**1.1. АРХИТЕКТУРА ПЕРСОНАЛЬНОГО КОМПА**

# **1.1.1. Устройство персонального компа**

Архитектура компа - его логическая организация, структура и ресурсы, которые может использовать прогер. вначале опишем логическую организацию компа.

Комп должен выполнять такие функции:

− ввод;

− вывод;

− обработка;

− хранение инфы;

− управление.

Для этого он включает в себя ф-циональные эл-ты, способные выполнять вышеперечисленные задачи.

Чтобы инфа попала на комп, ее надо ввести - клавиатура, дисководы (включая CD, DVD приводы), стример, сканер, МКфон. Для подключения к сети – модем, сетевая карта.

Устройства вывода - монитор, принтер, плоттер.

Обработка инфы - МКпроцессоры, в первую очередь – центральный МКпроцессор (CPU – central processing unit). Кроме “центральныого” есть другие - несколько независимых МКпроцессоров, например – МКпроцессор видеокарты.

3 группы Устройства хранения:

1) энергонезависимая память; она связана с тем, что должна быть какая-то инфа хотя бы для старта компа. Большинство МКсхем прошивается еще на стадии изготовления материнской платы, и изменить их содержание невозможно. Это так называемая ROM – read-only memory, / ПЗУ – постоянное запоминающее устройство;

2) энергозависимая память – хранит инфу какое-то временя, после чего эта инфа удаляется. Такие МКсхемы служат не только для чтения, но и для записи. + есть возможность осуществлять прямой доступ к любому участку энергозависимой памяти, поэтому ее называют RAM – random access memory (то есть память с произвольным доступом). Это оперативная память и сверхоперативная – то есть кэш-память;

3) накопители - дисководы, диски и кассетные устройства. Инфа на этих устройствах может храниться достаточно долго и при отключенном питании, но у этих устройств есть недостаток – они в тысячи раз медленнее, чем предыдущие два вида. Например, чтобы считать инфу с кассеты стримера, надо долго мотать эту кассету, то есть доступ последовательный. Но есть преимущество – хранимые объемы инфы в тысячи раз больше параметров предыдущих устройств.

Устройства управления – это Обработка инфы, ее ввод и вывод. Это узлы МКпроцессора.

# **1.1.2. Структурная схема IBM PC.**

Облик компа создавали три корпорации: Intel как производитель материнских плат и МКпроцессоров, Microsoft как корпорация, создающая ПО, и IBM как разработчик и изготовитель компов, внешних устройств и ПО.

IBM PC совместимый комп имеет такие **характеристики**:

1) МКпроцессор, программно совместимый с МКпроцессором Intel; то есть любая прога, написанная для данного МКпроцессора, может быть перенесена на МКпроцессоре Intel без каких-либо модификаций;

2) специфическая система распределения адресов оперативной памяти;

3) стандартизированное положение портов ввода-вывода (порт – это точка подключения внешнего устройства к внутренней шине МКпроцессора, прога может посылать данные в порты / получать их из портов; шина – это совокупность электрических соединений, обеспечивающих передачу данных и управляющих сигналов между компонентами компа);

4) традиционная система аппаратных прерываний (прерывание – это прекращение выполнения текущей команды для обработки некоторого события, после чего идёт возврат к прерванной проге; событие может быть вызвано командой / сигналом от внешнего устройства, аппаратное прерывание – то, которое вызвано внешним устройством);

5) система DMA (direct memory access) – система прямого доступа к памяти, минуя МКпроцессор;

6) набор системных устройств ввода-вывода;

7) унифицированные шины расширений (expansion bus) – шины для подключения дополнительных устройств;

8) BIOS (Basic Input Output System) – базовая система ввода-вывода, производящая начальную загрузку ОС и имеющая ограниченный набор функций по обслуживанию устройств ввода-вывода. Структуру IBM PC отображает схема на рис. 1.1.

