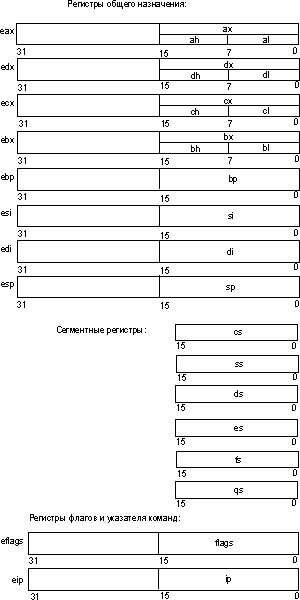
***ПРОГРАММНАЯ МОДЕЛЬ ПРОЦЕССОРОВ СЕМЕЙСТВА X86***

Пользовательские регистры

Как следует из названия, пользовательскими регистры называются потому, что программист может использовать их при написании своих программ. К этим регистрам относятся ([рис. 1](#Рис_1)):

* восемь 32-битных регистров, которые могут использоваться программистами для хранения данных и адресов (их еще называют [регистрами общего назначения](#регистры общего назначения) (РОН)):

1. **eax/ax/ah/al**;
2. **ebx/bx/bh/bl**;
3. **edx/dx/dh/dl**;
4. **ecx/cx/ch/cl**;
5. **ebp/bp**;
6. **esi/si**;
7. **edi/di**;
8. **esp/sp**.
9. [шесть регистров сегментов:](#Сегментные регистры) **cs, ds, ss, es, fs, gs**;
10. [регистры состояния и управления](#Регистры состояния и управления):
11. регистр флагов **eflags/flags**;
12. регистр указателя команды **eip/ip**.



**Рис. 1.** Пользовательские регистры микропроцессоров i486 и Pentium

Почему многие из этих регистров приведены с наклонной разделительной чертой?

Нет, это не разные регистры — это части одного большого 32-разрядного регистра. Их можно использовать в программе как отдельные объекты.

Так сделано для обеспечения работоспособности программ, написанных для младших 16-разрядных моделей микропроцессоров фирмы Intel, начиная с i8086.

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

Регистры общего назначения

Все регистры этой группы позволяют обращаться к своим “младшим” частям.

Для самостоятельной адресации можно использовать только младшие 16 и 8-битные части этих регистров. Старшие 16 бит этих регистров как самостоятельные объекты недоступны. Это сделано, как мы отметили выше, для совместимости с младшими 16-разрядными моделями микропроцессоров фирмы Intel.

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

1. **eax/ax/ah/al** (Accumulator register) — аккумулятор.

Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;

1. **ebx/bx/bh/bl** (Base register) — базовый регистр.

Применяется для хранения базового адреса некоторого объекта в памяти;

1. **ecx/cx/ch/cl** (Count register) — регистр-счетчик.

Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды.

К примеру, команда организации цикла [**loop**](file:///C:\www\doc2html\work\bestreferat-325568-14005728137941\input\Command.htm#loop) кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра ecx/cx;

1. **edx/dx/dh/dl** (Data register) — регистр данных.

Так же, как и регистр eax/ax/ah/al, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.

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

1. **esi/si** (Source Index register) — индекс источника.

Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;

1. **edi/di** (Destination Index register) — индекс приемника (получателя).

Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

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

1. **esp/sp** (Stack Pointer register) — регистр указателя стека.

Содержит указатель вершины стека в текущем сегменте стека.

1. **ebp/bp** (Base Pointer register) — регистр указателя базы кадра стека.

Предназначен для организации произвольного доступа к данным внутри стека.

Не спешите пугаться столь жесткого функционального назначения регистров АЛУ. На самом деле, большинство из них могут использоваться при программировании для хранения операндов практически в любых сочетаниях. Но, как мы отметили выше, некоторые команды используют фиксированные регистры для выполнения своих действий. Это нужно обязательно учитывать.

Использование жесткого закрепления регистров для некоторых команд позволяет более компактно кодировать их машинное представление. Знание этих особенностей позволит вам при необходимости хотя бы на несколько байт сэкономить память, занимаемую кодом программы.

Сегментные регистры cs, ss, ds, es, gs, fs.

Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых сегментами. Соответственно, такая организация памяти называется **сегментной**.

Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены сегментные регистры. Фактически, с небольшой поправкой, как мы увидим далее, в этих регистрах содержатся адреса памяти с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:

1. **Сегмент кода**. Содержит команды программы.

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

1. **Сегмент данных**. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр **ds** (data segment register) — сегментный регистр данных, который хранит адрес сегмента данных текущей программы.
2. **Сегмент стека**. Этот сегмент представляет собой область памяти, называемую стеком. Работу со стеком микропроцессор организует по следующему принципу: последний записанный в эту область элемент выбирается первым. Для доступа к этому сегменту служит регистр **ss** (stack segment register) — сегментный регистр стека, содержащий адрес сегмента стека.
3. **Дополнительный сегмент данных**.

Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре ds.

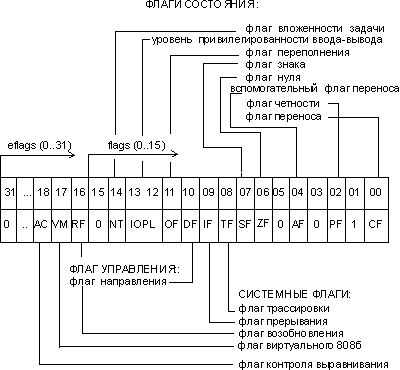
Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их адреса требуется указывать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

Адреса дополнительных сегментов данных должны содержаться в регистрах **es, gs, fs** (extension data segment registers).

Регистры состояния и управления *eflags и ip*

Они постоянно содержат информацию о состоянии, как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора.

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



**Рис. 2. Содержимое регистра eflags**

Исходя из особенностей использования, флаги регистра eflags/flags можно разделить на три группы:

1. **8 флагов состояния. Эти флаги могут изменяться после выполнения машинных команд.** Флаги состояния регистра eflags отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм.
2. **1 флаг управления - df (Directory Flag).** Значение флага df определяет направление поэлементной обработки цепочек данных: от начала строки к концу (df = 0) либо наоборот, от конца строки к ее началу (df = 1).
3. **5 системных флагов, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086.** Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.

**eip/ip** (Instraction Pointer register) — регистр-указатель команд.

Регистр eip/ip имеет разрядность 32/16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра eip/ip.

Типы данных. Переменные

В программе на ассемблере переменными являются регистры или ячейки памяти, в которых хранятся данные. Существует несколько типов данных-переменных:

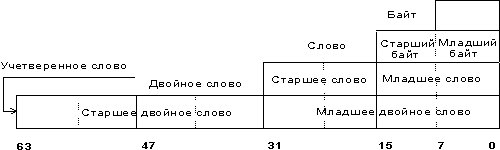
1. Непосредственные данные, представляющие собой числовые или символьные значения, являющиеся частью команды. 20d, 0a2h, 10111b
2. Данные [[простого типа](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideText%22%20l%20)](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideText%22%20l%20), описываемые с помощью ограниченного набора директив резервирования памяти, позволяющих выполнить самые элементарные операции по размещению и инициализации числовой и символьной информации.

Эти два типа данных являются элементарными, или базовыми; работа с ними поддерживается на уровне системы команд микропроцессора. Используя данные этих типов, можно формализовать и запрограммировать практически любую задачу. Но насколько это будет удобно — вот вопрос.

1. Данные [сложного типа](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideText%22%20l%20)**,** (массивы, структуры, записи и пр.) которые были введены в язык ассемблера с целью облегчения разработки программ. Сложные типы данных строятся на основе базовых типов, которые являются как бы кирпичиками для их построения. Введение сложных типов данных позволяет несколько сгладить различия между языками высокого уровня и ассемблером

***Физическая интерпретация*** данных простого типа основывается на размерности данных:

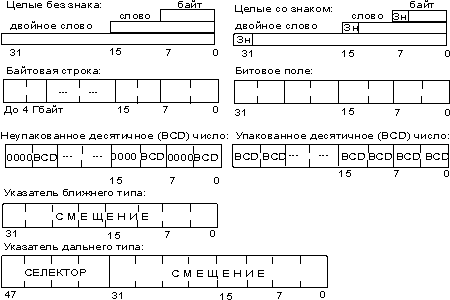
* байт — восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом;
* слово — последовательность из двух байт, имеющих последовательные адреса. Размер слова — 16 бит; биты в слове нумеруются от 0 до 15. Байт, содержащий нулевой бит, называется младшим байтом, а байт, содержащий 15-й бит - старшим байтом. Микропроцессоры Intel имеют важную особенность — младший байт всегда хранится по меньшему адресу. Адресом слова считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова.
* двойное слово — последовательность из четырех байт (32 бита), расположенных по последовательным адресам.
* учетверенное слово — последовательность из восьми байт (64 бита), расположенных по последовательным адресам.



**Рис. 3.** Основные типы данных микропроцессора

**Логическая интерпретация** этих типов:

* Целый тип со знаком — двоичное значение со знаком, размером 8, 16 или 32 бита. Знак в этом двоичном числе содержится в 7, 15 или 31-м бите соответственно. Ноль в этих битах в операндах соответствует положительному числу, а единица — отрицательному. Отрицательные числа представляются в дополнительном коде. Числовые диапазоны для этого типа данных следующие:
  + 8-разрядное целое — от –128 до +127;
  + 16-разрядное целое — от –32 768 до +32 767;
  + 32-разрядное целое — от –231 до +231–1.
* Целый тип без знака — двоичное значение без знака, размером 8, 16 или 32 бита. Числовой диапазон для этого типа следующий:
  + байт — от 0 до 255;
  + слово — от 0 до 65 535;
  + двойное слово — от 0 до 232–1.
* Указатель на память двух типов:
  + ближнего типа — 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента. Эти указатели могут также использоваться в сплошной (плоской) модели памяти, где сегментные составляющие одинаковы;
  + дальнего типа — 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части — селектора, и 32-разрядного смещения.
* Цепочка — представляющая собой некоторый непрерывный набор байтов, слов или двойных слов максимальной длины до 4 Гбайт.
* Битовое поле представляет собой непрерывную последовательность бит, в которой каждый бит является независимым и может рассматриваться как отдельная переменная. Битовое поле может начинаться с любого бита любого байта и содержать до 32 бит.
* Неупакованный двоично-десятичный тип — байтовое представление десятичной цифры от 0 до 9. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте. Значение цифры определяется младшим полубайтом.
* Упакованный двоично-десятичный тип представляет собой упакованное представление двух десятичных цифр от 0 до 9 в одном байте. Каждая цифра хранится в своем полубайте. Цифра в старшем полубайте (биты 4–7) является старшей.



