любого символа отводится один байт.

Строковые данные – это последовательности бит, байт, слов или двойных слов.

Указатели - существуют два типа указателей:

длинный указатель, занимающий 48 бит - селектор(16) + смещение(32)

и короткий указатель, занимающий 32 бита - только смещение.

Форматы команд

Команда – это цифровой двоичный код, состоящий из двух подпоследовательностей двоичных цифр, одна из которых определяет код операции (сложить, умножить, переслать), вторая – определяет операнды, участвующие в операции и место хранения результата.

Рассматриваемый процессор может работать с безадресными командами, одно-, двух- и трехадресными командами. Команда в памяти может занимать от 1 до 15 байт и длина команды зависит от кода операции, количества и места расположения операндов. Одноадресные команды могут работать с операндами, расположенными в памяти и регистрах, для двухадресных команд существует много форматов, такие, как:

R-R M-M R-M M-R R-D M-D,

где R – регистр, M – память, D – данные.

Форматы команд

Операнды могут находиться в регистрах, памяти и непосредственно в команде и размер операндов может быть - байт, слово или двойное слово.

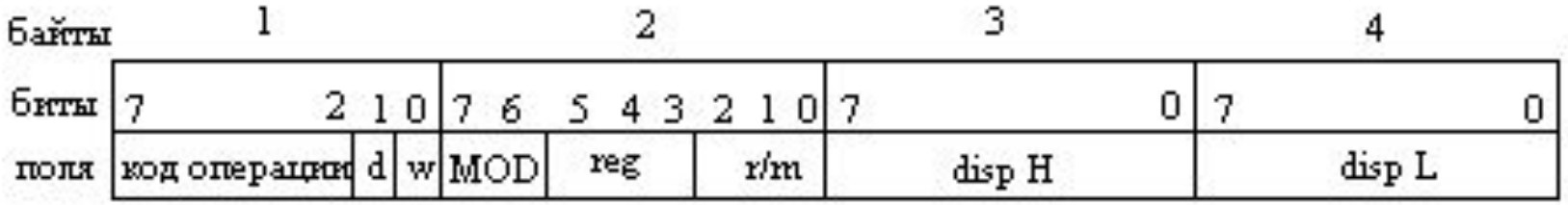
Исполняемый адрес операнда в общем случае может состоять из трех частей: <база> <индекс> <смещение>, например, [BX] [SI] M.

Существуют различные способы адресации операндов, такие как:

1. регистровая
2. непосредственная
3. прямая
4. косвенно-регистровая
5. по базе со смещением
6. прямая с индексированием
7. по базе с индексированием
8. косвенная адресация с масштабированием
9. базово-индексная с масштабированием
10. базово-индексная с масштабированием и смещением.

Адресации с 8 по 10 используются только в защищенном режиме.

Машинный формат двухадресной команды, для которой один операнд находится всегда в регистре, а второй – в регистре или памяти можно представить следующим образом:



“disp H/disp L” – “старшая / младшая часть смещения.

Поля “код операции” и иногда “reg” определяют выполняемую операцию.

Поле “d” определяет место хранения первого операнда.

Поле “w” определяет с какими данными работают: с байтами, или словами. Если w = 0, команда работает с байтами, w = 1 - со словами.

reg” - определяет один операнд, хранимый в регистре.

Поля “mod”, “disp H” и “disp L” определяют второй операнд, который может храниться в регистре или в памяти.

Если mod = 11, то второй операнд находится в регистре, он определяется полем “r/m”, а “disp H/disp L” – отсутствует, команда будет занимать 2 байта в памяти, если mod < > 11, то второй операнд находится в памяти.

Машинный формат двухадресной команды

Значение поля “mod” определяет как используется смещение:

mod { 00, disp – отсутствует
01, disp = disp L – с распространением знака до 16
10, смещение состоит из disp H и disp L.

Поля “reg” и “r/m” определяют регистры:

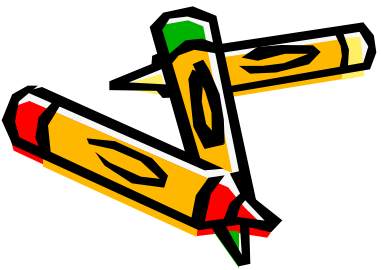
reg / r/m	000	001	010	011	100	101	110	111
w = 0	AL	CL	DL	BL	AH	CH	DH	BH
w = 1	AX	CX	DX	BX	SP	BP	SI	DI

Физический адрес определяется так:

r/m	ИА	ФА
000	(BX) + (SI) + disp	+ (DS)
001	(BX) + (DI) + disp	+ (DS)
010	(BP) + (SI) + disp	+ (SS)

Машинный формат двухадресной команды.

r/m	ИА	ФА
011	(BP) + (DI) + disp	+ (SS)
100	(SI) + disp	+ (DS)
101	(DI) + disp	+ (DS)
110	(BP) + disp	+ (SS)
111	(BX)+ disp	+ (DS)



Примеры команд с различной адресацией операндов.

В командах на Ассемблере результат всегда пересылается по адресу первого операнда.

1) Регистровая

MOV AX, BX ; (BX) → AX

Машинный формат: 1001 0011 1100 0011

“код операции” 100100

“d” = 1

“w” = 1

“mod” = 11

“reg” = 000

“r/m” = 011

2) Непосредственная

MOV AX, 25 ; 25 → AX

CONST EQU 34h ; именованная константа **CONST**

MOV AX, CONST ; 34h → AX

Примеры команд с различной адресацией операндов

3) Прямая

Если известен адрес памяти, начиная с которого размещается операнд, то в команде можно непосредственно указать этот адрес.

MOV AX, ES : 0001 ;

ES – регистр сегмента данных, 0001 – смещение внутри сегмента.

Содержимое двух байтов, начиная с адреса (ES) + 0001 пересылаются в AX - ((ES) + 0001) →AX.

Прямая адресация может быть записана с помощью символического имени, которое предварительно поставлено в соответствие некоторому адресу памяти, с помощью специальной директивы определения памяти, например: DB – байт,

DW – слово,

DD – двойное слово.

Если в сегменте ES содержится директива Var_p DW, тогда по команде

MOV AX, ES : Var_p ; ((ES) + Var_p) →AX.

Например, если команда имеет вид:

MOV AX, Var_p; ((DS) + Var_p) →AX.

Примеры команд с различной адресацией операндов

4) Косвенно-регистровая

Данный вид адресации отличается от регистровой адресации тем, что в регистре содержится не сам операнд, а адрес области памяти, в которой операнд содержится.

MOV AX, [SI] ;

Могут использоваться регистры:

SI, DI, BX, BP, EAX. EBX, ECX, EDX, EBP, ESI, EDI.

Не могут использоваться: AX, CX, DX, SP, ESP.

5) По базе со смещением

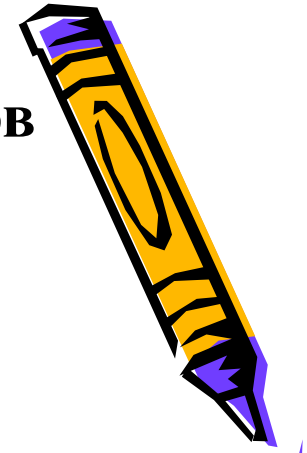
MOV AX, [BX]+2 ; ((DS) + (BX) + 2) →AX.

≡ MOV AX, [BX + 2] ;

≡ MOV AX, 2[BX] ;

MOV AX, [BP + 4] ; ((SS) + (BP) + 4) →AX.

Примеры команд с различной адресацией операндов



6) Прямая с индексированием

MOV AX, MAS[SI] ; ((DS) + (SI) + MAS) →AX

MAS – адрес в области памяти.

С помощью этой адресации можно работать с одномерными массивами. Символическое имя определяет начало массива, а переход от одного элемента к другому осуществляется с помощью содержимого индексного регистра.

7) По базе с индексированием

MOV AX, Arr[BX][DI] ; ((DS) + (BX) + (DI) + Arr) →AX.

Эта адресация используется для работы с двумерными массивами. Символическое имя определяет начало массива, с помощью базового регистра осуществляется переход от одной строки матрицы к другой, а с помощью индексного регистра - переход от одного элемента к другому внутри строки.

Особенности использования команд пересылки

1. Нельзя пересылать информацию из одной области памяти в другую;
2. Нельзя пересылать информацию из одного сегментного регистра в другой;
3. Нельзя пересылать непосредственный операнд в сегментный регистр, но если такая необходимость возникает, то нужно использовать в качестве промежуточного один из регистров общего назначения.

MOV DX, 100h

MOV DS, DX

4. Нельзя изменять командой MOV содержимое регистра CS.
5. Данные в памяти хранятся в «перевернутом» виде, а в регистрах в «нормальном» виде, и команда пересылки учитывает это, например, **R DW 1234h**

В байте с адресом R будет 34h, в байте с адресом R+1 будет 12h.

MOV AX, R; 12h → AH, 34h → AL.

Особенности использования команд пересылки

6. Размер передаваемых данных определяется типом операндов в команде.

X DB ? ; X - адрес одного байта в памяти.
Y DW ? ; Y определяет поле в 2 байта в памяти.
MOV X, 0 ; очищение одного байта в памяти.
MOV Y, 0 ; очищение двух байтов в памяти.
MOV AX, **0** ; очищение двух байтов регистра
MOV [SI], 0 ; сообщение об ошибке.

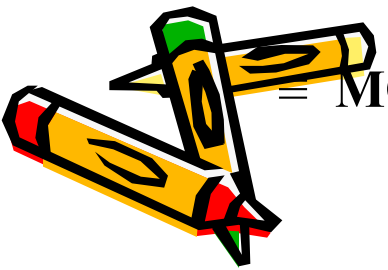
В последнем случае необходимо использовать специальный оператор PTR.

<тип> PTR <выражение>

Выражение может быть константным или адресным, а тип это:

BYTE, WORD, DWORD и т.д.

byte PTR 0 **;** 0 воспринимается как байт
word PTR 0 **;** 0 воспринимается как слово
byte PTR op1 **;** один байт в памяти начиная с этого адреса
MOV byte PTR [SI], 0;
≡ **MOV [SI], byte PTR 0**;
≡ **MOV [SI], word PTR 0** ; **0** → ((DS) +(SI))



Особенности использования команд пересылки

7. Если тип обоих операндов в команде определяется, то эти типы должны соответствовать друг другу.

MOV AH, 500 ; сообщение об ошибке.
MOV AX, X ; ошибка, X – 1байт, AX – 2 байта.
MOV AL, R ; ошибка
MOV AL, byte PTR R ; (AL) = 34h
MOV AL, byte PTR R+1 ; (AL) = 12h

К командам пересылки относят команду обмена значений операндов.

XCHG OP1, OP2 ; r↔r ∨ r↔m
MOV AX, 10h ;
MOV BX, 20h ;
XCHG AX, BX ; (AX) = 20h, (BX) = 10h

Для перестановки значений байтов внутри регистра используют **BSWOP**.

(EAX) = 12345678h
BSWOP EAX ; (EAX) = 78563412h

К командам пересылки относят:

Команды конвертирования:

CBW ; безадресная команда, (AL) → AX.

CWD ;(AX) → DX:AX

CW**E** ; (AX) → EAX (для i386 и выше)

CDF ; (EAX) → EDX:EAX (для i386 и выше)

Команды условной пересылки CMOVxx

CMOVL AL, BL ; если (AL) < (BL), то (BL) → (AL)

Загрузка адреса.

LEA OP1, OP2 ; вычисляет адрес OP2 и пересылает первому операнду, который может быть только регистром.

LEA BX, M[DX][DI]

Структура программы на Ассемблере

Ассемблер – это язык программирования низкого уровня и программа, написанная на Ассемблере, должна пройти три этапа обработки на компьютере, как и программа, написанная на любом другом языке программирования.