Центральный МКпроцессор

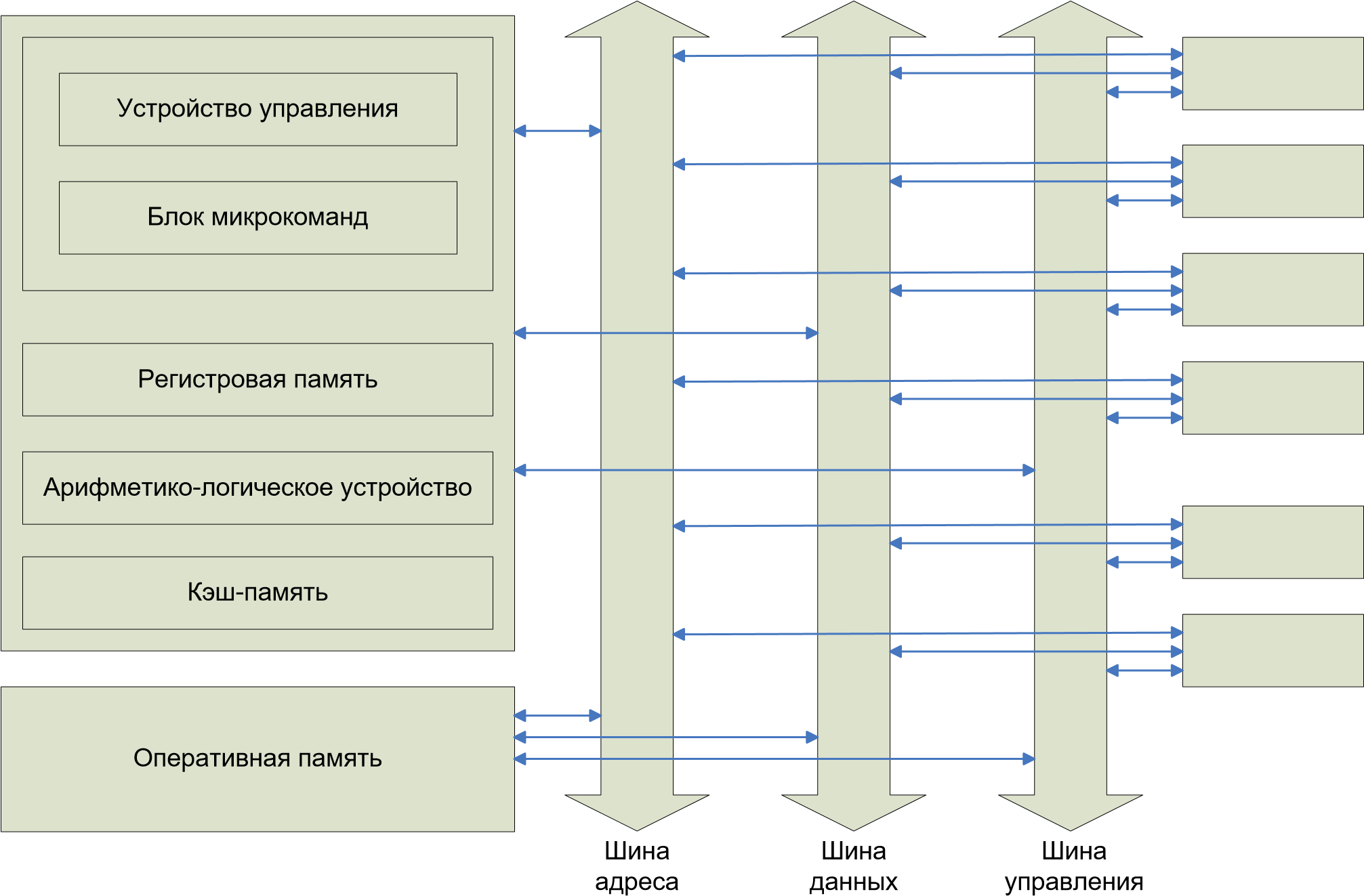


Рис. 1.1. Структурная схема компа.