**Рис. 4.** Основные логические типы данных микропроцессора

**Язык микроопераций. Ассемблер.**

Структура программы на ассемблере:

**Model small** ;модель программы, или же количество памяти на сегмент

**.data** ;сегмент данных

;описание переменных

**.stack 100h** ;сегмент стека

**.code ;**сегмент данных

;процедуры, макрокоманды

**main:**

;основная программа

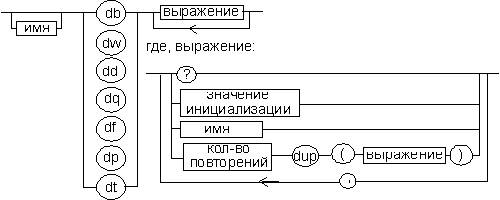
**end main**

**Директивы резервирования памяти**

Для описания простых типов данных в программе используются специальные директивы резервирования и инициализации данных, которые, по сути, являются указаниями транслятору на выделение определенного объема памяти. Если проводить аналогию с языками высокого уровня, то директивы резервирования и инициализации данных являются определениями переменных.

**Машинного эквивалента этим директивам нет; просто транслятор, обрабатывая каждую такую директиву, выделяет необходимое количество байт памяти и при необходимости инициализирует эту область некоторым значением.**

Директивы резервирования и инициализации данных простых типов имеют формат:



**Рис. 5.** Директивы описания данных простых типов

На рис. 5 использованы следующие обозначения:

* **?** показывает, что содержимое поля не определено, то есть при задании директивы с таким значением выражения содержимое выделенного участка физической памяти изменяться не будет. Фактически, создается неинициализированная переменная;
* **значение инициализации** — значение элемента данных, которое будет занесено в память после загрузки программы. Фактически, создается инициализированная переменная, в качестве которой могут выступать константы, строки символов, константные и адресные выражения в зависимости от типа данных. Подробная информация приведена в приложении 1;
* **выражение** — итеративная конструкция с синтаксисом, описанным на рис. 5.17. Эта конструкция позволяет повторить последовательное занесение в физическую память выражения в скобках n раз.
* **имя** — некоторое символическое имя метки или ячейки памяти в сегменте данных, используемое в программе.
* **db** **— резервирование памяти для данных размером 1 байт.** Директивой **db** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для чисел со знаком –128...+127;
    - для чисел без знака 0...255;
  + символьную строку из одного или более символов. Строка заключается в кавычки. В этом случае определяется столько байт, сколько символов в строке.
* **dw — резервирование памяти для данных размером 2 байта**. Директивой **dw** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для чисел со знаком –32 768...32 767;
    - для чисел без знака 0...65 535;
  + выражение, занимающее 16 или менее бит, в качестве которого может выступать смещение в 16-битовом сегменте или адрес сегмента;
  + 1- или 2-байтовую строку, заключенная в кавычки.
* **dd** — **резервирование памяти для данных размером 4 байта.** Директивой **dd** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - для i386 и выше:
      * для чисел со знаком –2 147 483 648...+2 147 483 647;
      * для чисел без знака 0...4 294 967 295;
  + относительное или адресное выражение, состоящее из 16-битового адреса сегмента и 16-битового смещения;
  + строку длиной до 4 символов, заключенную в кавычки.
* **df — резервирование памяти для данных размером 6 байт;**
* **dp** — **резервирование памяти для данных размером 6 байт.** Директивами **df** и **dp** можно задавать следующие значения:
  + выражение или константу, принимающую значение из диапазона:
    - * для чисел со знаком –2 147 483 648...+2 147 483 647;
      * для чисел без знака 0...4 294 967 295;
  + относительное или адресное выражение, состоящее из 32 или менее бит (для i80386) или 16 или менее бит (для младших моделей микропроцессоров Intel);
  + адресное выражение, состоящее из 16-битового сегмента и 32-битового смещения;
  + строку длиной до 6 байт, заключенную в кавычки.
* **dq — резервирование памяти для данных размером 8 байт.** Директивой **dq** можно задавать следующие значения:
  + относительное или адресное выражение, состоящее из 32 или менее бит
  + константу со знаком из диапазона –263...263–1;
  + константу без знака из диапазона 0...264–1;
  + строку длиной до 8 байт, заключенную в кавычки.
* **dt — резервирование памяти для данных размером 10 байт.** Директивой **dt** можно задавать следующие значения:
  + относительное или адресное выражение, состоящее из 32 или менее бит
  + адресное выражение, состоящее из 16-битового сегмента и 32-битового смещения;
  + константу со знаком из диапазона –279...279-1;
  + константу без знака из диапазона 0...280-1;
  + строку длиной до 10 байт, заключенную в кавычки;
  + упакованную десятичную константу в диапазоне 0...99 999 999 999 999 999 999.

Очень важно уяснить себе порядок размещения данных в памяти. Он напрямую связан с логикой работы микропроцессора с данными. Микропроцессоры Intel требуют следования данных в памяти по принципу: **младший байт по младшему адресу**.

Для иллюстрации данного принципа рассмотрим листинг 1, в котором определим сегмент данных. В этом сегменте данных приведено несколько директив описания простых типов данных.

Листинг 1. Пример использования директив резервирования и инициализации данных. Программа вводит строку с клавиатуры.

**model small**

**.stack 100h**

**.data**

message db 'Массив байт, содержащих символьные переменные',10,13 '$'

po db 1, 3, 4, 5, 0fh, 0bh, 32, 01011b

perem\_1 db 0ffh

perem\_2 dw 3a7fh

perem\_3 dd 0f54d567ah

k1 db 10

k2 db ?

mas db 10 dup ('?')

adr dw k1

adr\_full dd perem\_3

**.code**

**start:**

**mov ax,@data**

**mov ds,ax**

mov ah,0ah

mov dx,offset message ; mov dx, adr

int 21h

**mov ax,4c00h**

**int 21h**

**end start**

## Система команд

## Формат предложения ассемблера

[имя метки:] КОП [операнд1] [,операнд2] [;комментарии]

***Команды пересылки данных***

|  |  |  |
| --- | --- | --- |
| ***mov***<операнд назначения>,<операнд-источник> | | |
| можно | Нельзя | Должно быть |
| mov ах, вх; ах:=вх  mov ах,0а2h; ах:= 0а2h  mov per1,ax | mov ax,bh  mov per1, per2  mov ds,per1  mov cs,ds  mov cs,ax; пара cs:ip содержит адрес следующей команды | mov ah, bh  mov al, per2  mov per1,al  mov ax, per1  mov ds,ax  mov ax,ds либо push ds  mov cs,ax pop cs |

***xchg*** <операнд1>,<операнд2> ; двунаправленный обмен данными а:=в; в:=с; с:=а

xchg dl,dh; меняет местами данные

***Команды ввода-вывода в порт***

[**in**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#in) аккумулятор,номер\_порта — ввод в аккумулятор из порта

[**out**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#out)порт,аккумулятор — вывод содержимого аккумулятора в порт

## Команды работы с адресами и указателями памяти

[**lea**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#lea)назначение,источник — загрузка эффективного адреса источника в регистр-назначение;

**lea** dx, x ; аналогично команде mov dx,offset x

[**lds**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#lds) назначение,источник — загрузка эффективного адреса источника в регистр назначения и загрузка указателя (адрес сегмента где содержится источник) в регистр сегмента данных ds;

[**les**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#lds)назначение,источник —-//-регистр дополнительного сегмента данных es;

[**lgs**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#lds) назначение,источник — -//- регистр дополнительного сегмента данных gs;

[**lfs**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#lds) назначение,источник — -//- регистр дополнительного сегмента данных fs;

[**lss**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#lds) назначение,источник — -//- регистр сегмента стека ss.

les dx,per1 ;полный указатель на per1 в пару es:dx

**Команды работы со стеком**

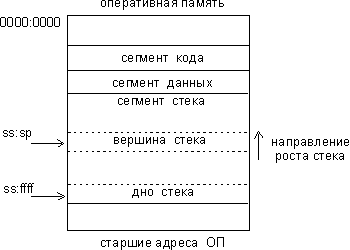
Для работы со стеком предназначены три регистра:

*ss* — сегментный регистр стека;

*sp/esp* — регистр указателя стека;

*bp/ebp* — регистр указателя базы кадра стека.

[**push**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#push) источник — запись значения источник в вершину стека.



*Алгоритм работы:*

* уменьшить значение указателя стека esp/sp на 4/2 (в зависимости от значения атрибута размера адреса — use16 или use32);
* записать источник в вершину стека (адресуемую парой ss:esp/sp).

Размер записываемых значений — слово или двойное слово. Также в стек можно записывать непосредственные значения. В стек можно класть значение сегментного регистра cs. Другой интересный момент связан с регистром sp. Команда push esp/sp записывает в стек значение esp/sp по состоянию до выдачи этой команды

Команда push используется совместно с командой pop для записи значений в стек и извлечения их из стека

[**pop**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#pop) назначение — запись значения из вершины стека по месту, указанному операндом назначение. Значение при этом “снимается” с вершины стека.

cx

bx

ax

…

ss:sp

Push ax

Push bx

push cx

…

pop cx

ss:bp

pop bx

pop ax

push ax

pop bx ; аналогично команде mov bx,ax

[**push**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#push)**a** - размещение в стеке регистров общего назначения в следующей последовательности: ax, cx, dx, bx, sp, bp, si, di

[**push**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#push)**ad** - размещение в стеке регистров общего назначения в следующей последовательности: eax, ecx, edx, ebx, esp, ebp, esi, edi

***pushf*** - размещение в вершине стека (ss:sp) содержимого регистра флагов flags

***pushfd*** - размещение в стеке содержимого регистра флагов eflags.

***popa*** - извлечение из стека регистров общего назначения di, si, bp, sp, bx, dx, cx, ax

***popad*** - извлечение из стека регистров общего назначения edi, esi, ebp, esp, ebx, edx, ecx, eax

***popf*** - извлечение из стека слова и восстановление его в регистр флагов flags

***popfd*** - извлечение из стека двойного слова и восстановление его в регистр флагов eflags

Организация вычислений

## Логические команды

Любая логическая команда меняет значение следующих флагов of, sf,zf,pf,cf (переполнение, знак, нуля, паритет, перенос)

[**and**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#and) операнд\_1,операнд\_2 — операция логического умножения (И - конъюнкция).

оп1:=оп1 ٧ оп2

and ah, 0a1h; ah:=ah٧0ah

and bx, cx; bx:=bx٧cx

and dx, x1; dx:=dx٧x1

[**or**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#or)операнд\_1,операнд\_2 — операция логического сложения (ИЛИ - дизъюнкцию)

or al, x1; оп1:=оп1 & оп2

or eax,edx

or dx, x1

[**xor**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#xor) операнд\_1,операнд\_2 — операция логического исключающего сложения (исключающего ИЛИ ИЛИ-НЕ)

[**test**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#test)операнд\_1,операнд\_2 — операция “проверить” (способом логического умножения).