I этап - преобразование исходного модуля в объектный – ассемблирование. Исходных модулей может быть 1 или несколько.

II этап - с помощью программы редактора связей объектные модули объединяются в загрузочный, исполняемый модуль.

III этап – выполнение программы.

Существует два типа исполняемых модулей (исполняемых файлов): exe-файл (<имя>.exe) и com-файл (<имя>.com). В результате выполнения второго этапа получается исполняемый exe-файл, чтобы получить com-файл, необходимо выполнить еще один этап обработки - преобразование exe-файла в com-файл.

Исходный файл на Ассемблере состоит из команд и директив. Команды преобразуются в машинные коды, реализующие алгоритм решения задачи. Директивы описывают, каким образом необходимо выполнять ассемблирование и объединение модулей. Они описывают форматы данных, выделяемые области памяти для программ и т.д.

Команды и директивы в Ассемблере

Команда на Ассемблере состоит из четырех полей:

[<имя>[:]] <код операции> [<операнды>] [;комментарии]

Поля отделяют друг от друга хотя бы одним пробелом. В квадратных скобках указаны необязательные поля, все поля, кроме <код операции>, могут отсутствовать. <имя> - символическое имя Ассемблера. Имя используется в качестве метки для обращения к этой команде, передачи управления на данную команду. [:] после имени означает, что метка является внутренней. Код операции определяет какое действие должен выполнить процессор. Поле <операнды> содержит адреса данных, или данные, участвующие в операции, а также место расположения результатов операции. Операндов может быть от 1 до 3, они отделяются друг от друга запятой.

Комментарии отделяются кроме пробела еще и ";" и могут занимать всю строку или часть строки.

Например:

JMP M1

; команда безусловной передачи управления на команду с меткой **M1**.

----------/-------/----------

M1: MOV AX, BX

; пересылка содержимого регистра **BX** в регистр **AX.**

В комментарии будем записывать в виде (BX) ____AX

----------/-------/----------

Команды и директивы в Ассемблере

Директива, как и команда, состоит из четырех полей:

[<имя>] <код псевдооперации> <операнды> [;комментарии]

Здесь <имя> - символическое имя Ассемблера,

<код псевдооперации> - определяет назначение директивы.

Операндов может быть различное количество и для одной директивы.

Например:

M1 DB 1, 0, 1, 0, 1 ; директива **DB** определяет 5 байтов памяти и заполняет их 0 или 1 соответственно, адрес первого байта – M1.

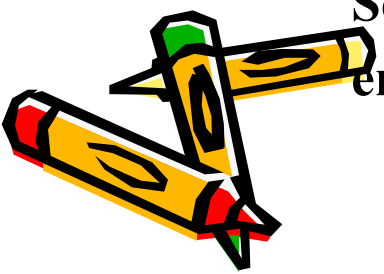
M2 DB ?,?,? ; директива **DB** определяет три байта памяти ничем их не заполняя, адрес первого – M2.

Proc ; директива начала процедуры,

endp ; директива конца процедуры,

Segment ; директива начала сегмента,

ends ; директива конца сегмента.



Исходный модуль на Ассемблере – последовательность строк, команд, директив и комментариев.

Исходный модуль просматривается Ассемблером, пока не встретится директива **end**. Обычно программа на Ассемблере состоит из трех сегментов: сегмент стека, сегмент данных, сегмент кода.

```
; сегмент стека
    Sseg Segment…
    -----/-------
    Sseg ends
; сегмент данных
    Dseg Segment…
    -----/-------
    Dseg ends
; сегмент кода
    Cseg Segment…
    -----/-------
    Cseg ends
end start
```

Каждый сегмент начинается директивой начала сегмента - **Segment** и заканчивается директивой конца сегмента - **ends**, в операндах директивы **Segment** содержится информация о назначении сегмента.

Назначение сегментов

В кодовом сегменте специальная директива….

ASSUME SS:SSeg, DS:DSeg, CS:CSeg, ES:DSeg;

на DSeg ссылаются и DS, и ES.

Кодовый сегмент оформляется как процедура, это может быть одна процедура или несколько последовательных процедур, или несколько вложенных процедур.

Структура кодового сегмента с использованием двух вложенных процедур выглядит следующим образом:

```
Cseg Segment…
ASSUME SS:SSeg, DS:DSeg, CS:CSeg
pr1 Proc
--------/--------
      pr2 Proc
-------/-----
      pr2 endp
-------/-----
pr1 endp
Cseg   ends
```

Назначение сегментов

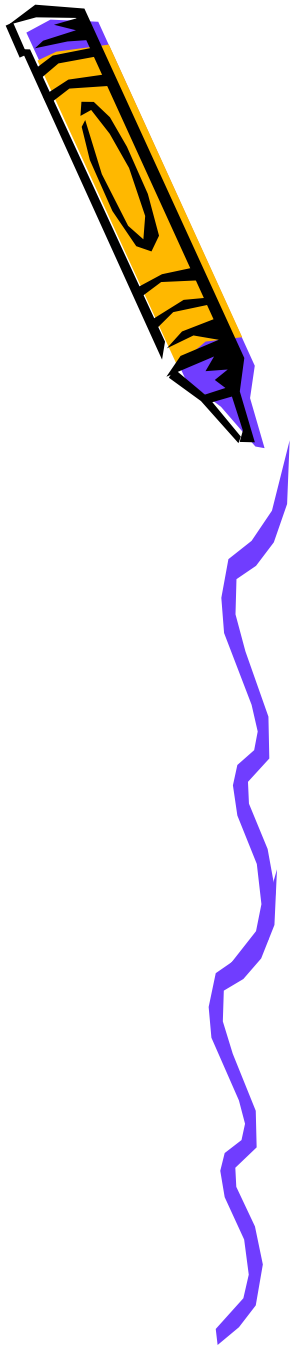
В сегменте стека выделяется место под стек.

В сегменте данных описываются данные, используемые в программе, выделяется место под промежуточные и окончательные результаты.

Кодовый сегмент содержит программу решения поставленной задачи.

```
        ; Prim1.ASM
; сегмент стека
        Sseg Segment…
                DB 256 DUP(?)
        Sseg ends
; сегмент даннх
        Dseg Segment…
            X DB ‘A’
            Y DB ‘B’
            Z DB ‘C’
        Dseg ends
;
```

```
Cseg Segment…
        ASSUME SS:SSeg, DS:DSeg, CS:CSeg
        Start Proc FAR
                Push DS
          Push AX
          MOV DX, DSeg
          MOV DS, DX
          CALL Main
          Ret
        Start endp
        Main Proc NEAR
          ADD AL, X
          MOV AX, Y
          --------/------
          Ret
        Main endp
        Cseg ends
                end Start
```



Структура программы

- Строки 1, 5, 11 – это комментарии.
- Кодовый сегмент содержит две последовательные процедуры. Первая процедура – внешняя, о б этом говорит параметр FAR.
- Строки 15 -18 – реализуют связь с операционной системой и определяют адрес начала сегмента данных.
- Строка 19 – это обращение к внутренней процедуре Main, строка 20, команда Ret – возврат в ОС.
- Main – внутренняя процедура, о чем говорит параметр NEAR в директиве начала процедуры Proc.
- Директива end имеет параметр Start, определяющий точку входа в программу, т.е. команду, с которой должно начинаться выполнение программы.
- Внутренняя процедура – это процедура, к которой можно обратиться только из того сегмента, в котором она содержится. К внешней процедуре можно обратиться из любого сегмента. По умолчанию (если в директиве начала процедуры параметр отсутствует) процедура является внутренней.

Слова, константы, выражения, переменные

Символические имена в Ассемблере могут состоять из строчных и прописных букв латинского алфавита, цифр от 0 до 9 и некоторых символов ‘_’, ‘.’, ‘?’, ….

В программе на Ассемблере могут использоваться константы пяти типов: целые двоичные, десятичные, шестнадцатеричные, действительные с плавающей точкой, символьные.

Целые двоичные – это последовательности 0 и 1 со следующим за ними символом **‘b’, например, 10101010b или 11000011b.**

Целые десятичные - это обычные десятичные числа, возможно заканчивающиеся буквой d, например, **– 125 или 78d.**

Целые шестнадцатеричные числа – должны начинаться с цифры и заканчиваются всегда ‘h’, если первый символ – **‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’,** то перед ним необходимо поставить 0, иначе они будут восприниматься как символические имена.

Числа действительные с плавающей точкой представляются в виде мантиссы и порядка, например, **– 34.751e+02 – это 3475.1 или 0.547e-2 – это 0.00547.**

Строковые данные – это последовательности символов, заключенные в апострофы или двойные кавычки, например,

'abcd', 'a1b2c3', ‘567'.

Слова, константы, выражения, переменные

Также, как и в языках высокого уровня, в Ассемблере могут использоваться именованные константы. Для этого существует специальная директива **EQU**. Например,

M EQU 27 ; директива EQU присваивает имени M значение 27.

Переменные в Ассемблере определяются с помощью директив определения данных и памяти, например,

v1 DB ?

v2 DW 34

или с помощью директивы ‘ = ’

v3 = 100

v3 = v3+1

Константы в основном используются в директивах определения или как непосредственные операнды в командах.

Выражения в Ассемблере строятся из операндов, операторов и скобок.

Операнды – это константы или переменные.

Операторы – это знаки операций (арифметических, логических, отношений и некоторых специальных)

Слова, константы, выражения, переменные

Арифметические операции: **‘+’, ‘-‘, ‘*’, ‘/’, mod**.

Логические операции: **NOT, AND, OR, XOR**.

Операции отношений: **LT(<), LE(≤), EQ(=), NE(≠), GT(>), GE(≥).**

Операции сдвига: сдвиг влево **(SHL),** сдвиг вправо **(SHR)**

Специальные операции: **offset и PTR**

offset <имя> - ее значением является смещение операнда, а операндом может быть метка ли переменная;

PTR – определяет тип операнда:

BYTE = 1 байт,

WORD = 2 байт,

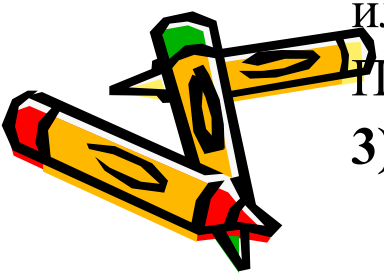
DWORD = 4 байт,

FWORD = 6 байт,

QWORD = 8 байт,

TWORD = 10 байт;

или тип вызова: **NEAR** – ближний, **FAR** – дальний.



Примеры выражений: **1) 10010101b + 37d 2) OP1 LT OP2**

3) (OP3 GE OP4) AND (OP5 LT OP6) 4) 27 SHL 3 ;

Директива определения

Общий вид директивы определения следующий

[<имя>] DX <операнды> <; комментарии>,

где X это B, W, D, F, Q или T.

В поле операндов может быть ‘?’, одна или несколько констант, разделенных запятой. Имя, если оно есть, определяет адрес первого байта выделяемой области. Директивой выделяется указанное количество байтов ОП и указанные операнды пересылаются в эти поля памяти. Если операнд – это ‘?’, то в соответствующее поле ничего не заносится.

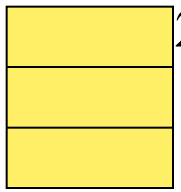
Пример:

R1 DB 0, 0, 0; выделено 3 поля, заполненных 0.

R1

R1+1

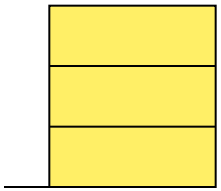
R2 DB ?, ?, ?R2



2

R2+1

R2+2



Директива определения

1) Если операндом является символическое имя IM1, которое соответствует смещению в сегменте 03AC1h, то после выполнения

M DD IM1

будет выделено 4 байта памяти. Адрес – M. Значение - 03AC1h.