Прога, которая должна быть выполнена, хранится в виде машинных **кодов** в **оперативной памяти**. Физически оперативная память - это набор ячеек, которые хранят либо высокое, либо низкое напряжение – 1 либо 0. Такая ячейка называется бит. Но работать с отдельными битами неудобно, поэтому есть более крупная единица инфы – байт, имеет 8 битов. Все команды и данные в компе закодированы нулями и единицами. В оперативной памяти каждый байт имеет свой адрес и адресовать отдельные биты невозможно.

В МКпроцессоре есть **внутренняя** то есть **регистровая** **память**. 1 / несколько регистров используют для адресации команд. Например, по значениям регистров вычислился адрес, по этому адресу из оперативной памяти выбирается очередная команда.

Затем эта команда передается в **устройство управления** и дешифрируется, то есть определяется, какая операция должна быть выполнена. По коду команды в устройстве управления определяют операнды, и если они есть у данной команды, то они также выбираются из оперативной памяти. После этого значения **регистров**, адресующих команды, изменяются и заданная операция выполняется. Такие действия идут непрерывно и циклически. Алгоритм немного меняется, если применяют команды переходов, в том числе и циклы.

По **шине адреса** устанавливается адрес нужной ячейки памяти / устройства, с которым будет идти обмен инфой. **Шину** **данных** применяют для передачи данных. А **шина управления** управляет этими процессами (например, один из сигналов на шине управления различает адрес памяти и адрес устройств ввода-вывода). Пример. 1) МКпроцессор читает содержание ячейки памяти. 2) Убедившись, что шина свободна, он ложит на шину адреса нужный адрес и устанавливает служебную инфу на шину управления: операция – чтение, устройство – оперативная память. 4) Теперь он ждет ответ от оперативной памяти. 5) Оперативная память увидела на шине к ней запрос на чтение инфы, и извлекает содержимое нужной ячейки и ложит его на шину данных.

**Кэш-память** согласует скорости работы медленных устройств (например, оперативная память) и быстрого МКпроцессора, чтобы избежать циклы ожидания в работе МКпроцессора, которые снижают производительность всей системы. Также, кэш-память согласует работу внешних устройств, например, накопителей и МКпроцессора.1 из контролеров кэш-памяти следит, чтобы команды и данные, которые нужны МКпроцессору в определенный момент, именно к этому моменту оказывались в кэш-памяти.

## 1.1.3. Логическая организация памяти.

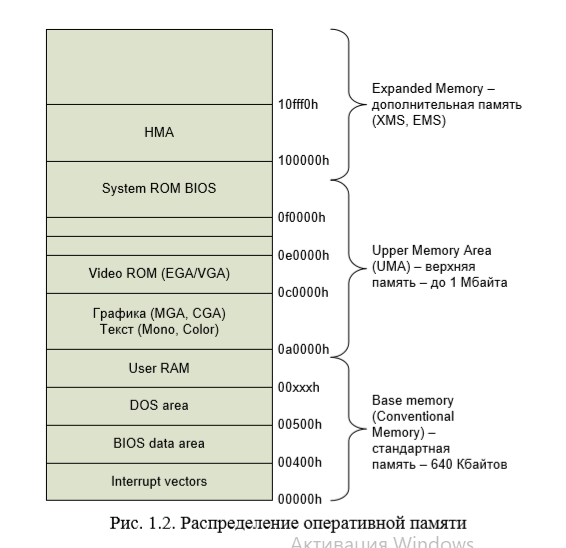
Первые МКпроцессоры в IBM РС - Intel 8086 и Intel 8088. Различия между ними - 808**6** имеет 1**6** битную шину данных, а 8088 – 8 битную. Для прогера разницы нет, но в цене **8088** дешевле. МКпроцессор Intel 8086 имеет 20 разрядную шину адреса, которая предоставляет доступ к одному мегабайту оперативной памяти:

220 байт = 1024 × 1024 байта = 1024 Кбайта = 1 Мбайт

Схема распределения памяти - на рис. 1.2, где числа представлены в 16 ричной системе счисления. В Ассемблере признак 16 ричной константы - это добавление h в конце числа.