Команда выполняет поразрядно логическую операцию И над битами операндов операнд\_1 и операнд\_2. Состояние операндов остается прежним, изменяются только флаги zf, sf, и pf, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

[**not**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#not) операнд — операция логического отрицания. Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

Пример программы логического сложения двух однобайтных чисел.

**model small**

**.stack 100h**

**.data**

x1 db 0c2h

x2 db 022h

y db ?

**.code**

**start:**

**mov ax,@data**

**mov ds,ax**

mov al, x1

or al, x2

mov y, al

**mov ax,4c00h**

**int 21h**

**end start**

## Арифметические операции над целыми двоичными числами

###### Сложение двоичных чисел без знака

***inc*** операнд - операция инкремента, то есть увеличения значения операнда на 1;

inc ax; ax:=ax+1

inc x1

***add*** оп1,оп2 - команда сложения с принципом действия: оп1 = оп1 + оп2 (addition)

add al, bl

add ax, 0fe2h

add ebx, x1+2

add x1, 0fh

add x2, ax

***adc*** оп1,оп2 - команда сложения с учетом флага переноса cf. оп1 = оп1 + оп2 + знач\_cf

###### Вычитание двоичных чисел без знака

***dec*** операнд — операция декремента, то есть уменьшения значения операнда на 1;

dec cx ;cx:=cx-1

dec x

***sub*** операнд\_1,операнд\_2 — команда вычитания; ее принцип действия:

операнд\_1 = операнд\_1 – операнд\_2

sub al, bl; al:=al-bl

sub ax, x1

sub x2, dx

sub eax, 0f35h

sub x2, 22h

***sbb*** операнд\_1,операнд\_2 — команда вычитания с учетом заема (флага cf ):

операнд\_1 = операнд\_1 – операнд\_2 – значение\_cf

Пример программы сложения двух однобайтных чисел.

**model small**

**.stack 100h**

**.data**

x1 db 0c2h

x2 db 022h

y db ?

**.code**

**start:**

**mov ax,@data**

**mov ds,ax**

mov al, x1

add al, x2

mov y, al

**mov ax,4c00h**

**int 21h**

**end start**

## Умножение двоичных чисел

***mul***множитель\_1 - операция умножения двух целых чисел без учета знака

*Алгоритм работы:*

Команда выполняет умножение двух операндов без учета знаков. Алгоритм зависит от формата операнда команды и требует явного указания местоположения только одного сомножителя, который может быть расположен в памяти или в регистре. Местоположение второго сомножителя фиксировано и зависит от размера первого сомножителя

mul dl; ax:=al\*dl, dl- множитель\_1 , al- множитель\_2

mul x1; dx:ax=ax\*0ad91h, x1 word- множитель\_1 , ax- множитель\_2

mul ecx; edx:eax=eax\*ecx, ecx- множитель\_1 , eax- множитель\_2

***imul*** множитель\_1 - операция умножения двух целочисленных двоичных значений со знаком

## Деление двоичных чисел

***div*** делитель - выполнение операции деления двух двоичных беззнаковых значений

*Алгоритм работы:*

Для команды необходимо задание двух операндов — делимого и делителя. Делимое задается неявно и размер его зависит от размера делителя, который указывается в команде

div dl ;ah:al=ax/dl, ax –делимое, dl- делитель , ah-частное, al -остаток

div x1 ;ax:dx=dx:ax/0ad91h, dx:ax –делимое, x1 word- делитель , ax-частное,

;dx -остаток

div ecx ;eax:edx=edx:eax/ecx, edx:eax –делимое, ecx- делитель , eax-частное,

;edx -остаток

***idiv*** делитель - операция деления двух двоичных значений со знаком

Пример программы умножения двух однобайтных чисел.

**model small**

**.stack 100h**

**.data**

x1 db 78

yl db ?

yh db ?

**.code**

**start:**

**mov ax,@data**

**mov ds,ax**

xor ax, ax

mov al, 25

mul x1

jnc m1 ;если нет переполнения

mov yh,ah

m1:

mov yl, al

**mov ax,4c00h**

**int 21h**

end start

Пример. Вычислите следующее выражение у=(х2-х3)/х1, х1,х2,х3 - однобайтные числа

**model small**

**.stack 100h**

**.data**

s1 db 'Введите х1',10,13,'$'

s2 db 'Введите х2',10,13,'$'

s3 db 'Введите х3',10,13,'$'

x1 db ?

x2 db ?

yc db ? ;частное

yo db ? ;остаток

**.code**

**start:**

**mov ax,@data**

**mov ds,ax**

mov ah,09h

mov dx, offset s1

int 21h ;вывод строки

mov ah,01h вводим х1

int 21h ;вводим число

sub al,30h ;al:=x1

mov x1,al

mov ah,09h

mov dx, offset s2

int 21h

mov ah,01h вводим х2

int 21h

sub al,30h ;al:=x2

mov x2,al

mov ah,09h

mov dx, offset s3

int 21h

mov ah,01h вводим х3

int 21h

sub al,30h ;al:=x3

mov bl,x2 ;bl:=x2

sub bl,al ;bl:=x2-x3

xchg al,bl ;al:=bl, al:=x2-x3

xor ah,ah ;ax:=x2-x3 вычисляем у

mov dl,x1 ;dl:=x1

div dl ;ax/dl, ax/x1

mov yc,ah

mov yo,al

; можно вывести результат на экран

**mov ax,4c00h**

**int 21h**

**end start**

### ***Команды сдвига***

Все команды сдвига обеспечивают манипуляции над отдельными битами операндов, они перемещают биты в поле операнда влево или вправо в зависимости от кода операции.

Все команды сдвига устанавливают флаг переноса cf.

[**shl**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#shl) **операнд,счетчик\_сдвигов** (Shift Logical Left) - логический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик\_сдвигов. Справа (в позицию младшего бита) вписываются нули;

[**shr**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#shr) **операнд,счетчик\_сдвигов** — логический сдвиг вправо.

shl al, 2

Было al=00001101

Стало al=00110100

shr ax, 8

Было ax=4faf, стало ax=004f

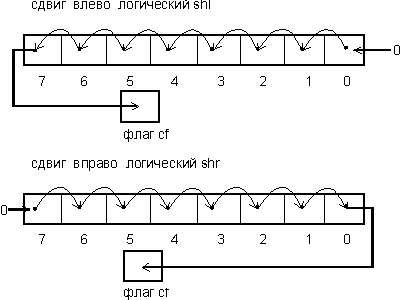
mov cl,05h

shl bh, cl

bh=00010110

bh=11000000

shr x1,3



*Алгоритм работы команд:*

* очередной “выдвигаемый” бит устанавливает флаг cf;
* бит, вводимый в операнд с другого конца, имеет значение 0;
* при сдвиге очередного бита он переходит во флаг cf, при этом значение предыдущего сдвинутого бита ***теряется!***

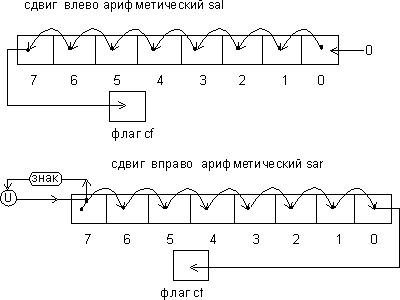
[**sal**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#sal) **операнд,счетчик\_сдвигов** (Shift Arithmetic Left)

[**sar**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#sar) **операнд,счетчик\_сдвигов**

арифметический сдвиг влево/вправо. Содержимое операнда сдвигается влево/ вправо на количество битов, определяемое значением *счетчик\_сдвигов*. Справа/ Слева в операнд вписываются нули.

Команда sal **не сохраняет знака**, но *устанавливает флаг cf в случае смены знака* очередным выдвигаемым битом. В остальном команда sal полностью аналогична команде shl;

Команда sar **сохраняет знак**, восстанавливая его после сдвига каждого очередного бита.



#### Команды циклического сдвига

[**rol**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#rol) **операнд,счетчик\_сдвигов** (Rotate Left) — циклический сдвиг влево.

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

[**ror**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#ror) **операнд,счетчик\_сдвигов** (Rotate Right) — циклический сдвиг вправо.

rol al, 3

Было al=10101100

Стало al=01100101

ror ax, 8

Было ax=4faf, стало ax=af4f

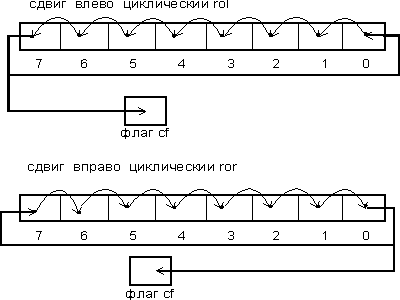
mov cl,05h

rol bh, cl

bh=00010110

bh=11000010

ror x1,2



Как видно из рис., команды простого циклического сдвига в процессе своей работы осуществляют одно полезное действие, а именно: циклически сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становиться значением флага cf.

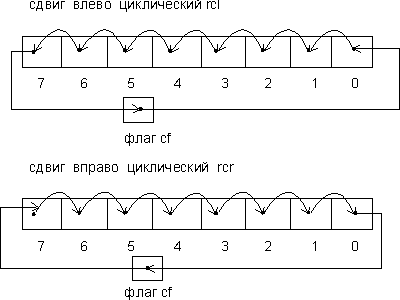
Команды циклического сдвига *через флаг переноса cf* отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а записывается сначала в флаг переноса cf. *Лишь следующее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита с другого конца операнда* (см. рис. 4).

[**rcl**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#rcl) **операнд,счетчик\_сдвигов** (Rotate through Carry Left) — циклический сдвиг влево через перенос.

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

[**rcr**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#rcr) **операнд,счетчик\_сдвигов** (Rotate through Carry Right) — циклический сдвиг вправо через перенос.

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



Из рис. 4 видно, что при сдвиге через флаг переноса появляется промежуточный элемент, с помощью которого, в частности, можно производить подмену циклически сдвигаемых битов, в частности, *рассогласование* битовых последовательностей.

Под рассогласованием битовой последовательности здесь и далее подразумевается действие, которое позволяет некоторым образом локализовать и извлечь нужные участки этой последовательности и записать их в другое место

Пример. Дано отрицательное число. Выведите на экран его значение по модулю деленное на 2.

Любое отрицательное число хранится в дополнительном формате

-1 ffh

-2 feh

…

-10 f6h

получить значение числа по модулю, можно осуществив логическое отрицание над числом и добавив 1.