2) Если необходимо выделить 100 байтов памяти и заполнить 1, то это можно сделать с помощью специального повторителя DUP.

D DB 100 DUP (1)

3) Определение одномерного массива слов, адрес первого элемента массива – имя **MAS,** значение его 1.

MAS DW 1, 7, 35, 75, 84

4) Определение двумерного массива:

```
Arr   DB      7, 94, 11, -5
      DB      5,  0,   1,  2
      DB      -5, 0, 15, 24
```

5) **Const EQU 100**

D DB Const DUP (?) ; выделить 100 байтов памяти. В директиве определения байта (слова) максимально допустимая константа – 255 (65535).

С помощью директивы определения байта можно определить строковую константу длинной 255 символов, а с помощью определения слова можно определить строковую константу, которая может содержать не более двух символов.

Команда прерывания Int, команды работы со стеком

С помощью этой команды приостанавливается работа процессора, управление передается DOC или BIOS и после выполнения какой-то системной обрабатывающей программы, управление передается команде, следующей за командой INT.

Выполняемые действия будут зависеть от операнда, параметра команды INT и содержания некоторых регистров.

Например, чтобы вывести на экран ‘!’ необходимо:

MOV AH, 6
MOV DL, ‘!’
INT 21h ; ….

Стек определяется с помощью регистров SS и SP(ESP).

Сегментный регистр SS содержит адрес начала сегмента стека.

ОС сама выбирает этот адрес и пересылает его в регистр SS.

Регистр SP указывает на вершину стека и при добавлении элемента стека содержимое этого регистра уменьшается на длину операнда.

Добавить элемент в стек можно с помощью команды

PUSH <операнд>,

где операндом может быть как регистр, так и переменная.

Удалить элемент с вершины стека можно с помощью операции

POP <операнд>,

Команда прерывания Int, команды работы со стеком

Для i186 и > PUSHA/ POPA позволяют положить в стек, удалить содержимое всех регистров общего назначения в последовательности AX, BX, CX, DX, SP, BP, SI, DI Для i386 и > PUSHAD/ POPAD позволяют положить в стек, удалить содержимое всех регистров общего назначения в последовательности EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI

К любому элементу стека можно обратиться следующим образом

MOV BP, SP; (SP)→BP

MOV AX, [BP+6]; (SS:(BP+6))→AX.

Пример программы использующей директивы пересылки содержимого 4 байтов памяти и вывод на экран.

```
TITLE Prim.asm
Page   , 120
; описание сегмента стека
SSeg Segment Para stack ‘stack’
     DB 100h DUP (?)
SSeg ends
```

Пример программы…

```
; описание сегмента данных
DSeg Segment Para Public ‘Data’
      DAN   DB  ’1’, ‘3’, ‘5’, ‘7’
      REZ    DB  4 DUP (?)
DSeg ends
; кодовый сегмент оформлен как одна внешняя процедура, к
; ней обращаются из отладчика
CSeg Segment Para Public ‘Code’
      ASSUME SS:SSeg, DS:DSeg, CS:CSeg
Start Proc FAR
      PUSH DS
      XOR AX, AX
      PUSH AX
                  MOV AX, DSeg;
                  MOV DS, AX;
```

```
; пересылка данных в обратной последовательности с выводом на экран
        MOV AH, 6
        MOV DL, DAN + 3
        MOV REZ, DL
        Int 21h  ; вывели на экран ‘7’
        MOV DL, DAN + 2
        MOV REZ + 1, DL
        Int 21h
        MOV DL, DAN +1
        MOV REZ + 2, DL
        Int 21h
        MOV DL, DAN
        MOV REZ + 3, DL
        Int 21h
;
        MOV AH, 4CH
        Int 21h
            Start endp
            CSeg ends
            end Start
```

Директива TITLE….. директива NAME….

Директива сегмента

Общий вид

<имя> Segment <ReadOnly> <выравнивание> <тип> <размер> <’ класс’>

Любой из операндов может отсутствовать.

1) Если есть <ReadOnly>, то будет выведено сообщение об ошибке при попытке записи в сегмент.

2) Операнд <выравнивание> определяет адрес начала сегмента.

- BYTE - адрес начала сегмента может быть любым,
- WORD - адрес начала сегмента кратен 2,
- DWORD - адрес начала сегмента кратен 4,
- Para - адрес начала сегмента кратен 16 – (по умолчанию),
- Page - адрес начала сегмента кратен 256.

Директива сегмента

3) <тип> определяет тип объединения сегментов.

Значение **stack** указывается в сегменте стека, для остальных сегментов – **public**. Если такой параметр присутствует, то все сегменты с одним именем и различными классами объединяются в один последовательно в порядке их записи.

Значение **‘Common’** говорит, что сегменты с одним именем объединены, но не последовательно, а с одного и того же адреса так, что общий размер сегмента будет равен не сумме, а максимуму из них.

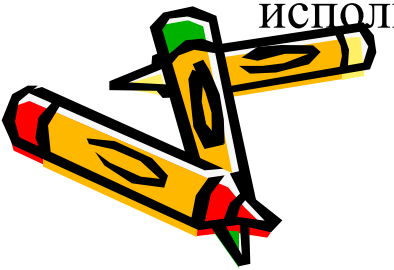
Значение IT <выражение> - указывает на то, что сегмент должен располагаться по фиксированному абсолютному адресу, определенному операндом <выражение>,

Значение ‘Private’ означает, что этот сегмент ни с каким другим объединяться не должен.

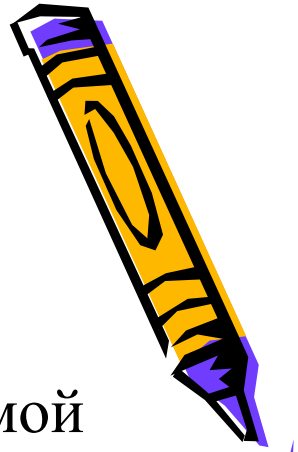
4) <разрядность> use 16 – сегмент до 64 Кб,

use 32 – сегмент до 4 ГБ

5) ‘<класс>’ – с одинаковым классом сегменты располагаются в исполняемом файле последовательно друг за другом.



Точечные директивы



В программе на Ассемблере могут использоваться упрощенные (точечные) директивы.

.MODEL - директива, определяющая модель выделяемой памяти для программы.

Модель памяти определяется параметром:

tiny – под всю программу выделяется 1 сегмент памяти,

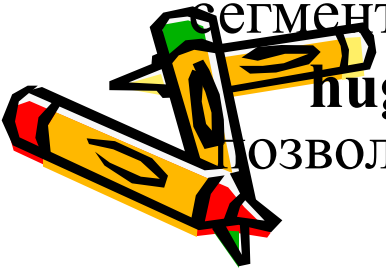
small – под данные и под программу выделяются по одному сегменту,

medium – под данные выделяется один сегмент, под программу выделяется несколько сегментов,

compact – под программу выделяется один сегмент, под данные выделяется несколько сегментов,

large - под данные и под программу выделяются по n сегментов,

huge – позволяет использовать сегментов больше, чем позволяет ОП.



Точечные директивы

Пример использования точечных директив в программе на Acc-ре.

```
.MODEL small
.STACK 100h
.DATA
St1  DB  ‘Line1’, ‘$’
St2  DB  ‘Line2’, ‘$’
St3  DB  ‘Line3’, ‘$’
.CODE
begin:   MOV AH, 9  ; 9 -  номер функции  вывода строки на экран
MOV DX, offset St1  ; адрес ST1 должен содержаться в регистре DX
                Int 21h
                MOV DX, offset St2
                Int 21h
                MOV AH, 4CH
                Int 21h
end begin
```

Точечные директивы

‘$’ – конец строки, которую необходимо вывести на экран

В результате выполнения программы:

Line1 Line2 Line3.

Если необходимо вывести

Line1

Line2

Line3,

то в сегмент данных необходимо внести изменения

St1 DB ‘Line1’, 13, 10, ‘$’

St2 DB ‘Line2’, 0Dh, 0Ah, ‘$’

St3 DB ‘Line3’, ‘$’

Com-файлы

После обработки компилятором и редактором связей получаем exe-файл, который содержит блок начальной загрузки, размером не менее 512 байт, но существует возможность создания другого вида исполняемого файла, который может быть получен на основе exe-файла с помощью системной обрабатывающей программы EXE2BIN.com или его можно создать с помощью среды разработки. Но не из всякого exe-файла можно создать com-файл. Исходный файл, для которого можно создать com-файл, должен удовлетворять определенным требованиям.

Отличия exe-файла от com-файла:

В com-файлах отсутствует блок начальной загрузки и следовательно он занимает меньше места, чем exe-файл.

exe-файл может занимать произвольный объем ОП. com-файл может занимать только один сегмент памяти.

Стек создается автоматически ОС, поэтому у пользователя нет необходимости выделять для него место. Данные располагаются там же, где и программа.

Т.к. вся программа содержится в одном сегменте, перед выполнением программы все сегментные регистры содержат в качестве значения адрес префикса программного сегмента – PSP

Com-файлы

PSP - 256 байтный блок, который содержится как в exe-файле, так и в com-файле, и т.к. адрес первой исполняемой команды отстоит на 256 (100h) байтов от адреса начала сегмента, то сразу после директивы **ASSUME** используется специальная директива **org 100h**, осуществляющая обход префикса программного сегмента

Пример создания com-файла.

1)
```
TITLE Prog_Com-файл
Page  60 , 85
CSeg Segment Para ‘Code’
ASSUME SS:CSeg, DS:CSeg, CS:CSeg
Org 100h
Start:    JMP Main
      St1  DB  ‘String1’, 13, 10, ‘$’
      St2  DB  ‘String2’,  ‘$’
```

```
Main        Proc
            MOV AH, 9
            LEA DX, St1
            Int 21h
            LEA DX, St2
            Int 21h
            MOV AH, 4CH
            Int 21h
Main        endp
CSeg        ends
            end Start
```

2)

```
            .Model tiny
            .Code
             JMP Met
          St1   DB 'String1', '$'
          Met:  MOV AH, 09h
                LEA  DX, St1
                Int 21h
                MOV AH, 4Ch
                Int 21h
                end Met
```

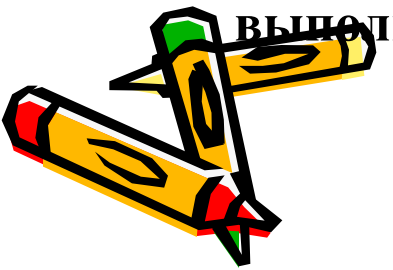
Примеры com-файлов

3) ----------

```
Beg     Proc
        MOV AH, 9
        LEA DX, St1
        Int 21h
        MOV AH, 4Ch
        Int 21h
Beg     endp
St1  DB  'String1', '$'
        end beg
```

Замечания:

- **Не каждый исходный файл удовлетворяет требованиям com-файла.**
- **Небольшие по объему программы рекомендуется оформлять как com-файлы.**
- **Исходный файл, написанный как com-файл, не может быть выполнен как exe-файл.**



Арифметические операции

Сложение (вычитание) беззнаковых чисел выполняется по правилам аналогичным сложению (вычитанию) по модулю 2^k принятым в математике….В информатике, если в результате более k разрядов, то к+1-й пересылается в CF.