Первые 640 Кбайт адресуемого пространства называют стандартной памятью (**conventional** memory). Остальные 384 Кбайта зарезервированы для системного использования и называются памятью в верхних адресах (**UMB** – upper memory blocks, High DOS Memory / upper memory area – **UMA**). Эта область памяти резервируется под размещение системной ROM BIOS, ROM-память дополнительных адаптеров и видеопамять.

Дополнительная (expanded) память – это область оперативной памяти, лежащая за пределами 1 Мбайта. Напрямую обратиться к этой памяти нельзя (шина адреса –20 битная), поэтому есть спецификация дополнительной памяти – EMS (expanded memory specification). Принцип её использования основан на переключении блоков (то есть страниц) памяти. Так как на всех компах область памяти **UMA** не заполнена полностью, то в ней создается “окно”. Прикладные проги с помощью спец драйверов загружаются в область дополнительной памяти, и затем с помощью тех же драйверов перемещаются на выполнение в “окно”. Таким образом, хотя МКпроцессор всегда обращается к данным, хранимым в “окне”, адреса этих данных можно смещать в дополнительной памяти относительно “окна” на несколько мегабайт. В компах на МКпроцессоре Intel 8086 для реализации дополнительной памяти применяются спец платы с аппаратной поддержкой “подкачки” блоков (то есть страниц) памяти и соответствующий программный драйвер.



Компы с МКпроцессором 80286 с 24 разрядной шиной адреса, адресуют 16 Мбайт памяти, а МКпроцессор с 32 разрядными шинами адреса – 4 Гбайта памяти.

## 1.1.4. Пространство портов ввода-вывода.

Все устройства на системной шине (это 3 шины – адреса, данных и управления) МКпроцессора - это либо адресуемая память, либо порты ввода-вывода. Порт - это схема, которая включает в себя 1 / несколько **регистров** ввода-вывода (то есть особых ячеек памяти). Теоретически МКпроцессор может управлять до 65 536 портами, но используют меньше. Каждый порт имеет свой номер; для дешифрации адресов портов используют только младшие 10 битов, что позволяет обращаться к портам в диапазоне адресов **000h – 3ffh**. Обмен инфой между МКпроцессором и портами идёт командами:

**in** (считать данные из порта),

out (записать данные в порт) и **ins** (получить эл-т строки из порта),

**outs** (записать эл-т строки в порт).

В таблице 1.1 - стандартные порты ввода-вывода.

# Таблица 1.1. Стандартные порты ввода-вывода

|  |  |
| --- | --- |
| **Адрес** | **Назначение** |
| 000h – 00fh | контроллер DMA #1 |
| 020h – 021h | контроллер прерываний #1 |
| 040h – 05fh | таймер |
| 060h | контроллер клавиатуры |
| 061h | источник NMI и управление звуком |
| 064h | контроллер клавиатуры |
| 070h – 07fh | память CMOS и маска NMI |
| 080h | диагностический регистр |
| 080h – 08fh | регистры страниц DMA |
| 0a0h – 0bfh | контроллер прерываний #2 |
| 0c0h – 0dfh | контроллер DMA #2 |
| 0f0h – 0ffh | сопроцессор 80287 |
| 170h – 177h | контроллер IDE #2 |
| 1f0h – 1f7h | контроллер IDE #1 |
| 200h – 207h | игровой адаптер |
| 238h – 23fh | COM4 |
| 278h – 27fh | параллельный порт LPT2 |
| 2c0h – 2dfh | EGA #2 |
| 2e0h – 2e7h | COM4 |
| 2e8h – 2efh | COM4 |
| 2f8h – 2ffh | COM2 |
| 300h – 31fh | плата прототипа |
| 338h – 33fh | COM3 |
| 370h – 377h | контроллер IDE #2 |
| 376h – 377h | порты команд IDE #2 |
| 378h – 37fh | параллельный порт LPT1 |
| 380h – 38fh | синхронный адаптер #2 |
| 3a0h – 3afh | синхронный адаптер #1 |
| 3b0h – 3bbh | монохромный адаптер MDA |
| 3bch – 3bfh | параллельный порт LPT1 платы MDA |
| 3c0h – 3cfh | EGA #1 |
| 3c0h – 3dfh | VGA |
| 3d0h – 3dfh | CGA/EGA |
| 3e0h – 3e7h | COM3 |
| 3e8h – 3efh | COM3 |
| 3f0h – 3f7h | контроллер IDE #1 |
| 3f6h – 3f7h | порты команд IDE #1 |
| 3f8h – 3ffh | COM1 |