**model small**

**.stack 100h**

**.data**

x db -12

**.code**

**start:**

mov ax,@data

**mov ds,ax**

mov al,x ;в al отрицательное число

not al

inc al ;число по модулю

shr al,1

;выводим результат на экран

aam ;

;преобразование двоичного числа меньшего 63h (9910), которое находится в al в его ;неупакованный BCD-эквивалент

; -разделить значение регистра al на 10;

; -записать частное в регистр ah, остаток — в регистр al.

mov dx,ax ;число в регистр dx

or dx,3030h ;получаю ASCII код числа

xchg dh,dl ;меняю местами старший и младший байт, для вывода символа из dl

mov ah,02h ;

int 21h ;вывожу старшую половинку числа

xchg dh,dl ;меняю местами старший и младший байт,

int 21h ;вывожу младшую половинку числа

**mov ax,4c00h**

**int 21h**

**end start**

**Команды передачи управления**

По принципу действия, команды микропроцессора, обеспечивающие организацию переходов в программе, можно разделить на три группы:

1. Команды безусловной передачи управления:

- [команда безусловного перехода](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20); jmp

- [вызова процедуры и возврата из процедуры](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20); call, ret

- вызова программных прерываний и возврата из программных прерываний. Int, iret

2. Команды [условной передачи управления](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20):

- [команды перехода по результату команды сравнения cmp](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20);

- команды [перехода по состоянию определенного флага](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20);

- команды [перехода по содержимому регистра ecx/cx](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20).

3. Команды [управления циклом](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20):

- команда [организации цикла со счетчиком ecx/cx](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20);

- команда организации цикла со счетчиком ecx/cx с возможностью [досрочного выхода из цикла по дополнительному условию](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson10%22%20l%20).

*jmp* адрес\_перехода - безусловный переход без сохранения информации о точке возврата. Аналог goto.

jmp m1 m4:

… …

m1: jmp m4

### **Условные переходы**

Команды условного перехода имеют одинаковый синтаксис:

**jcc метка\_перехода**

Мнемокод всех команд начинается с “**j**” — от слова *jump* (прыжок), *cc* — определяет конкретное условие, анализируемое командой. Что касается операнда *метка\_перехода*, то эта метка может находится только в пределах текущего сегмента кода, межсегментная передача управления в условных переходах не допускается.

Для того чтобы принять решение о том, куда будет передано управление командой условного перехода, предварительно должно быть сформировано условие, на основании которого и будет приниматься решение о передаче управления. Источниками такого условия могут быть:

* + - любая команда, изменяющая состояние арифметических флагов;
    - команда сравнения **cmp**, сравнивающая значения двух операндов;
    - состояние регистра ecx/cx.

Условные переходы по содержимому флагов

|  |  |  |  |
| --- | --- | --- | --- |
| Название флага | Номер бита в eflags/flag | Команда условного перехода | Значение флага для осуществления перехода |
| Флаг переноса cf | 1 | jc | cf = 1 |
| Флаг четности pf | 2 | jp | pf = 1 |
| Флаг нуля zf | 6 | jz | zf = 1 |
| Флаг знака sf | 7 | js | sf = 1 |
| Флаг переполнения of | 11 | jo | of = 1 |
| Флаг переноса cf | 1 | jnc | cf = 0 |
| Флаг четности pf | 2 | jnp | pf = 0 |
| Флаг нуля zf | 6 | jnz | zf = 0 |
| Флаг знака sf | 7 | jns | sf = 0 |
| Флаг переполнения of | 11 | jno | of = 0 |

**jcxz метка\_перехода** (Jump if cx is Zero) — переход, если cx ноль;

**jecxz** метка\_перехода (Jump Equal ecx Zero) — переход, если **ecx** ноль.

Пример программы: определите, равны ли два числа вводимые пользователем с клавиатуры.

**model small**

**.stack 100h**

**.data**

s1 db 'числа равны$'

s2 db 'числа не равны$'

.code

start:

mov ax,@data

mov ds,ax

mov ah,01h

int 21h ;ввели первое число

mov dl,al

mov ah,01h

int 21h ;ввели второе число

sub al,dl ;сравнили числа

jnz m1

mov dx, offset s1

jmp m2

m1: mov dx, offset s2

m2: mov ah,09h

int 21h ;выводим информационную строку

**mov ax,4c00h**

**int 21h**

**end start**

Команда сравнения cmp

*cmp* операнд\_1,операнд\_2 - сравнивает два операнда и по результатам сравнения устанавливает флаги. Команда сравнения cmp имеет интересный принцип работы. Он абсолютно такой же, как и у команды вычитания sub. Единственное, чего она не делает — это запись результата вычитания на место первого операнда.

*Алгоритм работы:*

-выполнить вычитание (операнд1-операнд2);

-в зависимости от результата установить флаги, операнд1 и операнд2 не изменять (то есть результат не запоминать).

Условные переходы после команд сравнения

|  |  |  |  |
| --- | --- | --- | --- |
| Типы операндов | Мнемокод команды условного перехода | Критерий условного перехода | Значения флагов для осществления перехода |
| Любые | je | операнд\_1 = операнд\_2 | zf = 1 |
| Любые | jne | операнд\_1<>операнд\_2 | zf = 0 |
| Со знаком | jl/jnge | операнд\_1 < операнд\_2 | sf <> of |
| Со знаком | jle/jng | операнд\_1 <= операнд\_2 | sf <> of or zf = 1 |
| Со знаком | jg/jnle | операнд\_1 > операнд\_2 | sf = of and zf = 0 |
| Со знаком | jge/jnl | операнд\_1 => операнд\_2 | sf = of |
| Без знака | jb/jnae | операнд\_1 < операнд\_2 | cf = 1 |
| Без знака | jbe/jna | операнд\_1 <= операнд\_2 | cf = 1 or zf=1 |
| Без знака | ja/jnbe | операнд\_1 > операнд\_2 | cf = 0 and zf = 0 |
| Без знака | jae/jnb | операнд\_1 => операнд\_2 | cf = 0 |

Пример программы: определите, равны ли два числа вводимые пользователем с клавиатуры.

**model small**

**.stack 100h**

**.data**

s1 db 'числа равны$'

s2 db 'числа не равны$'

.code

start:

mov ax,@data

mov ds,ax

mov ah,01h

int 21h ;ввели первое число

mov dl,al

mov ah,01h

int 21h ;ввели второе число

cmp al,dl ;сравнили числа

jne m1

mov dx, offset s1

jmp m2

m1: mov dx, offset s2

m2: mov ah,09h

int 21h ;выводим информационную строку

**mov ax,4c00h**

**int 21h**

##### **end start**

##### **Организация циклов**

*loop* метка\_перехода (Loop) — повторить цикл

Работа команды заключается в выполнении следующих действий:

- декремента регистра ecx/cx;

- сравнения регистра ecx/cx с нулем:

- если (ecx/cx) > 0, то управление передается на метку перехода;

- если (ecx/cx) = 0, то управление передается на следующую после loop команду

mov cx, количество циклов

м1: тело цикла

loop m1

*loope/loopz* метка\_перехода (Loop till cx <> 0 or Zero Flag = 0) — повторить цикл, пока cx <> 0 или zf = 0.

*loopne/loopnz* метка\_перехода (Loop till cx <> 0 or Not Zero flag=0) — повторить цикл пока cx <> 0 или zf = 1

Недостаток команд организации цикла loop, loope/loopz и loopne/loopnz в том, что они реализуют только короткие переходы (от –128 до +127 байт).

Организация вложенных циклов

mov cх,n ; в сх заносим количество итераций внешнего цикла

m1:

push cx

…

Внешний цикл

mov cx,n1; в сх заносим количество итераций внутреннего цикла

m2:

тело внутреннего цикла

loop m2

…

pop cx

loop m1

Пример программы: Напишите программу подсчета у=1+2+3+…+n, n не более 10000.

model small

.stack 100h

.data

yb dd ?

ym dw ?

s1 db 'введите n',10,13,'$'

.code

start:

mov ax,@data

mov ds,ax

mov dx, offset s1

mov ah,09h

int 21h

mov cx,3

m: shl bx,4

mov ah,01h

int 21h вводим n в регистр bx

sub ax,130h

add bx,ax

loop m

mov cx,bx

xor dx,dx

xor al,al

m1: add dx,cx считаем у

jnc m2

mov al,1

m2: loop m1

cmp al,1

je m3

mov ym,dx

m3: mov yb,edx

mov ax,4c00h

int 21h

end start

**Цепочечные команды**

(Команды обработки строк символов)

**Цепочка** – это последовательность элементов, размер которых может быть байт, слово, двойное слово. Содержимое этих элементов может быть любое – символы, числа.

В системе команд микропроцессора имеется семь *операций-примитивов* обработки цепочек.

Каждая из них реализуется в микропроцессоре тремя командами, в свою очередь, каждая из этих команд работает с соответствующим размером элемента — байтом, словом или двойным словом.

Вместе с цепочечными командами обычно применяют **префиксы повторений**, которые ставятся перед командой в поле [метки]. Цепочечная команда без префикса выполняется один раз. С префиксом цепочечные команды выполняются циклично.

*rep* (REPeat) - команда выполняется, пока содержимое в ecx/cx не станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически уменьшает содержимое ecx/cx на единицу. Та же команда, но без префикса, этого не делает.

*repe* или *repz* (REPeat while Equal or Zero) - команда выполняется до тех пор, пока содержимое ecx/cx не равно нулю или флаг zf равен 1. Как только одно из этих условий нарушается, управление передается следующей команде программы

*repne* или *repnz* (REPeat while Not Equal or Zero) - команда циклически выполняется до тех пор, пока содержимое ecx/cx не равно нулю или флаг zf равен нулю. При невыполнении одного из этих условий работа команды прекращается.

***Формирования физического адреса операндов*** адрес\_источника и адрес\_приемника происходит следующим образом:

адрес\_источника — пара **ds:esi/si**;

адрес\_приемника — пара **es:edi/di**

Важный момент, касающийся всех цепочечных команд, — это ***направление обработки цепочки***. Есть две возможности:

- от начала цепочки к ее концу, то есть в направлении возрастания адресов;

- от конца цепочки к началу, то есть в направлении убывания адресов.

Цепочечные команды сами выполняют модификацию регистров, адресующих операнды, обеспечивая тем самым автоматическое продвижение по цепочке. Количество байт, на которые эта модификация осуществляется, определяется кодом команды. А вот знак этой модификации определяется значением флага направления **df** (Direction Flag) в регистре eflags/flags. Состоянием флага **df** можно управлять с помощью двух команд, не имеющих операндов:

[**cld**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#cld) (Clear Direction Flag) — очистить флаг направления **df = 0**, значение индексных регистров esi/si и edi/di будет автоматически увеличиваться (операция инкремента) цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов;

[**std**](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#std) (Set Direction Flag) — установить флаг направления **df = 1**, то значение индексных регистров esi/si и edi/di будет автоматически уменьшаться (операция декремента) цепочечными командами, то есть обработка будет идти в направлении убывания адресов.