$X + Y = (X + Y) \bmod 2^k = X + Y$ и CF = 0, если $X + Y < 2^k$

$X + Y = (X + Y) \bmod 2^k = X + Y - 2^k$ и CF = 1, если $X + Y >= 2^k$

Пример, работая с байтами, получим:

$250 + 10 = (250 + 10) \bmod 2^8 = 260 \bmod 256 = 4$

$260 = 1\ 0000\ 0100_2$, CF = 1, результат $- 0000\ 0100_2 = 4$

$X - Y = (X - Y) \bmod 2^k = X - Y$ и CF = 0, если $X >= Y$

$X - Y = (X - Y) \bmod 2^k = X + 2^k - Y$ и CF = 1, если $X < Y$

Пример: в байте

$1 - 2 = 2^8 + 1 - 2 = 257 - 2 = 255$, CF = 1

Арифметические операции

Сложение (вычитание) знаковых чисел сводится к сложению (вычитанию) с использованием дополнительного кода.

$$X = 10^n - |X|$$

В байте: -1 = 256 – 1 = 255 = 11111111_2

-3 = 256 – 3 = 253 = 11111101_2

3 + (-1) = (3 + (-1)) mod 256 = (3+255) mod 256 = 2

1 + (-3) = (1 + (-3)) mod 256 = 254 = 1$1111110_2$

Ответ получили в дополнительном коде, следовательно результат получаем в байте по формуле $X = 10^n - |X|$ **, т.е.**

x = 256 – 254 = |2| и знак минус. Ответ -2.

Переполнение происходит, если есть перенос из старшего цифрового в знаковый, а из знакового нет и наоборот, тогда

OF = 1. Программист сам решает какой флажок анализировать

OF или CF, зная с какими данными он работает.

Арифметические операции изменяют значение флажков

OF, CF, SF, ZF, AF, PF.

Сложение и вычитание в Ассемблере

Арифм-ие операции изменяют значение флажков OF, CF, SF, ZF, AF, PF.

В Ассемблере команда ‘+’

ADD OP1, OP2 ; (OP1) + (OP2) → OP1

ADC OP1, OP2 ; (OP1) + (OP2) + (CF) → OP1

XADD OP1, OP2; i486 и >

(OP1) ↔ (OP2) (меняет местами), (OP1) + (OP2) → OP1

INC OP1 ; (OP1) + 1 → OP1

В Ассемблере команда ‘-‘

SUB OP1, OP2 ; (OP1) – (OP2) → OP1

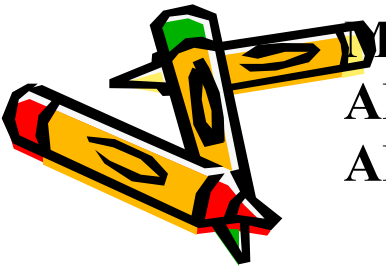
SBB OP1, OP2 ; (OP1) – (OP2) – (CF) → OP1

DEC OP1 ; (OP1) – 1 → OP1.

Примеры:

X = 1234AB12h, Y = 5678CD34h, X + Y =

```
MOV AX, 1234h
MOV BX, 0AB12h
MOV CX, 5678h
MOV DX, 0CD34h
ADD BX, DX
ADC AX, CX
```



Сложение и вычитание в Ассемблере

X – Y = **SUB BX, DX**
SBB AX, CX

В командах сложения и вычитания можно использовать любые способы адресации:

ADD AX, mas[SI]
SUB DX, mas[BX][DI]
ADD CX, 32h

Пример1:

MOV AL, 95h

ADD AL, 82h

95h + 82h = 117h $95 = 10010101_2$ $82 = 10000010_2$

$10010101_2 + 10000010_2 = 1\ 0001\ 0111_2$,

CF = 1, OF = 1, SF = 0, ZF = 0, AF = 0, PF = 1.

```
  10010101
+ 10000010
-----------
  1 00010111
```

Пример2:

MOV AL, 9h

SUB AL, 5h

9h – 5h = 4h 5 = 00000101 -5 = 11111011 9 = 00001001

9 + (-5) = 11111011 + 00001001 = 1 0000 0100

CF = 1, OF = 0, SF = 0, ZF = 0, AF = 1, PF = 0.

Умножение и деление в Ассемблере

Умножение беззнаковых чисел.

MUL OP2 ; (OP2)*(AL) ∨ (AX) ∨ (EAX) → AX ∨ DX:AX ∨ EDX:EAX

Умножение знаковых чисел.

IMUL OP2; аналогично MUL

IMUL OP1, OP2 ; i386 и > **IMUL op1, op2, op3 ;** i186 и >

OP1 всегда регистр, OP2 – непосредственный операнд, регистр или память.

При умножении результат имеет удвоенный формат по отношению к сомножителям. Иногда мы точно знаем, что результат может уместиться в формат сомножителей, тогда мы извлекаем его из AL, AX, EAX.

Размер результата можно выяснить с помощью флагов OF и CF.

Если OF = CF = 1, то результат занимает двойной формат,

и OF = CF = 0 , результат умещается в формат сомножителей.

Остальные флаги не изменяются.

Деление беззнаковых чисел: Деление знаковых чисел.

DIV OP2 ; OP2 = r ∨ m **IDIV OP2** ; OP2 = r ∨ m

(AX) ∨ (DX:AX) ∨ (EDX:EAX) делится на указанный операнд и результат помещается в AL ∨ AX ∨ EAX,

остаток помещается в AH ∨ DX ∨ EDX.

Умножение и деление в Ассемблере

Значение флагов не меняется, но может наступить деление на 0 или переполнение, если 1) op2 = 0, 2) частное не умещается в отведенное ему место. Например:

MOV AX, 600

MOV BH, 2

DIV BH ; 600 div 2 = 300 - не умещается в AL.

При использовании арифметических операций необходимо следить за размером операндов…..

Пример:

Необходимо цифры целого беззнакового байтового числа N записать в байты памяти, начиная с адреса D как символы. N - (abc)

c = N mod 10

b = (N div 10) mod 10

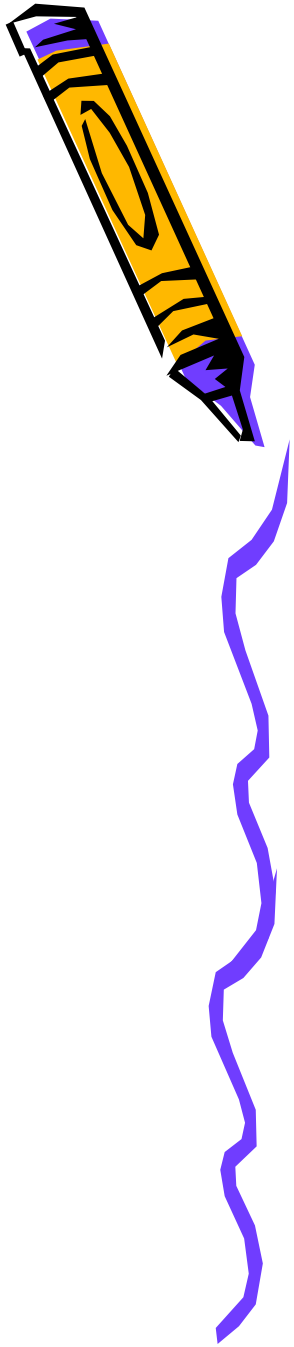
a = (N div 10) div 10

Перевод в символы: код(i) = код (‘0’) + i

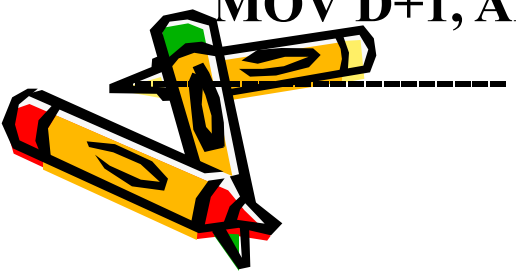
N DB ?

D DB 3 Dup (?)

Умножение и деление в Ассемблере



```
MOV BL, 10 ; делитель
MOV AL, N ; делимое
MOV AH, 0 ;  расширяем делимое до слова
; или CBW AH конвертируем до слова
DIV BL   ; A L= ab, AH = c
ADD AH, ‘0’
MOV D+2, AH
MOV AH, 0
DIV BL        ; AL = a, AH = b
ADD AL,  ‘0’
MOV D, AL
ADD AH, ‘0’
MOV D+1, AH
```



Директивы внешних ссылок

Директивы внешних ссылок позволяют организовать связь между различными модулями и файлами, расположенными на диске

Public <имя> [, <имя>,…,<имя>] –

определяет указанные имена как глобальные величины, к которым можно обратиться из других модулей. <имя> – имена меток и переменных, определенных с помощью директивы **‘=’ и EQU**.

Если некоторое имя определено в модуле А как глобальное и к нему нужно обратиться из других модулей В и С, то в этих модулях должна быть директива вида

EXTRN <имя>:<тип> [,<имя>:<тип>…]

Здесь имя то же, что и в **Public**, а тип определяется следующим образом:

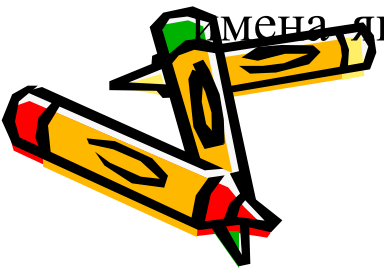
если <имя> – это имя переменной, то типом может быть:

BYTE, WORD, DWORD, FWORD, QWORD, TWORD;

если <имя> – это имя метки, то типом может быть

NEAR, FAR.

Директива **EXTRN** говорит о том, перечисленные имена являются внешними для данного модуля.



Директивы внешних ссылок

Пример:
В модуле А содержится:

Public TOT
-------/-------
TOT DW 0 ;

чтобы обратиться из В и С к имени TOT, в них должна быть директива

EXTRN TOT:WORD

В Ассемблере есть возможность подключения на этапе ассемблирования модулей, расположенных в файлах на диске

INCLUDE <имя файла>

Пример:
INCLUDE C:\WORK\Prim.ASM
Prim.ASM, расположенный в указанной директории, на этапе ассемблирования записывается на место этой директивы.

Команды управления

Команды управления позволяют изменить ход вычислительного процесса.

К ним относятся команды безусловной передачи управления, команды условной передачи управления, команды организации циклов.

Команды безусловной передачи управления имеют вид

JMP <имя>,

где имя определяет метку команды, которая будет выполняться следующей за этой командой. Эта команда может располагаться в том же кодовом сегменте, что и команда **JMP** или в другом сегменте.

JMP M1 ; по умолчанию M1 имеет тип NEAR

Если метка содержится в другом сегменте, то в том сегменте, в который передается управление, должно быть **Public M1**, а из которого –

EXTRN M1: FAR.

Кроме того, передачу можно осуществлять с использованием прямой адресации (**JMP M1**) или с использованием косвенной адресации (**JMP [BX]**).

Команда, осуществляющая близкую передачу, занимает 3 байта, а дальняя – 5 байтов. А если передача осуществляется не далее как на -128 или 127 байтов, то можно использовать команду безусловной передачи данных, занимающую 1 байт.

Команды безусловной передачи управления

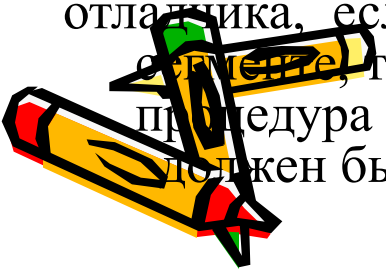
```
        ADD AX, BX
        JMP Short M1
M2: ------/-------
M1: MOV AX, CX
----------/----------
```

К командам безусловной передачи данных относятся команды обращения к подпрограммам, процедурам, и возврат из них.