## 1.1.5. Прерывания (аппаратные и программные).

Прерывание – это сигнал МКпроцессору, что в системе произошло событие. Тогда выполнение текущей последовательности команд останавливается и начинает выполняться подпрога, соответствующая данному прерыванию. Есть аппаратные и программные прерывания.

Аппаратные прерывания (IRQ – interrupt request) передаются по спец линиям системной шины, это запросы от внешних устройств (например, нажатие клавиши на клавиатуре). Программные прерывания - для вызова спец подпрограмм и для их использования в МКпроцессоре есть спец команды (например, по команде **int 21h** вызывается прерывание ОС DOS).

МКпроцессор в одно и то же время может обслужить только одно событие. **Контроллер прерываний** нужен для выбора этого события, он устанавливает для каждого из своих входов определенный уровень значимости – то есть приоритет. Наивысший приоритет имеет линия запроса прерывания **IRQ0**, а наименьший - **IRQ7** (таблица 1.2). Их количество увеличено до 15 – то есть увеличено число входов контроллера. И теперь приоритеты прерываний распределись так: линии прерывания IRQ8 – IRQ15 имеют приоритет ниже, чем IRQ1, но выше IRQ3.

# Таблица 1.2. Аппаратные прерывания **в порядке уменьшения приоритета**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя | | Вектор | Описание | |
| IRQ0 | | 08h | таймер | |
| IRQ1 | | 09h | клавиатура | |
| IRQ2 | | 0ah | недоступно | |
| IRQ8 | | 70h | CMOS RTC – часы реального времени | |
| IRQ9 | | 71h | резерв | |
| IRQ10 | | 72h | резерв | |
| IRQ11 | | 73h | резерв | |
| IRQ12 | | 74h | PS/2 мышь | |
| IRQ13 | | 75h | математический сопроцессор | |
| IRQ14 | | 76h | HDC – контроллер жестких дисков | |
| IRQ15 | | 77h | резерв | |
| IRQ3 | | 0bh | COM2, COM4 | |
| IRQ4 | | 0ch | COM1, COM3 | |
| IRQ5 | | 0dh | LPT2, звук | |
| IRQ6 | | 0eh | FDC – контроллер гибких дисков | |
| Имя | Вектор | Описание |
| IRQ7 | 0fh | LPT1 – принтер |

## 1.1.6. Прямой доступ к памяти. DMA.

Режим прямого доступа к памяти (DMA – direct memory access) – это когда периферийное устройство (флкшка) связано с оперативной памятью напрямую, а не через внутренние регистры МКпроцессора. Это эффективно, когда нужна высокая скорость обмена для большого количества инфы.

Для начала процесса прямого доступа на системной шине используют нужные сигналы. В IBM РС для этого используют спец МКсхемы.

---

Часть 2.