*Типовой набор действий* для выполнения любой цепочечной команды:

* Установить значение флага df в зависимости от того, в каком направлении будут обрабатываться элементы цепочки — в направлении возрастания или убывания адресов.
* Загрузить указатели на адреса цепочек в памяти в пары регистров ds:(e)si и es: (e)di.
* Загрузить в регистр ecx/cx количество элементов, подлежащих обработке.
* Выдать цепочечную команду с префиксом повторений.

**Пересылка цепочек**

[movs](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideLessonLesson11%22%20l%20) адрес\_прием, адрес\_источника (MOVe String)- переслать цепочку;

[movsb](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideTextCommand.htm#movs) MOVe String Byte) — переслать цепочку байт;

*movsw* (MOVe String Word) — переслать цепочку слов;

*movsd* (MOVe String Double word) — переслать цепочку двойных слов.

Команда копирует байт, слово или двойное слово из цепочки источника, в цепочку приемника. Размер пересылаемых элементов ассемблер определяет, исходя из атрибутов идентификаторов. К примеру, если эти идентификаторы были определены директивой db, то пересылаться будут байты, если идентификаторы были определены с помощью директивы dd, то пересылке подлежат двойные слова.

Для цепочечных команд с операндами типа **movs адрес\_приемника,адрес\_источника**, не существует машинного аналога. При трансляции в зависимости от типа операндов транслятор преобразует ее в одну из трех машинных команд: **movsb, movsw** или **movsd**.

Сама по себе команда movs пересылает только один элемент, исходя из его типа, и модифицирует значения регистров esi/si и edi/di. Если перед командой написать префикс **rep**, то одной командой можно переслать до 64 Кбайт данных. Число пересылаемых элементов должно быть загружено в *счетчик* — регистр **cx** (use16) или **ecx** (use32).

**Пример проги. Пересылка строк командой movs**

MODEL small

STACK 256

.data

source db 'Тестируемая строка','$' ;строка-источник

dest db 19 DUP (' ') ;строка-приёмник

.code

main:

mov ax,@data ;загрузка сегментных регистров

mov ds,ax ;настройка регистров DS и ES на адрес сегмента данных

mov es,ax

cld ;сброс флага DF — обработка строки от начала к концу

lea si,source ;загрузка в si смещения строки-источника

lea di,dest ;загрузка в DS смещения строки-приёмника

mov cx,20 ;для префикса rep — счетчик повторений (длина строки)

rep movs dest,source ;пересылка строки

lea dx,dest

mov ah,09h ;вывод на экран строки-приёмника

int 21h

mov ax,4c00h

int 21h

end main

## Операция сравнения цепочек

*cmps адрес\_приемника,адрес\_источника*(CoMPare String) — сравнить строки;

*cmpsb* (CoMPare String Byte) — сравнить строку байт;

*cmpsw* (CoMPare String Word) — сравнить строку слов;

*cmpsd* (CoMPare String Double word) — сравнить строку двойных слов.

Алгоритм работы команды cmps заключается в последовательном выполнении вычитания (элемент цепочки-источника — элемент цепочки-получателя) над очередными элементами обеих цепочек. Принцип выполнения вычитания командой cmps аналогичен команде сравнения cmp. Она, так же, как и cmp, производит вычитание элементов, не записывая при этом результата, и устанавливает флаги zf, sf и of.

После выполнения вычитания очередных элементов цепочек командой cmps, индексные регистры esi/si и edi/di автоматически изменяются в соответствии со значением флага df на значение, равное размеру элемента сравниваемых цепочек.

Чтобы заставить команду cmps выполняться несколько раз, то есть производить последовательное сравнение элементов цепочек, необходимо перед командой cmps определить префикс повторения. С командой cmps можно использовать префикс повторения **repe/repz** или **repne/repnz**:

* repe или repz — если необходимо организовать сравнение до тех пор, пока не будет выполнено одно из двух условий: достигнут конец цепочки (содержимое ecx/cx равно нулю) или в цепочках встретились разные элементы (флаг zf стал равен нулю);
* repne или repnz — если нужно проводить сравнение до тех пор, пока: не будет достигнут конец цепочки (содержимое ecx/cx равно нулю) или в цепочках встретились одинаковые элементы (флаг zf стал равен единице).

#### Пример программы Сравнение двух строк командой cmps

MODEL small

STACK 256

.data

sov db 0ah,0dh,'Строки совпадают.','$'

nesov db 0ah,0dh,'Строки не совпадают','$'

s1 db '0123456789',0ah,0dh,'$';исследуемые строки

s2 db 10

s3 db 11 dup (0)

.code

main:

mov ax,@data ;загрузка сегментных регистров

mov ds,ax

mov es,ax ;настройка ES на DS

;вводим строку

mov ah, 0аh

mov dx, offset s2

int 21h

;поиск совпадающих элементов, сброс флага DF - сравнение в направлении возрастания адресов

cld

lea si,s1 ;загрузка в si смещения string1

lea di,s3 ;загрузка в di смещения string2

mov cx,10 ;длина строки для префикса repe

repe cmpsb ;сравнение строк (пока сравниваемые элементы строк равны)

;выход при обнаружении не совпавшего элемента

jcxz equal ;cx=0, то есть строки совпадают

lea dx, nesov

jmp exit ;выход

equal: lea dx, sov

exit: mov ah,09h

int 21h ;вывод сообщения

mov ax,4c00h

int 21h

end main ;конец программы

## Операция сканирования цепочек

*scas адрес\_приемника* (SCAning String) — сканировать цепочку;

*scasb* (SCAning String Byte) — сканировать цепочку байт;

*scasw* (SCAning String Word) — сканировать цепочку слов;

*scasd* (SCAning String Double Word) — сканировать цепочку двойных слов

Эти команды осуществляют поиск искомого значения, которое находится в регистре al/ax/eax. Принцип поиска тот же, что и в команде сравнения cmps, то есть последовательное выполнение вычитания

(содержимое регистра\_аккумулятора – содержимое очередного\_элемента\_цепочки).

В зависимости от результатов вычитания производится установка флагов, при этом сами операнды не изменяются.

Так же, как и в случае команды cmps, с командой scas удобно использовать префиксы **repe/repz** или **repne/repnz**:

* + - **repe** или **repz** — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий: достигнут конец цепочки (содержимое ecx/cx равно 0) или в цепочке встретился элемент, отличный от элемента в регистре al/ax/eax;
    - **repne** или **repnz** — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий достигнут конец цепочки (содержимое ecx/cx равно 0)или в цепочке встретился элемент, совпадающий с элементом в регистре al/ax/eax.

#### Пример проги. найти количество \* в строке

MODEL small

STACK 256

.data

s1 db 20

s2 db 21 dup ?

s3 db 'количество \* в строке',0ah,0dh,'$'

.code

main:

mov ax,@data

mov ds,ax

mov es,ax ;настройка ES на DS

;вводим строку

mov ah,0ah

mov dx, offset s1

int 21h

;поиск \*

mov al,'\*' ;символ для поиска — `а`(кириллица)

cld ;сброс флага df

**lea di, s2** ;загрузка в es:di смещения строки

inc di ;первый элемент в s3 это количество введенных символов, его игнорируем

xor bl ;обнуляем счетчик звездочек

mov cx,20 ;для префикса repne — длина строки

m1: repne scasb ;пока искомый символ и символ в строке не совпадут идет поиск, ;выход при совпадении

je found ;если равны - переход на обработку

;вывод количества звездочек в строке

mov ah,09h

mov dx,offset s3

int 21h ;вывод сообщения nochar

mov al,bl

aam

or ax, 3030h

mov dx,ax

xchg dh,dl

mov ah, 02h

int 21h

xchg dh,dl

int 21h

mov ax,4c00h

int 21h

;суммируем количество звездочек

found: inc bl

jmp m1

end main

## Загрузка элемента цепочки в аккумулятор

*lods адрес\_источника*(LOaD String) — загрузить элемент из цепочки в регистр-аккумулятор al/ax/eax;

*lodsb* (LOaD String Byte) — загрузить байт из цепочки в регистр al;

*lodsw* (LOaD String Word) — загрузить слово из цепочки в регистр ax;

*lodsd* (LOaD String Double Word) — загрузить двойное слово из цепочки в регистр eax.

Эта операция-примитив позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор al, ax или eax. Эту операцию удобно использовать вместе с поиском (сканированием) с тем, чтобы, найдя нужный элемент, извлечь его (например, для изменения).

## Перенос элемента из аккумулятора в цепочку

## stos адрес\_приемника (STOre String) — сохранить элемент из регистра-аккумулятора al/ax/eax в цепочке;

## stosb (STOre String Byte) — сохранить байт из регистра al в цепочке;

## stosw (STOre String Word) — сохранить слово из регистра ax в цепочке;

## stosd (STOre String Double Word) - сохранить двойное слово из регистра eax в цепочке.

Эта операция-примитив позволяет произвести действие, обратное команде lods, то есть сохранить значение из регистра-аккумулятора в элементе цепочки. Эту операцию удобно использовать вместе с операцией поиска (сканирования) scans и загрузки lods, с тем, чтобы, *найдя нужный элемент, извлечь его в регистр и записать на его место новое значение*.

Сложные структуры данных

Одномерные массивы

Все элементы массива располагаются в памяти последовательно

Описание элементов массива

mas db 1,2,3,4,5

mas dw 5 dup (0)

Доступ к элементам массива

mov ax,mas[si] ; в si номер элемента в массиве

mov mas[si], ax ; в di номер элемента в массиве

Используя команды i486 можно использовать адресацию с масштабированием, при размере элементов больше байта

Mov ax, mas[si\*2] ;

**Пример программы Найти в строке хотя бы один нулевой элемент**

model small

.stack 100h

.data

bufer dw 25 ;формирую размер буфера для ввода строки

mas dw 25 dup (' ') ;формирую буфер

adr dw bufer ;описываю адрес

subj1 db ‘в строке найден нулевой элемент', '$'

subj2 db ‘в строке не найден нулевой элемент', '$'

.code

main:

mov ax,@data

mov ds,ax

mov ah,0ah

mov dx, adr

int 21h ; ввод строки с клавиатуры

;поиск нулевого элемента

xor si, si

mov cx, mas[si] ;загружаем в сх количество элементов в строке

mov ax, 030h ;в ax загружаем ASCII код нуля

m1: inc si либо **inc si**

inc si **cmp ax, mas[si\*2]**

cmp ax, mas[si]

je m2 ;если в строке найдем нулевой элемент, то выходим из цикла на вывод subj1

loop m1

;нормальный выход из цикла означает что в строке нет нулевых элементов

mov ah,09h

lea dx, subj2

int 21h

jmp exit

m2: mov ah, 09h

lea dx,subj1

int 21h

exit: mov ax,4c00h

int 21h

end main

**Двумерные массивов**

**!Специальных средств для описания двумерных массивов в ассемблере нет!**

Двумерный массив описывается также как и одномерный массив, отличие заключается в трактовке расположения элементов. Пусть последовательность элементов трактуется как двумерный массив, расположенный по строкам, тогда адрес элемента [i,j] вычисляется так

База+колич\_элем\_строке\*размер\_элем\*I+j

Для определения базы используют имя массива, для второго слагаемого регистр bx , для третьего si, это базово-индексная адресация.