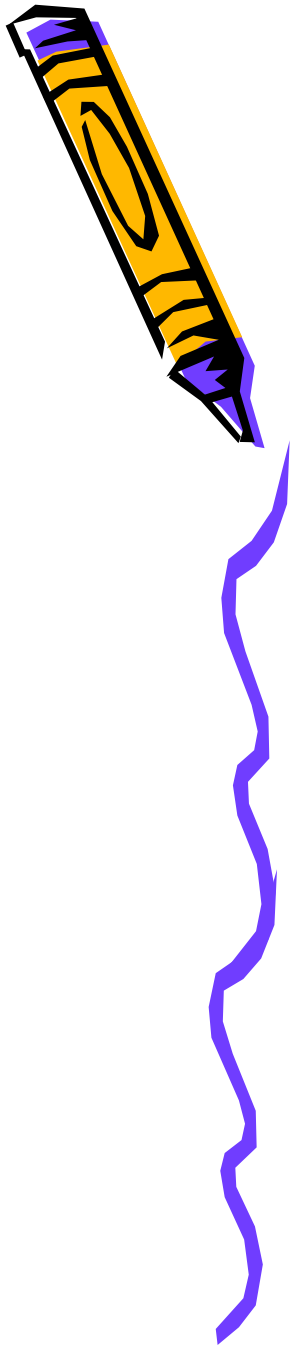
Процедура обязательно имеет тип дальности и по умолчанию тип NEAR, а FAR необходимо указывать.

```
PP Proc FAR
------/-------
PP endp
```

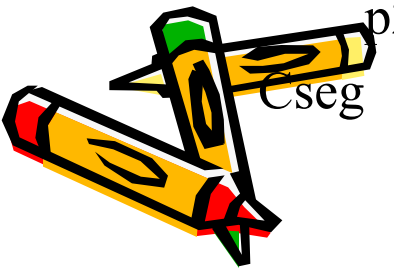
Процедура типа NEAR может быть вызвана только из того кодового сегмента, в котором содержится ее описание. Процедура типа FAR может быть вызвана из любого сегмента. Поэтому тип вызова функции (дальность) определяется следующим образом: главная программа всегда имеет тип FAR, т.к. обращаются к ней из ОС или отладчика, если процедур несколько и они содержатся в одном кодовом сегменте, то все остальные, кроме главной, имеют тип NEAR. Если процедура описана в кодовом сегменте с другим именем, то у нее должен быть тип FAR.



Процедуры Near и Far



```
1)      Cseg    segment….
                assume …..
            p1  proc far
            ------------------------
                call p2
            m:  mov AX, BX
            -------------------------
                ret
            p1  endp
              p2    proc near
                m1: mov CX, DX
                ------------------------
                ret
            p2  endp
        Cseg    ends
```



Процедуры Near и Far

```
2) extrn p2: far                    public p2
      cseg     segment……                cseg1    segment…..
               assume ……                assume ……
      p1       proc far       p2    proc far
      ----------------------          ---------------------
               call p2                     ret
   ------------------------            p2   endp
               ret              cseg1      ends
      p1       endp
      cseg     ends
```

Команды безусловной передачи управления

Команда вызова процедуры:

CALL <имя> ;

Адресация может быть использована как прямая, так и косвенная.

При обращении к процедуре типа NEAR в стеке сохраняется адрес возврата, адрес команды, следующей за CALL содержится в IP или EIP.

При обращении к процедуре типа FAR в стеке сохраняется полный адрес возврата CS:EIP.

Возврат из процедуры реализуется с помощью команды **RET.**

Она может иметь один из следующих видов:

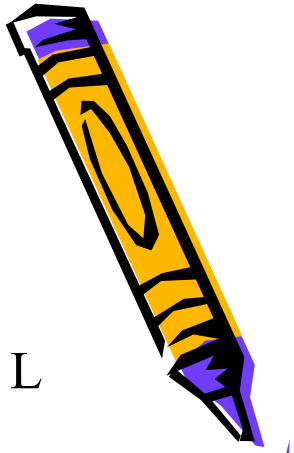
RET [n] ; возврат из процедуры типа NEAR, и из процедуры типа FAR

RETN [n] ; возврат только из процедуры типа NEAR

RETF [n] ; возврат только из процедуры типа FAR

Параметр n является необязательным, он определяет какое количество байтов удаляется из стека после возврата из процедуры.

Примеры прямого и косвенного перехода



1) ----------

a dw L ; значением a является смещение для переменной L

jmp L ; прямой переход по адресу L

jmp a ; косвенный переход - goto (a) = goto L

---

2) ----------

mov DX, a ; значение a пересылается в DX

jmp DX ; косвенный переход - goto (DX) = goto L

---

3) ----------

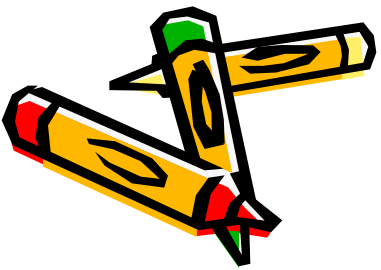
jmp z ; ошибка

Z DW L

3) jmp word ptr z

---

z DW L



Команды условной передачи управления

Команды условной передачи управления можно разделить на 3 группы:

- команды, используемые после команд сравнения
- команды, используемые после команд, отличных от команд сравнения, но реагирующие на значения флагов

  JZ/JNZ
  JC/JNC
  JO/JNO
  JS/JNS
  JP/JNP
- команды, реагирующие на значение регистра CX

В общем виде команду условной передачи управления можно записать так: **Jx <метка>**

Здесь x – это одна, две, или три буквы, которые определяют условия передачи управления. Метка, указанная в поле операнда, должна отстоять от команды не далее чем

-128 ÷ +127 байт.

Команды условной передачи управления

Примеры:

JE M1 ; передача управления на команду с меткой M1, если ZF = 1

JNE M2 ; передача управления на команду с меткой M2, если ZF = 0

JC M3 ; передача управления на команду с меткой M3, если CF = 1

JNC M4 ; передача управления на команду с меткой M4, если CF = 0

ADD AX, BX

JC M

если в результате сложения CF = 1, то управление передается на команду с меткой M, иначе – на команду, следующую за JC

SUB AX, BX

JZ Met

если результатом вычитания будет 0, то ZF = 1 и управление передается на команду с меткой Met.

Часто команды передачи управления используются после команд сравнения **<метка> CMP OP1, OP2**

По этой команде выполняется (OP1) – (OP2) и результат никуда не посылается, формируются только флаги.

Команды условной и безусловной передачи управления

условие	Для беззнаковых чисел	Для знаковых чисел
>	JA	JG
=	JE	JE
<	JB	JL
>=	JAE	JGE
<=	JBE	JLE
<>	JNE	JNE

Команды управления

Команды условной передачи управления могут осуществлять только короткий переход, а команды безусловной передачи управления могут реализовать как короткую передачу так и длинную. Если необходимо осуществить условный дальний переход, то можно использовать jx вместе jmp следующим образом:

if AX = BX goto m следует заменить на:

if AX < > BX goto L

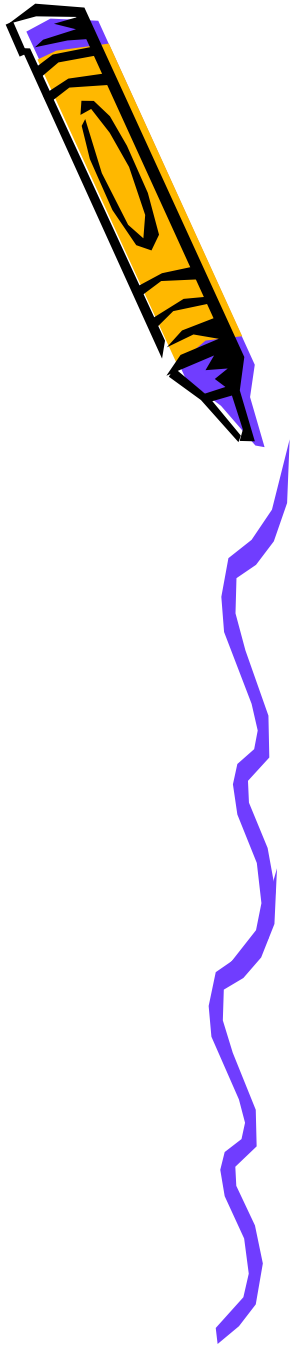
Goto m ; m – дальняя метка

L: ---------- ; L – близкая метка

На Ассемблере это будет так:

```
        cmp AX, BX
        jne L
        Jmp m
m:      ----------
L:      ----------
```

Команды управления



С помощью команд jx и jmp можно реализовать цикл с предусловием:

1) while x > 0 do S;

```
beg:    cmp x, byte ptr 0
        jle fin
        S
        jmp beg
fin:    ----------
```

и с постусловием:

2) do S while x > 0;

```
        beg:
                S
                cmp x, byte ptr 0
                jg beg
        fin:    _____
```

Команды для организации циклов

1) **loop <метка>**

2) **loope < метка > loopz < метка >**

3) **Loopne < метка > loopnz < метка >**

По команде в форме 1): (CX) = (CX) – 1 и если (CX) < > 0, ⟶ <метка>

По 2): (CX) = (CX) – 1 и если (CX) < > 0 и одновр-но ZF =1, ⟶ <метка>

Цикл завершается, если или (CX) = 0 или ZF=0 или (CX)=(ZF)=0

По 3): (CX) = (CX) – 1 и если (CX) < > 0 и одновр-но ZF=0, ⟶ <метка>

Выход из цикла осуществляется, если или (CX) = 0 или ZF=1 или одновременно (CX) = 0 и (ZF) = 1.

Примеры: 1)

```
        mov CX, 100
m1:     mov AX, DX
        ------------------------
        loop m1
```

2)

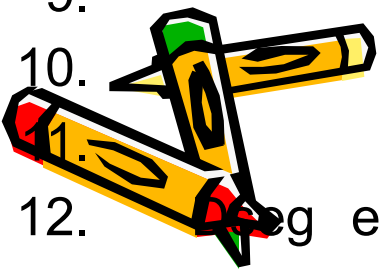
```
        mov SI, 0
        mov CX, 100
m1:     push CX
        ---------------------
m2:     inc SI
        pop CX
        loop m1
        m2:
```

2) если CX нужно использовать внутри цикла

Пример использования команд усл. перехода, сравнения и циклов

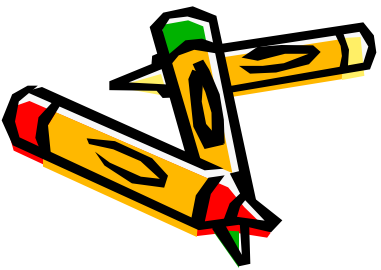
Дана матрица целых байтовых величин, размером 4*5, необходимо подсчитать количество нулей и заменить их числом 0FFh. Под стек отведем 256 байтов, программу оформим как две последовательные процедуры: внешняя (FAR)– это связь с ОС, внутренняя (NEAR) – решение поставленной задачи.

```
1.      ; prim.asm
2.      title prim.asm
3.      page , 132
4.      Sseg   segment para stack ‘stack’
5.                db 256 dup (?)
6.      Sseg   ends
7.      Dseg   segment para public ‘data’
8.      Dan    db 0,2,5,0,91 ; адрес первого элемента массива
9.                db  4,0,0,15,47  ;  имя - Dan
10.               db  24,15,0,9,55
11.               db  1,7,12,0,4
12.     Dseg   ends
```

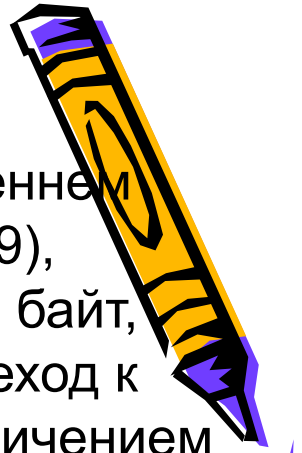


```
15. Cseg      segment para public ‘code’
16.             Assume cs: cseg, ds:dseg, ss:sseg
17.       start   proc far
18.               push DS ; для связи
19.               push AX ; с ОС
20.               mov BX, Dseg     ; загрузка адреса сегмента данных
21.               mov DS, BX ; в  регистр DS
22.               call main
23.               ret
24.       start   endp
25.       main    proc near
26.               mov BX, offset Dan
27.               mov CX, 4      ; количество повторений внешнего цикла
28.       nz1:    push CX
29.                     mov DL, 0     ;  счетчик нулей в строке матрицы
30.               mov SI, 0
31.               mov CX, 5   ; количество повторений внутреннего цикла
```

```
32.     nz2:    push CX
33.             cmp byte ptr [BX+SI], 0
34.             jne mz
35.             mov byte ptr [BX+SI], 0FFh
36.             inc DL
37.     mz:     inc SI
38.             pop CX
39.     kz2:    loop nz2
40.                     add DL, '0'    ; вывод на экран
41.                     mov AH, 6      ; количества нулей
42.                     int 21h
43.             add BX, 5      ; переход к следующей строке матрицы
44.             pop CX
45.     kz1:    loop nz1
46.             ret
47.     main    endp
48.     Cseg    ends
49.             end start
```



Организация циклов



Задача решена с помощью двух вложенных циклов, во внутреннем осуществляется просмотр элементов текущей строки (32-39), увеличение счетчика нулей и пересылка константы 0FFh в байт, содержащий ноль. Во внешнем цикле осуществляется переход к следующей строке очисткой регистра SI (строка 30) и увеличением регистра BX на количество элементов в строке (40).