**1.2. РЕГИСТРОВАЯ АРХИТЕКТУРА МКПРОЦЕССОРА PENTIUM.**

# **1.2.1. Общие сведения.**

Начиная с МКпроцессора 80386 и до пятого поколения процессоров Intel вводят 32 разрядные регистры. Эти МКпроцессоры имеют 32 битную шину адреса, что позволяет адресовать 4 Гбайта оперативной памяти, а при помощи виртуальной памяти – до 64 Тбайт инфы.

Виртуальная память – это работа с объемом памяти больше, чем физически доступная память в компе.

Но для совместимости с ПО, которое написано для процессоров с 16 битными регистрами, осталась возможность использовать регистры длиной 16 битов.

2 режима работы МКпроцессоры Pentium:

1) реальный режим – **Real** Address Mode – можно использовать только первый мегабайт оперативной памяти; здесь выполняемая прога монопольно использует все ресурсы компа;

2) Защищенный режим – **Protected** Virtual Address Mode – адресация до

4 Гбайт физической памяти и механизм защиты (то есть МКпроцессор может разрешать либо нет выполнять какое-то действие, например, доступ к определенному участку оперативной памяти).

**АЛУ** (то есть арифметико-логическое устройство) - выполняет целочисленные операции, используя операнды длиной 8, 16 и 32 бита. Для работы с числами с плавающей точкой используется пристройка к процессору – это сопроцессор операций с плавающей точкой (FPU – floating-point processor).

Регистры – это ячейки памяти МКпроцессора. К некоторым из них можно получить доступ из проги, некоторые невидимы для прогера. МКпроцессор Pentium имеет 32 регистра, которые делится на 2 группы:

1. **пользовательские регистры**. Это:

1.1. регистры общего назначения; (1.2.2.)

1.2. указатель инструкций (команд);

1.3. указатель флагов;

1.4. регистры сегментов;

1. **системные регистры**. Это:

2.1. адресные регистры;

2.2. управляющие регистры;

2.3. регистры отладки;

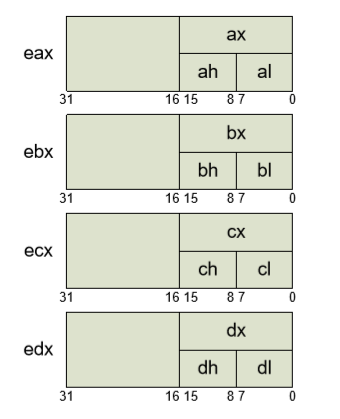
2.4. регистры тестирования;

2.5. модельно-специфические регистры.

Рассмотрим **пользовательские регистры**.

**1.2.2. Регистры общего назначения (eax, ebx, ecx, edx).**

Регистры eax, ebx, ecx, edx имеют такую структуру:



Любой из этих регистров содержит данные размером 1 байт (al – low, ah – high), 2 байта (ax) и 4 байта (eax – extended). Заметь: расширенная (extended) часть регистра не может использоваться отдельно.

Стандартные назначения этих 4х регистров:

1) eax/ax/ah/al (**accumulator** register) – это **аккумулятор**. Для хранения промежуточных данных. В некоторых командах его использование обязательно, например, в командах умножения MUL и IMUL;

2) ebx/bx/bh/bl (**base** register) – это **базовый** регистр. lля хранения базового адреса некоторого объекта в памяти. Например, для косвенной адресации данных в команде пересылки MUV;

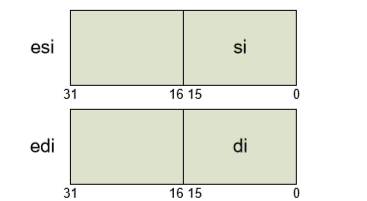
3) ecx/cx/ch/cl (count register) – это регистр-счетчик. Для команд, производящих повторяющиеся действия. Его использование неявно и скрыто в алгоритме работы команды. Например, команда организации цикла **loop** кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра ecx/cx;

4) edx/dx/dh/dl (data register) – это регистр **данных**. Так же, как и регистр eax/ax/ah/al, он хранит промежуточные данные. Иногда его использование обязательно (например, в командах деления DIV и IDIV).