Описание массива:

Mas1 db 10 dup (3 dup (?))

Mas2 db 1,2,3,4,5

3,4,5,6,7

4,7,9,2,0

**Пример** поиска максимального элемента в каждой строке однобайтного массива mas, размером 5\*10, с занесением максимальных элементов в массив max (1\*5). Инициализацию массива mas рассматривать не будем.

…

xor di, di ;обнуляем индексы массива max

xor bx, bx ;обнуляем индексы строк массива mas

xor si, si ;обнуляем индексы столбцов массива mas

mov cx,5 ;в cx количество строк, внешний цикл

m1: push cx

mov cx, 10 ;в сх количество столбцов, внутренний цикл

mov al, mas[si+bx];первый элемент из 1 строки mas в аl

m2: inc si

cmp al, mas[si+bx] ;сравниваем со следующим элем. строки

jb m3 ;если меньше на m3

mov al, mas[si+bx] ;иначе в аl заносим больший элемент

m3: loop m2 ;после выхода из цикла в ах максимальный элемент в данной строке

mov max[di],al ;кладем максимальный элемент в массив max

inc di

xor si,si ;обнуляем номер столбца

add bx, 10 ;переходим на следующую строку

pop cx ;достаем сх

loop m1

…

**Структура** – это тип данных, состоящий из фиксированного числа элементов разного типа.

Для использования структур в программе необходимо выполнить три действия:

* 1. **Задать шаблон структуры**. По смыслу это означает определение нового типа данных (схемы или шаблона), который впоследствии можно использовать для определения переменных этого типа. Память при этом не выделяется, это информация для транслятора о расположении полей и их значению по умолчанию.

Синтаксис описания шаблона структуры:

***имя\_структуры STRUC***

***<описание полей>*** ; последовательность директив описания данных dd,dw,db…

***имя\_структуры ENDS***

* 1. **Определить экземпляр структуры.** Этот этап подразумевает инициализацию конкретной переменной заранее определенной (с помощью шаблона) структурой. В данном случае транслятору дается указание выделить память и присвоить этой области символическое имя.

**Описать структуру в программе можно только один раз, а определить – любое количество раз.**

Определение данных с типом структуры имеет следующий вид:

***[имя переменной] имя\_структуры <[список значений]>***

* 1. Организовать [обращение к элементам структуры](D:DASHKALibrari%20ot%20DariaAssemblerAssemblerguideText%22%20l%20).

Для того чтобы сослаться в команде на поле некоторой структуры, используется следующее выражение:

***имя\_переменной.имя\_поля\_структуры*,**

**Фрагмент программы.** Найти из списка студентов отличника по всем предметам.

Model small

.586p

Stud struc

Name db 20 dup (‘’)

Phisika db ?

Matem db ?

Stud ends

.data

Bufer Stud <””,’’,’’,’’> ;зарезервировали пустую структуру

S1 Stud <”Ivanov”,’5’,’5’,’5’>

S2 Stud <”Petrov”,’3’,’3’,’3’>

S3 Stud <”Сидоров”,’5’,’2’,’5’>

…

mov bx, offset s1

mov al, [bx].Phisika

cmp al, ‘5’

jne m2

mov al, [bx].Matem

cmp al, ‘5’

jne m2

;нашли одного из отличников, надо вывести его фамилию

m2: mov al, s2.Phisika ;ищем следующего отличника

mov ah, s2.Matem

cmp ax, ‘55’

jne m3

;нашли одного из отличников, надо вывести его фамилию

m3: ;ищем следующего отличника

**Объединение -** тип данных, позволяющий трактовать одну и ту же область памяти как имеющую разные типы и имена.

Описаниеобъединений в программе напоминает описание структур, то есть сначала описывается шаблон, в котором с помощью директив описания данных перечисляются имена и типы полей:

***имя\_объединения UNION***

***<описание полей>***

***имя\_объединения ENDS***

Отличие объединений от структур состоит, в частности, в том, что при определении переменной типа объединения память выделяется в соответствии с размером максимального элемента. Обращение к элементам объединения происходит по их именам, но при этом нужно, конечно, помнить о том, что все поля в объединении накладываются друг на друга.   
Одновременная работа с элементами объединения исключена. В качестве элементов объединения можно использовать и структуры.

Model small

.586p

st union

stu1 dw ?

stu2 db ?

stu3 dd ?

st ends

.data

q st <> ;пустое объединение

z st <12ffh> ;заполнили объединение значением

…

mov q.stu2, al ;в переменную положили содержимое регистра al

mov q.stu3, edx

mov bx, z.stu1

mov dx, offset q.stu1

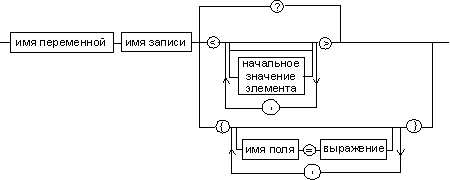
**Запись** - структурный тип данных, состоящий из фиксированного числа элементов длиной от одного до нескольких бит. При описании записи для каждого элемента указывается его длина в битах и, что необязательно, некоторое значение. Суммарный размер записи определяется суммой размеров ее полей и не может быть более **8, 16** или **32** бит. Если суммарный размер записи меньше указанных значений, то все поля записи “прижимаются” к младшим разрядам

Использование записей в программе, так же, как и структур, организуется в три этапа:

1. Описание шаблона записи

*имя\_записи RECORD <описание элементов>*

2. Для использования шаблона записи в программе необходимо определить переменную с типом данной записи, для чего применяется следующая синтаксическая конструкция (рис. 7):



3. Организация работы с записями. Обычные механизмы адресации бессильны, поскольку они работают на уровне байтов, а не отдельных битов.

* + - каждому имени элемента записи ассемблер присваивает числовое значение, равное количеству сдвигов вправо, которое нужно произвести, для того чтобы этот элемент оказался прижатым к началу ячейки памяти;
    - размер элемента записи в битах можно узнать с помощью оператора width;
    - оператор mask позволяет локализовать биты нужного элемента записи;
    - все действия по преобразованию элементов записи производятся с помощью логических команд;
    - команда setfield устанавливает значение некоторого поля записи

*setfield имя\_элемента\_записи регистр\_ назначение, регистр\_источник*

* + - команда getfield осуществляет выборку некоторого поля записи

*getfield имя\_элемента\_записи регистр\_назначение, регистр\_ источник*

Процедуры. Макрокоманды

***Процедура***, часто называемая подпрограммой, - это правильным образом оформленная совокупность команд, которая будучи однократно описана, при необходимости может быть вызвана в любом месте программы. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры.

Описание процедуры может размещается в любом месте программы, но таким образом чтобы на нее случайным образом не попало управление:

* + - в начале программы, до первой исполняемой команды;
    - в конце, после команды возвращающей управление операционной системе;
    - промежуточный вариант, тело процедуры располагается внутри другой процедуры или основной программы. В этом случае необходимо предусмотреть обход процедуры командой jmp;
    - в другом модуле.

Синтаксис описания процедуры:

*Имя\_процедуры PROC* заголовок

Команды, директивы тело процедуры

*[ret]* возврат из процедуры

*[имя\_процедуры] ENDP* конец процедуры

Вызов процедуры осуществляется командой

*CALL [модификатор] имя\_процедуры*

Команда call передает управление по адресу с символическим адресом имя\_процедуры, с сохранением в стеке адреса возврата, команды следующей после команды call.

Возврат из процедуры осуществляется по команде

*RET [число]*

Команда ret считывает адрес возврата из стека и загружает его в регистры cs и ip/eip, возвращая таким образом управление команде, следующей за командой call. Число – необязательный параметр, обозначающий количество элементов, удаляемых из стека при возврате из процедуры. Размер элемента зависит от используемой модели сегментации 32 или 16 разрядной.

Передача аргументов из/в процедуру может осуществляться через регистры, переменные или стек.

**Пример.**

|  |  |
| --- | --- |
| Model small  .stack 100h  .data  w db 25 dup (?)  .code  vvod proc  mov ah, 0ah  lea dx, w  int 21h  ret  vvod endp  main: | …  Call schet  Call vvod  …  exit:  mov ax,4c00h  int 21h  schet proc  ..  ret  schet endp  end main |

Макрокоманда является одним из многих механизмов замены текста программы. С помощью макрокоманды в текст программы можно вставлять последовательности строк и привязывать их к месту вставки.

**Макрокоманда** представляет собой строку, содержащую некоторое имя – имя макрокоманды, предназначенное для того, чтобы быть замещенным одной или несколькими другими строками при трансляции.

Для работы с макрокомандой вначале необходимо задать ее шаблон-описание, так называемое макроопределение.

Имя\_макрокоманды MACRO [список\_формальных\_аргументов]

<Тело макроопределения>

ENDM

Существует три варианта расположения макроопределений:

* + - в начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы. В данном случае макрокоманды будут актуальны только в пределах этой программы;
    - в отдельном файле. Для того, чтобы использовать эти макроопределения в других программах, необходимо в начале исходного текста этих программ записать директиву

include имя\_файла

* + - в макробиблиотеке. Макробиблиотека создается в том случае, когда написанные макросы используются практически во всех программах. Подключается библиотека директивой include. Недостаток этого и предыдущего методов в том, что в исходный текст программы включаются абсолютно все макроопределения. Для исправления ситуации можно использовать директиву purge, в качестве операндов которой перечисляются макрокоманды, которые не должны включаться в текст программы.

Include macrobibl.inc ;в исходный текст программы будут вставлены строки из macrobibl.inc

Purge outstr, exit ;за исключением макроопределений outstr, exit

Активизация макроса осуществляется следующим образом:

*Имя\_макрокоманды список\_ фактических\_ аргументов*

|  |  |
| --- | --- |
| Model small  Vivod macro rg  Mov dl, rg  Mov ah, 02h  Int 21h  endm  .data  ..  .code  ..  vivod al  .. | Model small  sravnenie macro rg, met  cmp rg , ‘a’  ja met  add rg, 07h  met: add rg, 30h  endm  .data  ..  .code  ..  sravnenie al, m1  .. |

Функционально макроопределения похожи на процедуры. Сходство их в том, что и те, и другие достаточно один раз где-то описать, а затем вызывать их специальным образом. На этом их сходство заканчивается, и начинаются различия, которые в зависимости от целевой установки можно рассматривать и как достоинства и как недостатки:

- в отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды. Процедуры в этом отношении объекты менее гибки;

- при каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу. При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным, хотя быстродействие несколько снижается за счет необходимости осуществления переходов.