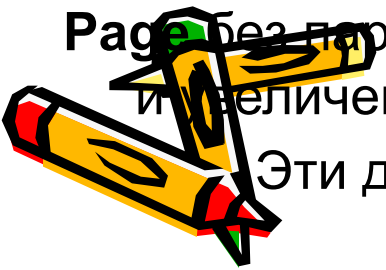
Физически последняя команда программы (49) в качестве параметра указывает метку команды, с которой необходимо начинать выполнение программы.

Директива **title** задает заголовок каждой странице листинга, заголовок может содержать до 60 символов.

Директива **page** устанавливает количество строк на странице листинга – 1-й параметр (здесь он отсутствует, значит берется значение по умолчанию 57) и количество символов в каждой строке (здесь 132, возможно от 60 до 132, по умолчанию – 80).

Page без параметров осуществляет перевод печати на новую страницу и увеличение на 1 номера страницы листинга.

Эти директивы могут отсутствовать.



Массивы в Ассемблере

Массивы в языке Ассемблер описываются директивами определения данных, возможно с использование конструкции повторения DUP.

Например, **x DW 30 dup (?)**

Так можно описать массив x, состоящий из 30 элементов длиной в слово, но в этом описании не указано как нумеруются элементы массива, т.е. это может быть **x[0..29] и x[1..30] и x[k..29+k].**

Если в задаче жестко не оговорена нумерация элементов, то в Ассемблере удобнее считать элементы от нуля, тогда адрес любого элемента будет записываться наиболее просто:

адрес (x[i]) = x + (type x) * i

В общем виде, когда первый элемент имеет номер k , для одномерного массива будет: **адрес (x[i]) = x + (type x) * (i – k)**

Для двумерного массива - **A[0..n-1, 0..m-1]** адрес (i,j) – го элемента можно вычислить так:

адрес (A[i,j]) = A + m * (type A) * i + (type A) *j

Массивы в Ассемблере

С учетом этих формул для записи адреса элемента массива можно использовать различные способы адресации.

Для описанного выше массива слов, адрес его i-го элемента равен:

x + 2*i = x + type (x) * i,

Т.е. адрес состоит из двух частей: постоянной x и переменной 2 * i, зависящей от номера элемента массива. Логично использовать адресацию прямую с индексированием: x – смещение, а 2*i – в регистре модификаторе SI или DI **x[i]**

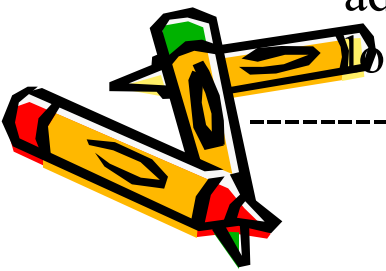
Для двумерного массива, например:

A DD n DUP (m Dup (?)) ; A[0..n - 1, 0..m - 1] получим

адрес (A[i,j]) = A + m * 4 * i + 4 *j,

Т.е. имеем в адресе постоянную часть A и две переменных **m * 4 * i** и **4 *j ,** которые можно хранить в регистрах. Два модификатора есть в адресации по базе с индексированием, например: **A[BX][DI].**

Фрагмент программы, в которой в регистр AL записывается количество строк матрицы X DB 10 dup (20 dup (?)), в которых начальный элемент повторяется хотя бы один раз.



```
--------------------------------
        mov AL, 0    ; количество искомых строк
        mov CX, 10   ; количество повторений внешнего цикла
        mov BX, 0    ; начало строки 20*i
m1:       push CX
        mov AH, X[BX]  ; 1-й элемент строки в AH
        mov CX, 19   ; количество повторений внутреннего цикла
        mov DI, 0    ; номер элемента в строке ( j )
m2: inc DI           ; j = j + 1
        cmp AH, X[BX][DI]   ; A[i,0] = A[i,j]
        loopne m2    ; первый не повторился? Переход на m2
        jne L      ; не было в строке равных первому? Переход на L
        inc AL     ; первый повторился, увеличиваем счетчик строк
         L:  pop CX  ; восстанавливаем CX для внешнего цикла
        add BX, 20   ; в BX начало следующей строки
        loop m1
-----------------------------
```

Команды побитовой обработки данных

К командам побитовой обработки данных относятся логические команды, команды сдвига, установки, сброса и инверсии битов.

Логические команды: and, or, xor, not. Для всех логических команд, кроме not, операнды одновременно не могут находиться в памяти, OF = CF = 0, AF – не определен, SF, ZF, PF определяются результатом команды.

and OP1, OP2 ; (OP1) логически умножается на (OP2), результат → OP1

Пример: (AL) = 1011 0011, (DL) = 0000 1111,

and AL, DL ; (AL) = 0000 0011

Второй операнд называют **маской**. Основным назначением команды and является установка в ноль с помощью маски некоторых разрядов первого операнда. Нулевые разряды маски обнуляют соответствующие разряды первого операнда, а единичные оставляют соответствующие разряды первого операнда без изменения. Маску можно задавать непосредственно в команде и можно извлекать из регистра или памяти.

Команды побитовой обработки данных

Например:

1) and CX, 0FFh ; маской является константа
2) and AX, CX ; маска содержится в регистре
3) and AX, TOT ; маска в ОП по адресу (DS) + TOT
4) and CX, TOT[BX+SI] ; …в ОП по адресу (DS) + (BX) + (SI) + TOT
5) and TOT[BX+SI], CX ; в ноль устанавливаются некоторые разряды ОП
6) and CL, 0Fh ; в ноль устанавливаются старшие 4 разряда регистра CL

Команда – **or OP1, OP2** ; результатом является…..

Эта команда используется для установки в 1 заданных битов 1-го операнда с помощью маски OP2. … Например:

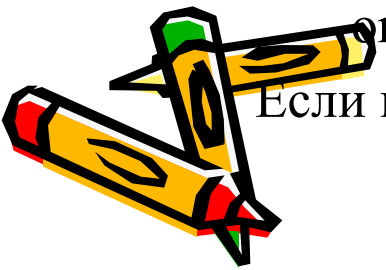
(AL) = 1011 0011, (DL) = 0000 1111

or AL, DL ; (AL) = 1011 1111 ……

В команде могут использоваться различные операнды:

or CX, 00FFh ; or TAM, AL ; or TAM[BX][DX], CX

Если во всех битах результата будет 0, то ZF = 1.



Команды побитовой обработки данных

Команда **xor OP1, OP2** ; 1 xor 1 = 0, 0 xor 0 = 0, в ост. сл. = 1

Например: (AL) = 1011 0011, маска = 000 01111

xor AL, 0Fh ; (AL) = 1011 1100

Команда **not OP** ; результат – инверсия значения операнда

Если (AL) = 0000 0000, **not AL** ; (AL) = 1111 1111

Значения флагов не изменяются.

Примеры.

1) xor AX, AX ; обнуляет регистр AX быстрее, чем mov и sub
2) xor AX, BX; меняет местами значения AX и BX

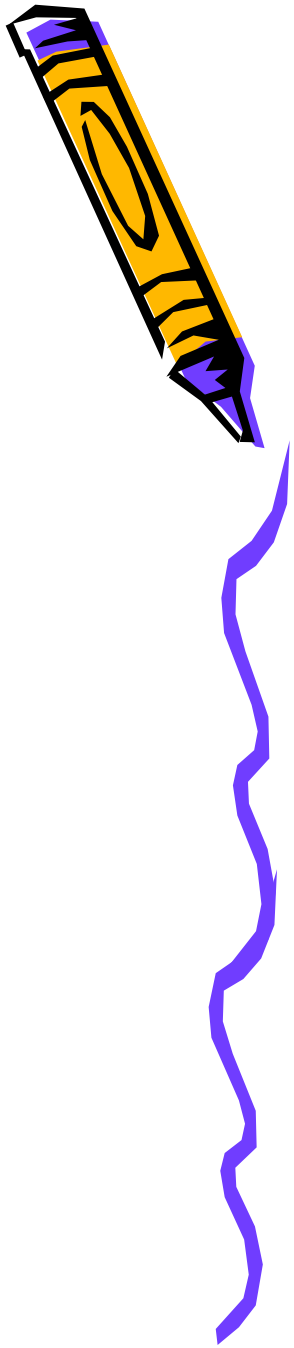
   xor BX, AX ; быстрее, чем команда

   xor AX, BX; xchg AX, BX
3) Определить количество задолжников в группе из 20 студентов. Информация о студентах содержится в массиве байтов

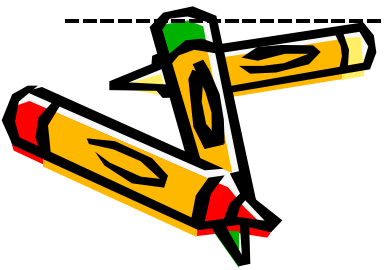
   X DB 20 DUP (?), причем в младших 4 битах каждого байта содержатся оценки, т.е. 1 – сдал экзамен, 0 – «хвост».

   В DL сохраним количество задолжников.

Команды побитовой обработки данных



```
---------------------------
        mov DL, 0
        mov SI, 0 ; i = 0
        mov  CX, 20 ;  количество повторений цикла
nz: mov AL, X[SI]
        and AL, 0Fh ; обнуляем старшую часть байта
        xor AL, 0Fh ;
        jz  m       ; ZF = 1, хвостов нет, передаем на повторение цикла
        inc DL    ; увеличиваем количество задолжников
m: inc SI       ;переходим к следующему студенту
        loop nz
        add DL, “0”
        mov AH, 6
        int 21h
---------------------------
```



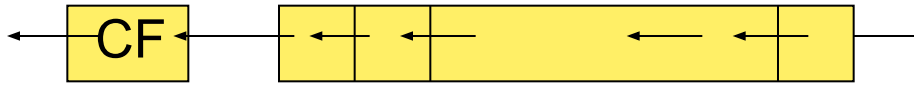
Команды побитовой обработки данных, команды сдвига

Формат команд арифметического и логического сдвига можно представить так: **s**XY OP1, OP2 ; <комментарий>

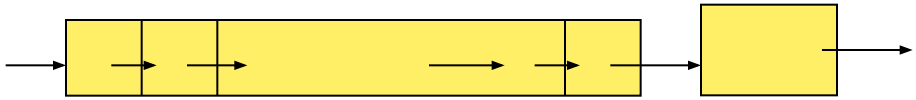
Здесь X - h или a, Y – l или r; OP1 – r или m, OP2 – d или CL

И для всех команд сдвига в CL используются только 5 младших разрядов, принимающих значения от 0 до 31. При сдвиге на один разряд:

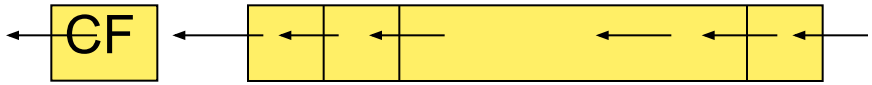
shl :



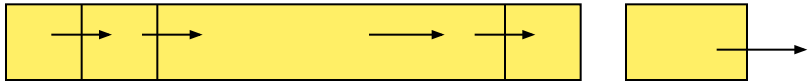
shr :



sal :



sar :



Здесь знаковый бит распространяется на сдвигаемые разряды. Например, (AL) = 1101 0101

sar AL, 1 ; (AL) = 1110 1010 и CF = 1

Сдвиги больше, чем на 1,эквивалентны соответствующим сдвигам на 1, выполненным последовательно.

Сдвиги повышенной точности для i186 и выше:

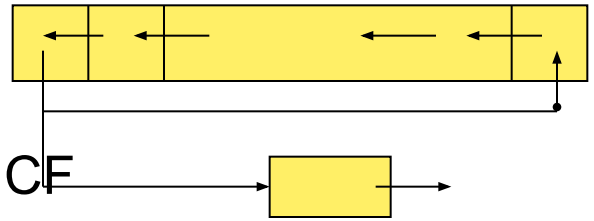
shrd OP1, OP2, OP3 ;

shld OP1, OP2, OP3 ;

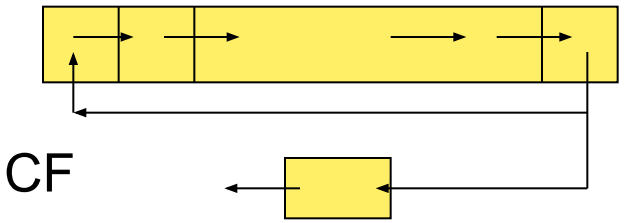
Содержимое первого операнда (OP1) сдвигается на (OP3) разрядов также, как и в командах shr и shl но бит, вышедший за разрядную сетку, не обнуляется, а заполняется содержимым второго операнда, которым может быть только регистр.

Циклические сдвиги:

rol op1, op2



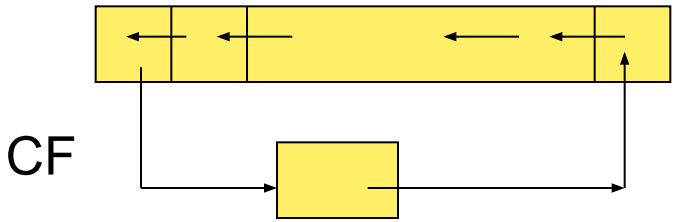
ror op1, op2



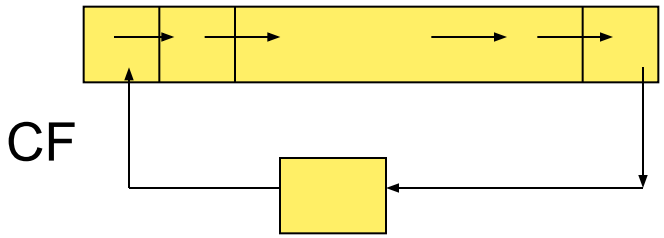
После выполнения команды циклического сдвига CF всегда равен последнему биту, вышедшему за пределы приемника.

Циклические сдвиги с переносом содержимого Флажка CF:

rcl OP1, OP2



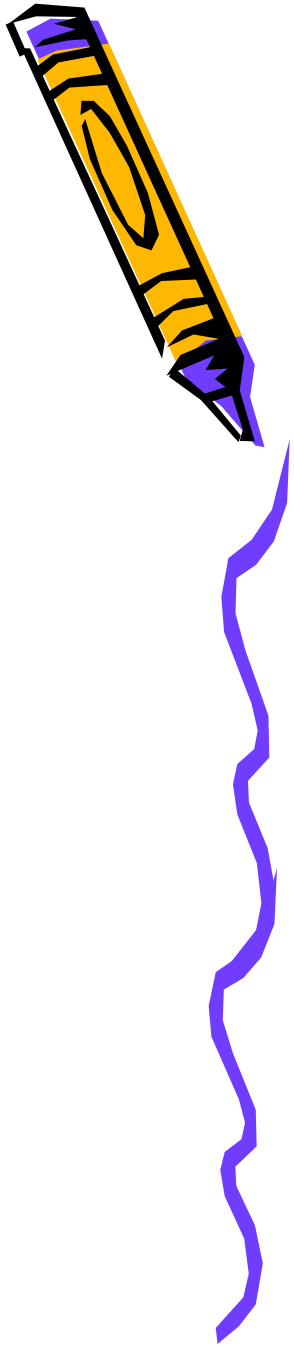
rcr OP1, OP2



Для всех команд сдвига флаги ZF, SF, PF устанавливаются в соответствии с результатом. AF – не определен. OF – не определен при сдвигах на несколько разрядов, при сдвиге на 1 разряд в зависимости от команды:

- для циклических команд, повышенной точности и sal , shl флаг OF = 1, если после сдвига старший бит изменился;
- после sar OF = 0;
- после shr OF = значению старшего бита исходного числа.

Для самостоятельного изучения команды:



BT <приемник>, <источник>

BTS <приемник>, <источник>

BTR <приемник>, <источник>

BTC <приемник>, <источник>

BSF <приемник>, <источник>

BSR <приемник>, <источник>

Комбинированный тип данных в Ассемблере. Структуры.

Структура состоит из полей-данных различного типа и длины, занимая последовательные байты памяти. Чтобы использовать переменные типа структура, необходимо вначале описать тип структуры, а затем описать переменные такого типа. Описание типа структуры:

<имя типа> struc

<описание поля>

<описание поля>

<имя типа> ends

<имя типа> - это идентификатор типа структуры, struc и ends - директивы, причем <имя типа > в директиве ends также обязательно…Для описания полей используются директивы определения DB, DW, DD,….Имя, указанное в этих директивах, является именем поля, но имена полей не локализованы внутри структуры, т.е. они должны быть уникальными в рамках всей программы, кроме того, поля не могут быть структурами – не допускаются вложенные структуры.

Например,

```
TData  struc          ;  TData – идентификатор типа
       y  DW  2000    ; y, m, d – имена полей. Значения, указанные
       m  DB  ?       ; в поле операндов директив DW и DB ,        d  DB
28     ; называются значениями полей, принятыми
TData ends            ; по умолчанию.
```

? – означает, что значения по умолчанию нет.

На основании описания типа в программу ничего не записывается и память не выделяется. Описание типа может располагаться в любом месте программы, но только до описания переменных данного типа. На основании описания переменных Ассемблером выделяется память в соответствии с описанием типа в последовательных ячейках, так что в нашем случае размещение полей можно представить так:

```
          2б   1б   1б   размеры полей в байтах
TData   | y  |  m |  d |    |
          +0   +2   +3   смещение относительно начала
                         структуры.
```

Описание переменных типа структуры осуществляется с помощью директивы вида:

имя переменной имя типа <нач. значения>

Здесь уголки не метасимволы, а реальные символы языка, внутри которых через запятую указываются начальные значения полей.

Нач-ым значением может быть: 1) ? 2) выражение 3) строка 4) пусто.

Например:

```
                        y    m   d
dt1  TData  <?, 6, 4>          ?    6   4
dt2  TData  <1999, , >      1999    ?  28
dt3  TData  < , , >         2000    ?  28
```

Идентификатор типа TData используется как директива для описания переменных также, как используются стандартные директивы DB, DW и т.д. Если начальные значения не будут умещаться в отведенное ему при описании типа поле, то будет фиксироваться ошибка. Правила использования начальных значений и значений по умолчанию:

- приоритетными являются начальные значения полей, при описании переменных, т.е. если при описании переменной для поля указан ? или какое-либо значение, то значения этих полей по умолчанию игнорируются.

Правила использования начальных значений и значений по умолчанию:

1) если в поле переменной указан знак **?**, то это поле не имеет начального значения, даже если это поле имеет значение по умолчанию (поле **y** переменной **dt1**);

2) Если в поле переменной указано выражение или строка, то значение этого выражения или сама строка становится начальным значением этого поля (поля **m** и **d** переменной **dt1** и поле **y** переменной **dt2**);

3) Если начальное значение поля переменной «**пусто**» - ничего не указано при описании переменной, то в качестве начального устанавливается значение по умолчанию – значение, указанное при описании типа, если же в этом поле при описании типа стоит знак **?**, то данное поле не имеет никакого начального значения (поля **m** переменных **dt2** и **dt3**).

Значения по умолчанию устанавливаются для тех полей, которые являются одинаковыми для нескольких переменных одного типа, например, год поступления на факультет одинаков для группы студентов. Любая переменная может изменять свое значение в процессе выполнения программы и поэтому структура может не иметь как значений по умолчанию, так и начальных значений.

Отсутствие начального значения отмечается запятой.

Если отсутствуют начальные значения нескольких последних полей, то запятые можно не ставить. Если отсутствуют значения первого поля или полей, расположенных в середине списка полей, то запятые опускать нельзя. Например:

dt4 TData <1980, ,> можно dt4 TData <1980>

dt5 TData <, , 5> нельзя заменить на dt5 TData < 5 >.

Если отсутствуют все начальные значения, опускаются все запятые, но угловые скобки сохраняются:

dt6 TData < >

При описании переменных, каждая переменная описывается отдельной переменной, но можно описать массив структур, для этого в директиве описания переменной указывается несколько операндов и (или) конструкция повторения DUP. Например:

dst TData <, 4, 1>, 25 DUP (< >)

Описан массив из 26 элементов типа TData, и первый элемент (первая структура) будет иметь начальные значения 2000, 4, 1, а все остальные 25 в качестве начальных будут иметь значения, принятые по умолчанию: 2000, ?, 28.

Имя первой структуры dst, второй – dst+4, третьей – dst+8 и т.д.

Работать с полями структуры можно также, как с полями переменной комбинированного типа в языках высокого уровня:

<имя переменной > . < имя поля>

Например, dt1.y, dt2.m, dt3.d

Ассемблер приписывает имени типа и имени переменной размер (тип), равный количеству байтов, занимаемых структурой

type TData = type dt1 = 4

И это можно использовать при программировании, например, так:

```
; выполнить побайтовую пересылку dt1 в dt2
        mov CX, type TData  ; количество повторений в CX
        mov     SI, 0           ; i = 0
  m:    mov AL, byte ptr dt1[SI]  ; побайтовая пересылка
        mov byte ptr dt2[SI], AL  ; dt1 в dt2
            inc SI          ; i = i+1
            loop m          ; использование byte ptr обязательно….
```

Точка, указанная при обращении к полю, это оператор Ассемблера, который вычисляет адрес по формуле:

<адресное выражение> + <смещение поля в структуре>

Тип полученного адреса совпадает с типом поля, т.е.

type (dt1.m) = type m = byte

Адресное выражение может быть любой сложности, например:

1) mov AX, (dts+8).y
2) mov SI, 8
   inc (dts[SI]).m ; Аисп = (dts + [SI]). m = (dts + 8).m
3) lea BX, dt1
   mov [BX].d, 10 ; Аисп = [BX] + d = dt1.d

Замечания:

1) type (dts[SI]).m = type (dts[SI].m) = 1, но
   type dts[SI].m = type dts = 4
2) Если при описании типа структуры в директиве, описывающей некоторое поле, содержится несколько операндов или конструкция повторения, то при описании переменной этого типа данное поле не может иметь начального значения и не может быть определено знаком ?, это поле должно быть пустым.