**Подключение процедур и макросов во внешнем файле**

Внешний файл с расширением inc, а в файле которой использует процедуру или макрос присутствуют следующие строки

Model small

Include [NAME].INC

А вызов макросов и процедур как обычно.

Работа с портами ввода вывода

Адресное пространство памяти в любой микропроцессорной системе семейства х86 делится на пространство адресов памяти и пространство портов ввода/вывода. Это обусловлено архитектурной реализацией и исторической эволюцией процессоров х86.

Для обращения к пространству ввода/вывода используются команды *in, out, ins, outs*.

*In регистр, номер порта* – ввод данных из порта в регистр

*Out номер порта, регистр* – вывод данных из регистра в порт

*Ins, Outs* - работают с элементами строки памяти.

In ax, 064h ; запись слова в ах из порта 064h

Out 064,al ; запись байта в порт

In ax, dx ; косвенная адресация порта через регистр dx, применяется при использовании 16 разрядного адреса порта

Через порты ввода/вывода осуществляется программирование, контроль и отладка работы периферийных устройств. Зачастую один адрес памяти может служить портом ввода при чтении данных, и портом вывода при записи.

Распределением адресов управляет BIOS через регистры конфигурирования чипсета. Обычно для совместимости аппаратного и программного обеспечения распределение адресов стандартно для любых микропроцессорных систем семейства х86.

***ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ***

Состав любой вычислительной системы одинаков: микропроцессор, электронная память подсистема ввода-вывода. Эти устройство объединяет системная шина, состоящая из следующих шин: данных, адреса и управления.

Каждый микропроцессор имеет определенное число элементов памяти, называемых регистрами, арифметико-логическое устройство и устройство управления. Регистры используются для временного хранения выполняемой команды, адресов памяти, обрабатываемых данных и другой внутренней информации МП. В АЛУ производится арифметическая и логическая обработка данных. Устройство управления реализует временную диаграмму и вырабатывает необходимые управляющие сигналы для внутренней работы МП и связи его с другой аппаратурой через внешние шины МП.

Структуры различных типов МП могут существенно различаться, однако наиболее важными параметрами являются архитектура, адресное пространство памяти, разрядность шины данных, быстродействие. Архитектуру МП определяет разрядность слова и внутренней шины данных МП. Первые МП основывалисьна4-разрядной архитектуре. Первые ПЭВМ использовали МП с 8-разрядной архитектурой, а современные МП основаны на МП с 64- и 32-разрядной архитектурой.

Микропроцессоры используют *последовательный принцип выполнения команд*, *принцип параллельной работы*, или *конвейерный метод* выполнения команд.

Последовательный принцип, при котором очередная операция начинается только после выполнения предыдущей.

Принцип параллельной работы, при которой одновременно с выполнением текущей команды производятся предварительная выборка и хранение последующих команд.

В МП с 32-разрядной архитектурой используется конвейерный метод выполнения команд, при котором несколько внутренних устройств МП работают параллельно, производя одновременно обработку нескольких последовательных команд программы.

Адресное пространство памяти определяется разрядностью адресных регистров и адресной шины МП. Для выборки команд и обмена данными с памятью МП имеют шину данных, разрядность которой, как правило, совпадает с разрядностью внутренней шины данных, определяемой архитектурой МП.

Одним из важных параметров МП является *быстродействие,* определяемое тактовой частотой его работы, которая обычно задается внешними синхросигналами. Выполнение простейших команд (например, сложение двух операндов из регистров или пересылка операндов в регистрах МП) требует минимально двух периодов тактовых импульсов (для выборки команды и ее выполнения). Более сложные команды требуют для выполнения до 10—20 периодов тактовых импульсов. Если операнды находятся не в регистрах, а в памяти, дополнительное время расходуется на выборки операндов в регистры и записи результата в память.

Скорость работы МП определяется не только тактовой частотой, но и набором его команд, их гибкостью, развитой системой прерываний.

**Электронная память**

Содержит операнды и программу, которую выполняет МП. Используются два типа электронной памяти: постоянные запоминающие устройства (ПЗУ) и оперативные запоминающие устройства (ОЗУ).

В ПЗУ хранится информация, которую ЭВМ может использовать сразу же после включения питания. Она включает программы инициализации периферийных микросхем, программы ядра ОС и программы обработки прерываний.

Постоянное запоминающее устройство является энергонезависимой памятью: после выключения питания информация в нем сохраняется. Информация в ОЗУ разрушается при выключении питания.

**В ОЗУ хранятся оперативные данные и программы, используемые МП. Поэтому микросхемы ОЗУ по быстродействию должны быть согласованы с МП, а емкость ОЗУ (вместе с ПЗУ) должна приближаться к пределу, определяемому адресным пространством МП.**

**Схемы ввода-вывода**

Связь МП с контроллерами ПУ обычно осуществляется через порты ввода-вывода под непосредственным управлением МП или под управлением специализированных контроллеров. Связь МП с ПУ производится через стандартизованные интерфейсы ПУ.

Организация и быстродействие схемы ввода-вывода влияет на быстродействие всей вычислительной системы.

**Микропроцессоры**

*Архитектура однокристального 16-ти разрядного микропроцессора К1810ВМ86.*

*Аналог I8086.*

Ориентирован на параллельное выполнение выборки и команд, может быть условно разделен на две части, работающие асинхронно: устройство сопряжения с магистралью (БИ – интерфейсный блок) и блок обработки (БО).

*Интерфейсный блок* обеспечивает формирование 20-разрядного физического адреса памяти, выборку команд и операндов из памяти, организацию очередности команд и запоминание результатов выполнения команд в памяти.

БИ состоит из очереди команд, сегментных регистров, регистра адреса команд, сумматора адреса (SM) и управления машинными циклами.

Устройство сопряжения готово выполнить цикл выборки слова из памяти всякий раз, когда в очереди освобождаются, по меньшей мере, два байта, БО извлекает из нее коды команд по мере необходимости. Очередь организована по принципу «первым пришел — первого обслужили», а шесть ее уровней позволяют удовлетворять запросы БО в кодах команд достаточно эффективно, сокращая тем самым до минимума затраты времени МП на ожидание выборки команд из памяти. Выполнение команд происходит в логической последовательности, предписанной программой, поскольку в очереди находятся те команды, которые хранились в ячейках памяти, непосредственно следующих за текущей командой. При передаче управления в другую ячейку памяти ход выполнения программы нарушается. Устройство сопряжения очищает регистры очереди, выбирает команду по адресу перехода, передает ее УО и начинает новое заполнение этих регистров. При возврате из подпрограммы или из прерывания происходит восстановление очереди команд, адреса которых автоматически вычисляются в СМА. Если МП необходимо выполнить цикл чтения или записи, то выборка команд приостанавливается на время цикла.

*Блок обработки* предназначен для выполнения операций по обработке данных и состоит из блока микропрограммного управления (БМУ), АЛУ, восьми регистров общего назначения (РОН) и регистра флагов (F).

Команды, выбранные БИ из памяти и записанные в очередь команд, по запросам от БО поступают в БМУ. Это устройство, содержащее память микрокоманд, декодирует команды и вырабатывает последовательность микрокоманд, управляющую процессом обработки. В АЛУ выполняются арифметические и логические операции над 8- и 16-разрядными числами с фиксированной запятой.

*Программно-доступными функциональными частями* МП являются регистры общего назначения (для хранения операндов и результатов выполнения команд), сегментные (для хранения базовых адресов текущих сегментов памяти), адреса команд и регистр флагов.



***Функциональное назначение выводов микропроцессора* *К1810ВМ86***

Вход MN/MX служит для выбора режима функционирования, который предлагает пользователю выбор состава выходных управляющих сигналов в соответствии со степенью сложности проектируемой МП-системы.

В минимальном режиме (вывод MN/MX подключен к шине питания), ориентированном на малые вычислительные системы, МП выдает сигналы управления обменом с памятью и внешними устройствами, а также обеспечивает доступ к системной магистрали по запросу прямого доступа к памяти, используя сигналы HOLD и HLDA. Если вывод MN/MX подключен к шине «Земля» (общий), то МП находится в максимальном режиме и может работать в сложных одно- и многопроцессорных системах. При работе в этом режиме изменяются функции ряда выводов МП.

|  |  |
| --- | --- |
| **Обозначение выводов** | **Функциональное назначение выводов** |
| AD0 .. AD15 | 16-разрядная двунаправленная мультиплексированная шина адреса/данных |
| A16/S3 .. A19/S6 | 4-х разрядная выходная шина микропроцессора, по которой в такте Т1 передаются 4 старших разряда адреса памяти, а в тактах Т2, Т3, Т4, при выполнении операций обращения к памяти и области ввода/вывода – признаки состояния микропроцессора.  S4, S3 – указывают номер одного из 4 сегментных регистров, который в данном цикле участвует в формировании исполнительного адреса.  S5 – указывает состояние триггера разрешения прерывания  S6 – всегда равен 0. |
| BHE/S7 | Выход, 0 на котором в Т1 указывает, что по шине адреса/данных передаются 8-разрядное слово. Сделано для совместимости со старым ПО.  В тактах Т2, Т3, Т4 на этом выходе присутствует S7 – признак состояния МП. Если S7=1 – МП находится в состоянии захвата шин внешним устройством. |
| RD | Чтение, выход, 0 на котором означает, что МП осуществляет чтение из памяти или портов ввода/вывода. |
| RЕАDY | Готовность, вход для подачи сигнала окончания цикла работы устройств памяти или ввода/вывода. Используется для синхронизации более медленных памяти или ВУ. |
| INTR | Вход маскируемых запросов на прерывание. Наличие запроса на этом входе анализируется в конце выполнения каждой команды |
| TEST | Вход, проверяемый по команде WFT |
| NMI | Вход немаскируемых запросов на прерывание |
| RESET | Вход начальной установки микропроцессора |
| CLK | Вход для подачи тактовых импульсов |
| MN/MX | Вход для подачи сигнала переключения минимального/максимального режима. |
| *Минимальный режим* | |
| INTA | Подтверждение прерывания, выходной сигнал стробирующий ввод информации в МП из источника прерывания, вызвавшего переход в режим прерывания |
| ALE | Строб адреса, выходной сигнал, стробирующий в такте Т1 передачу адресной информации с шин МП AD0 .. AD15 на другие элементы системы |
| DEN | Разрешение обмена данными, выходной сигнал, управляющий выдачей информации из шинных формирователей при выполнении команд чтения/записи. |
| DT/R | Ввод/вывод данных, выходной сигнал, указывающий на направление передачи. 1 – МП выдает информацию, 0 – МП принимает данные. |
| M/IO | Память/внешнее устройство, выходной сигнал отличающий передачу данных для памяти или для внешнего устройства. |
| WR | Запись, выходной сигнал указывающий на то, что МП выдает информацию для записи в память/ВУ. |
| HLDA | Разрешение прямого доступа, подтверждение захвата шин МП внешним устройством. |
| HOLD | Запрос прямого доступа (захвата шины) |
| *Максимальный режим* | |
| QS0, QS1 | Состояние очереди команд |
| S0 .. S2 | Тип цикла обмена, указывает на одну из возможных ситуаций:  000 – признак INTA  001 – ввод информации с ВУ  010 – вывод данных на ВУ  011 – останов  100 – выборка команды  101 – чтение из памяти  110 – запись в память |
| LOСK | Сигнал блокировки, индицирующий, что другое устройство не может занять системную магистраль (запрет на захват системной шины) |
| RQ/GT0, RQ/GT1 | Запрос/разрешение доступа к шине |

***Шинные циклы К1810ВМ86***

В максимальном режиме, управляющие сигналы системной шины вырабатываются системным контроллером по сигналам состояния процессора. Все сигналы управления (кроме ALE) активны по низкому уровню, что дает возможность раздельно управлять шиной несколькими устройствами.