Одно исключение: если поле описано как строка, то оно может иметь начальным значением строку той же длины или меньшей, в последнем случае строка дополняется справа пробелами.

Например: student struc

```
f  DB  10 DUP (?)      ; фамилия
i  DB  “ ******* “     ; имя
gr  DW  ?              ; группа
oz  DB   5, 5, 5       ; оценки
student ends
```

Описание переменных:

st1 student <“Petrov”, >; нельзя, т.к. поле f не строка

st2 student < , “Petr”, 112, > ; можно, f – не имеет начального
; значения

st3 student < , “Aleksandra” >
; нельзя, в i 10 символов, а допустимо не больше 7.

Примеры программ с использованием данных типа структура.

```
; prim1.asm – прямое обращение к полям структуры
        .model tiny
        .code
        org 100h  ; обход 256 байтного префикса пр-го сегмента – PSP…
Start:    mov AH, 9
          mov DX, offset message
          int 21h
;
          lea DX, st1.s
          int 21h
          lea DX, st1.f
          int 21h
          lea DX, st1.i
          int 21h
          ret
```

```
message DB “ hello”, 0dh, 0ah, ”$”
    tst struc; описание типа структуры
        s  DB  “student”,”$”
        f   DB  “Ivanov “,”$”
        i   DB   “Ivan    “,”$”
    tst  ends
    st1  tst  <  > ; описание переменной типа tst
end start
```

org 100h - все сегментные регистры вначале выполнения программы содержат адрес блока PSP, который резервируется непосредственно перед EXE и COM файлами. Смещением для 1-ой команды программы является адрес 100h. Переход на первую выполняемую команду и происходит с помощью директивы ORG 100h.

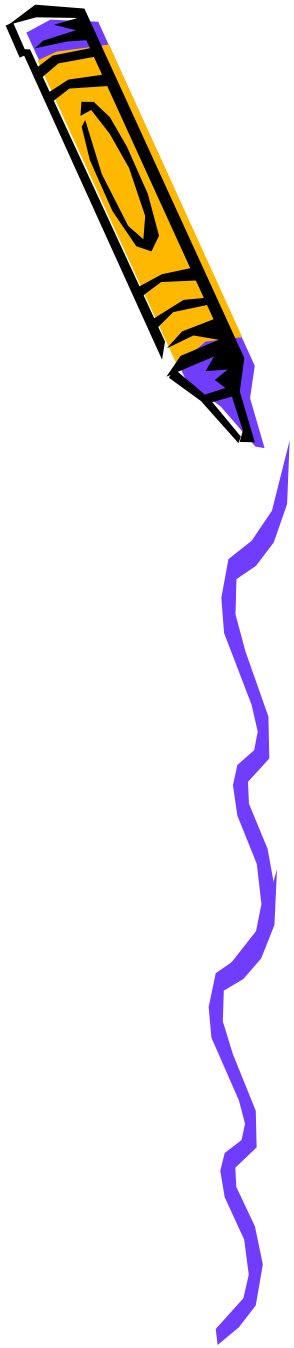
Prim2.asm – обращение к полям структуры в цикле

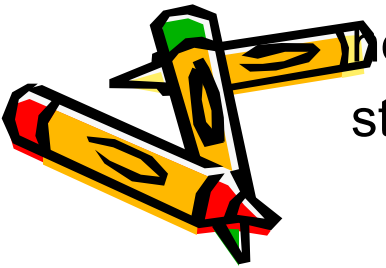
```
        .model tiny
        .code
        org 100h  ; обход 256 байтного префикса пр-го сегмента – PSP…
Start:  mov AH, 9
        mov DX, offset message
        int 21h
        mov  SI, 0
        mov CX, 3
m1: lea  DX, st1[SI]
        int 21h
        add SI, 9
        loop m1
        ret
message  DB  “hello”,0dh,0ah,”$”
        tst struc;  описание типа структуры
            s  DB  “student ”,”$”
            f   DB  “Ivanov  “,”$”
            i   DB   “Ivan     “,”$”
                tst  ends
        st1   tst  <  >
                end start
```

Prim3.asm – обращение к полям структур: цикл в цикле для работы с 2-мя структурами

```
        .model tiny
        .code
        org 100h  ; обход блока PSP
Start:    mov AH, 9
          mov DX, offset message
          int 21h
        lea BX, st1   ; адрес первой записи в BX
        mov CX, 2          ; количество повторений внешнего цикла
m2:   push CX
        mov SI, 0
        mov CX, 3          ; количество повторений внутреннего цикла
m1:   push CX
          lea DX, [BX] [SI]  ; адресация по базе с индексированием
                int 21h
                add SI, 9  ; переход к следующему полю
                pop  CX
                loop m1
```

```
  add BX, type  tst  ; переход к следующей записи
; BX + количество байтов, занимаемой структурой типа tst
    pop CX
    loop m2
    ret
message  DB  “hello”,0dh,0ah,”$”
    tst struc; описание типа структуры
        s  DB  ?
        f   DB  ?
        i   DB   ?
    tst  ends
        st1   tst  < “student $”,”Inanov  $”,”Ivan,   $” >
        st2   tst  < “student $”,”Petrov  $”,”Petr,    $” >
            end start
            Результат работы программы:
            hello
            student Ivanov  Ivan,   student  Petrov  Petr
```





Записи в Ассемблере

Запись – это упакованные данные, которые занимают не отдельные, полные ячейки памяти (байты или слова), а части ячеек.

Запись в Ассемблере занимает байт или слово (другие размеры ячеек для записи не допускаются), а поля записи - это группы последовательных битов.

Поля должны быть прижаты друг к другу, между ними не должно быть пробелов.

Размер поля в битах может быть любым, но в сумме размер всех полей не должен быть больше 16.

Сумма размеров всех полей называется размером записи. …..

Если размер записи меньше 8 или 16, то поля прижимаются к правой границе ячейки, оставшиеся левые биты равны нулю, но к записи не относятся и не рассматриваются.

Поля имеют имена, но обращаться к ним по именам нельзя, так как наименьший адресуемый элемент памяти это байт.

Для работы с записью необходимо описать вначале тип записи, а затем описать переменные этого типа.

Описание типа может располагаться в любом месте программы, но до описания переменных этого типа.

Директива описания типа записи имеет вид:

<имя типа записи > record <поле> {, <поле>}

<поле> :: = <имя поля> : <размер> [= <выражение>]

Здесь <размер> и <выражение> - это константные выражения.

<размер> определяет размер поля в битах, <выражение> определяет значение поля по умолчанию. Знак ? не допускается.

Например: графическое представление

TRec record A : 3, B : 3 = 7 7 6 A B имена полей

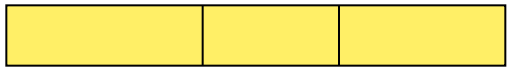
0 7

3 3 размер в битах

TData record Y : 7, M : 4, D : 5 Y M D

0 0 0

7 4 5



Год, записанный двумя последними цифрами $2^6 < Y_{max} = 99 < 2^7$

Имена полей, также как и в структурах, должны быть уникальными в рамках всей программы, в описании они перечисляются слева направо. <выражение> может отсутствовать, если оно есть, то его значение должно умещаться в отведенный ему размер в битах. Если для некоторого поля выражение отсутствует, то его значение по умолчанию равно нулю, не определенных полей не может быть.

Определенное директивой record имя типа (Trec, TData) используется далее как директива для описания переменных – записей такого типа.

имя записи имя типа записи <начальные значения>,

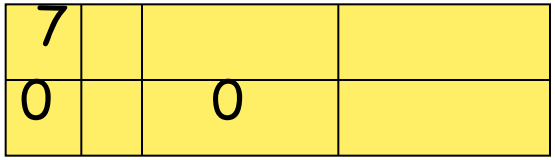
угловые скобки здесь не метасимволы, а символы языка, внутри которых через запятую указываются начальные значения полей.

Начальными значениями могут быть:

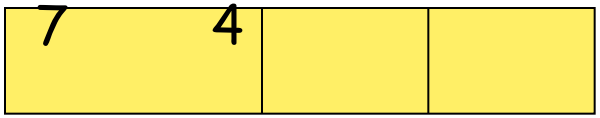
1) константное выражение, 2) знак ?, 3) пусто

В отличие от структуры, знак ? определяет нулевое начальное значение, а «пусто», как и в структуре, определяет начальное значение равным значению по умолчанию. Например:

```
Rec1  TRec  < 3, > ;      7 6 A   B   0
                      0  0    3
Rec2  TRec  < , ? >         0  0
```




```
Dat1  TData  < 80, 7, 4 > ;      15   Y        M        D  0
                           80
```



также , как и для структур:

Dat1 TData < 00, , > == Dat1 TData < 00 >
Dat2 TData < , , > == Dat2 TData < >

Одной директивой можно описать массив записей, используя несколько параметров в поле операндов или конструкцию повторения, например,

MDat TData 100 Dup (< >)

Описали 100 записей с начальными значениями, равными принятыми по умолчанию.

Со всей записью в целом можно работать как обычно с байтами или со словами, т.е. можно реализовать присваивание Rec1 = Rec2 :

mov AL, Rec2
mov Rec1, AL

Для работы с отдельными полями записи существуют специальные операторы **width и mask**.

width <имя поля записи>
width <имя записи или имя типа записи>

Значением оператора **width** является размер в битах поля или всей записи в зависимости от операнда.

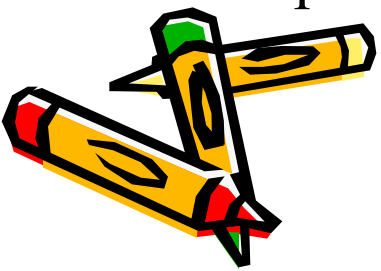
оператор mask имеет вид:

Mask <имя поля записи>
Mask <имя записи или имя типа записи>

Значением этого оператора является «маска» - это байт или слово, в зависимости от размера записи, содержащее единицы в тех разрядах, которые принадлежат полю или всей записи, указанных в качестве операнда, и нули в остальных, не используемых разрядах. Например:

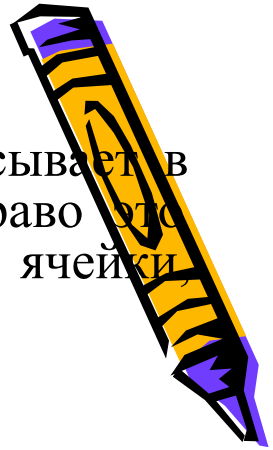
mask A = 00111000b
mask B = 00000111b
mask Y = 1111111000000000b
mask Rec1 = mask TRec = 00111111b

Этот оператор используется для выделения полей записи.



Пример. Выявить всех родившихся 1-го числа, для этого придется выделять поле D и сравнивать его значение с 1-ей.

```
m1:     ----------------------
        mov AX,  Dat1
        and  AX,  mask D
        cmp  AX,  1
        je   yes
no: ---------------------
        ---------------------
        jmp      m1
yes: ----------------------
```



При работе с записями, ассемблер имени любого поля приписывает в качестве значения число, на которое нужно сдвинуть вправо это поле, чтобы оно оказалось прижатым к правой границе ячейки, занимаемой записью. Так значением поля D для записи типа

TData является ноль,

для поля M – 5, для поля Y – 9.

Значения имен полей используются в командах сдвига, например, определить родившихся в апреле можно так:

```
m1:  ---------------------------
     mov AX, Dat ; AX = Y M D
     and  AX, mask M  ;  AX = 0 M 0
     mov CL,  M  ;  CL = 5
     shr  AX,  CL  ;  AX = 0 0 M
     cmp  AX,  4   ;  M = 4 ?
     je    yes
no: ------------------
          jmp m1
          ----------------------
yes:  ------------------------
```