Для обращения к устройствам ввода/вывода процессор имеет отдельные инструкции IN и OUT, результатом выполнения которых является формирование шинных сигналов IORD и IOWR. В циклах ввода/вывода используют только младшие 16 бит шины адреса, что позволяет адресовать до 64кбайт регистров ввода/вывода. Адрес устройства задается либо в команде, либо берется из регистра DX.

Циклы обращения к портам отличаются от циклов памяти использованием шины адреса. При обращении к портам линии адреса А16..А19 всегда содержат 0, а линии А8..А15 содержат старший байт адреса, только при косвенной адресации через регистр DX. При обращении по непосредственному адресу линии А8..А15 содержат 0.

Цикл подтверждения прерывания аналогичен циклу чтения порта но вместо сигнала IORD, активен сигнал INTA, а состояние шины адреса процессором в это время не управляется.

***САМОСТОЯТЕЛЬНО***

**Организация систем ввода-вывода.**

*Ввод-вывод по прерываниям.*

*Программный ввод-вывод.*

*Прямой доступ памяти.*

*Контроллер прямого доступа памяти.*

**Прерывания.**

*Назначение и типы прерываний.*

*Механизм обработки программный прерываний.*

*Механизм обработки аппаратных прерываний.*

*Контроллер прерываний.*

**Память.**

*Сегментная организация памяти.*

*Виды памяти. ОЗУ. ПЗУ.*

*Типы ОЗУ: статическая память, динамическая память.*

*Типы ПЗУ.*

*Иерархия памяти.*

*КЭШ память.*

*Принципы действия КЭШ памяти.*

*Организация КЭШ памяти.*

*Распределение адресного пространства.*

*Новые виды памяти.*

**Построение вычислительных систем.**

*Организация аппарата программного автомата.*

*Способы повышения производительности ЭВМ.*

*Многомашинные и мультипроцессорные ВС.*

*Мультипрограммные ВС.*

*Тенденции развития ВС и средств ВТ.*

*Новые виды памяти.*

#### FeRAM

Ферроэлектрическая память -- Ferroelectric RAM (FeRAM), это энергонезависимый тип памяти, аналогичный Flash памяти, что означает возможность хранения данных без использования источников энергии. Чипы FeRAM имеют маленькую емкость, на уровне килобит, но производство 1 Мбит чипов FeRAM уже не за горами, этим занимается компания NEC.

#### DRDRAM

Технология памяти Rambus основной архитектурой для изготовления системной памяти персональных компьютеров, в результате чего с 1999 года начнется вытеснение с этого рынка памяти типа SDRAM. Зная возможности Intel, можно с большой долей уверенности сказать, что так оно и будет. Специалисты корпорация Intel опробовали различные технологии памяти типа DRAM, прежде чем остановить свой выбор на технологии Rambus. Intel лицензировала архитектуру RDRAM у компании Rambus, после чего обе фирмы начали совместные разработки по созданию нового типа памяти, получившего наименование [DRDRAM](D:DASHKALibrari%20ot%20DariaКомпьютерная%20графикаmainboardrambus.html) (Direct RDRAM). Кстати, компания Rambus выдала лицензии на свою технологию изготовления памяти семи крупнейшим производителям чипов DRAM и около 15 производителям контроллеров. Результатом выбора корпорацией Intel технологии Rambus может стать появление более быстрой и более совершенной памяти, которая будет применятся повсеместно.

#### DDR SDRAM

Новый тип памяти DDR SDRAM (Double Data Rate SDRAM), появился в следствии улучшений архитектуры SDRAM, поэтому другое название этого типа памяти - SDRAM II. Лидерство в разработке этого типа памяти принадлежит корпорации Samsung. В настоящее время многие крупные производители чипов памяти заявили о намерении продвигать эту архитектуру. Однако, в свете того, что Intel собирается продвигать другую архитектуру памяти - DRDRAM, будущее DDR SDRAM представляется туманным.

Память типа DDR SDRAM может передавать и принимать данные по восходящему и нисходящему уровню сигнала шины, в отличие от обычной памяти типа SDRAM, которая передает данные только по восходящему уровню сигнала. При этом команды и адреса в DDR SDRAM все равно передаются по верхнему фронту сигнала. Память типа DDR SDRAM имеет большую ширину полосы пропускания, но только в случае передачи длинных пакетов данных. Максимальная величина ширины полосы пропускания DDR SDRAM может достигать 1.6 Гб/сек при частоте шины 100MHz.

Осенью 1998 компания Fujitsu Microelectronics представит первые образцы модулей DIMM, созданных по технологии DDR SDRAM. Работает эта память на частоте 125 Мгц, с пропускной способностью около 200 миллионов операций в секунду, что примерно соответствует DRDRAM. При всех своих достоинствах, эта технология прекрасно работает с нынешними машинами, являясь эволюционным развитием DRAM, в отличие от совершенно новой технологии Rambus, для которой опять понадобятся новые чипсеты системных плат и т.д.

#### ESDRAM

Enhanced SDRAM (ESDRAM - улучшенная SDRAM) -- более быстрая версия SDRAM, сделанная в соответствии со стандартом JEDEC компанией Enhanced Memory Systems ([EMS](http://www.edram.com/)). С точки зрения времени доступа производительность ESDRAM в два раза выше по сравнению со стандартной SDRAM. В большинстве приложений ESDRAM, благодаря более быстрому времени доступа к массиву SDRAM и наличию кэша, обеспечивает даже большую производительность, чем DDR SDRAM.

Более высокая скорость работы ESDRAM достигается за счет дополнительных функций, которые используются в архитектуре этой памяти. ESDRAM имеет строку кэш-регистров (SRAM), в которых хранятся данные, к которым уже было обращение. Доступ к данным в строке кэша осуществляется быстрее, чем к ячейкам SDRAM, со скоростью 12 ns, т.к. не требуется обращаться к данным в строке через адрес в колонке. При этом скорость работы ячеек ESDRAM составляет 22 ns в отличие от стандартной скорости работы ячеек SDRAM, имеющей значения 50 - 60 ns.

При этом стоит заметить, что память ESDRAM полностью совместима со стандартной памятью JEDEC SDRAM на уровне компонентов и модулей, по количеству контактов и функциональности. Однако, чтобы использовать все преимущества этого типа памяти, необходимо использовать специальный контроллер (чипсет).

Увеличение производительности при использовании ESDRAM достигается за счет применения двухбанковой архитектуры, которая состоит из массива SDRAM и SRAM строчных регистров (кэш). Строчные регистры вместе с быстрым массивом SDRAM обеспечивают более быстрый доступ для чтения и записи данных по сравнению со стандартной SDRAM. ESDRAM может работать в режиме "упреждающего обращения" к массиву SDRAM, в результате следующий цикл записи или чтения может начаться в момент, когда выполнение текущего цикла не завершено. Возможность использовать такой режим напрямую зависит от центрального процессора, управляющего работой конвейера адресации.

С точки зрения применения в качестве системной (оперативной) памяти компьютера чипсет VCS-164 (Polaris) компании [VLSI Technology](http://www.vlsi.com/) поддерживает ESDRAM, правда, этот чипсет рассчитан для применения в системах на базе процессора Digital [Alpha](http://www.ixbt.com/cpu/alpha.html). ESDRAM полностью соответствует спецификации Intel [PC-100](http://www.ixbt.com/mainboard/pc100.html) SDRAM, и соответственно, совместима с чипсетами Intel 440BX и Via Technologies [MVP-3](http://www.ixbt.com/mainboard/mvp-3.html).

#### FCRAM

Fast Cycle Random Access Memory (FCRAM). Разработчик этого типа памяти - компания Fujitsu. В основу этого типа памяти легла принципиально иная концепция по сравнению с DRAM. Время выполнения цикла соответствует всего 20нс, т.е. в 3-4 раза меньше, чем у нынешних модулей DRAM. Это достигнуто благодаря двум принципиальным моментам. Во-первых, в отличие от современных чипов памяти, где сначала выясняется адрес строки (RAS), а потом, после некоторой задержки, адрес столбца (CAS), где находится нужная ячейка, в FCRAM мгновенно выясняются обе координаты. Во-вторых, существующая память типа DRAM имеет время выполнения цикла 70нс, из-за того, что после выполнения каждой операции над ячейкой должна пройти команда сброса. В FCRAM же встроена цепь автоматического сброса, благодаря чему возможна конвейерная обработка команд, где следующая команда начинает выполняться еще до окончания выполнения предыдущей. Итог - время выполнения цикла - 20нс. В результате мы получаем очень интересный гибрид. По скорости работы FCRAM более напоминает SRAM, а по объему - обычную память DRAM.

#### MRAM

MRAM - Magnetic random access memory. Разработчик - компания Toshiba. Уже есть пробный образец основной структуры чипа MRAM, воплощающего в себе новую технологию памяти, потенциально способную превзойти существующие типы DRAM и по скорости, и по объему, и по энергопотреблению. Уже этот тестовый образец демонстрирует выдающиеся скоростные качества - цикл чтения занимает всего 6 нс.

Согласно заявлению компании, технология хранения информации в чипе MRAM заключается в создании элемента с мелкими частицами платины и кобальта, находящихся между двумя магнитными слоями. Запись и чтение происходят путем изменения магнитной активности в контуре.