эти регистры не обязательно должны использоваться по стандартному назначению – поэтому их называют регистрами общего назначения.

## 1.2.3. Регистры индексов (esi, edi) и указателей (ebp, esp).

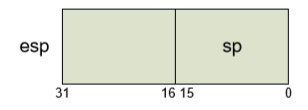
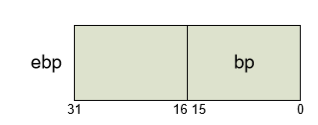
2 регистра для реализации строковых / цепочечных операций – то есть операций, производящих обработку последовательностей однотипных эл-тов. Регистры имеют структуру:



− esi/si (source index register) – это индекс источника. Этот регистр в цепочечных операциях имеет текущий адрес эл-та в цепочке-источнике;

− edi/di (destination index register) – это индекс приемника (получателя). Этот регистр в цепочечных операциях имеет текущий адрес в цепочке-приемнике.

Cтек – это особая структура данных. Это длинный цилиндр, его структура типа LIFO. Для работ со стеком есть спец команды и спец регистры. Но в стеке компа есть возможность прямого доступа к содержимому стеку. Структура регистров – на рис:

− **esp/sp (stack pointer register**) – это регистр указателя стека. Имеет указатель вершины стека в текущем сегменте стека; вершина стека – это верхний единственный доступный эл-т;

− **ebp/bp (base pointer register)** – это регистр указателя базы. Для произвольного доступа к данным внутри стека.

Регистры указателей и индексов можно использовать не только стандартно, в них можно записывать любую инфу и использовать в работе с разными командами; но лучше этого не делать.

## 1.2.4. Сегментные регистры (cs, ss, ds, es, fs, gs).

МКпроцессор использует сегментную модель адресации оперативной памяти, то есть такую формулу для определения адреса:

**address = segment × 16 + offset,**

где **address** – это вычисляемый адрес,

**segment** – это сегмент / часть адреса,

**offset** – это смещение относительно данного сегмента.

Такая формула связана с тем, что шина данных первых IBM PC 20 разрядная, а регистры – только 16 битные; поэтому, напрямую указать 20 битный адрес нельзя. Сегмент – это область памяти, где любой байт можно адресовать только с помощью смещения. А так как смещение хранится в 16 битном регистре, то длина сегмента = 64 Кбайта.

Прога имеет несколько сегментов (в реальном режиме работы МКпроцессора), по их количеству и по характеру использования различают несколько моделей памяти. Например, для модели памяти **small** (ее структура - на рис. 1.3) выделяют 2 сегмента: 1 – под коды команды, второй – под данные и стек.

Код программы

Статические переменные

Куча

–

динамически

распределяемая память

near

-

heap

Стек

–

существует для

хранения вспомогательных

переменных

far

-

heap

младшие

адреса

старшие

адреса

сегмент кода

(

64

Кбайта

)

сегмент данных

и стека

(

64

Кбайта

)

Рис. 1.3. Расположение проги в оперативной памяти для модели **small**

Для поддержания памяти сегментной организации в реальном режиме МКпроцессор имеет 6 сегментных регистров, длина каждого – по 16 бит. Все сегментные регистры имеют жесткое закрепление:

− **cs (code segment)** – адрес начала сегмента кода; сегмент имеет команды, которые отправляются на выполнение МКпроцессору;

− **ds (data segment)** – адрес начала сегмента данных; сегмент имеет данные, которые обрабатывает прога;

− **ss (stack segment)** – адрес начала сегмента стека;

− **es, fs, gs** – адреса начал дополнительных сегментов данных; его используют если не хватает объема основного сегмента данных.

Также сегментные регистры используют для адресации в защищенном режиме работы МКпроцессора, но механизм формирования физического адреса отличен от механизма в реальном режиме.

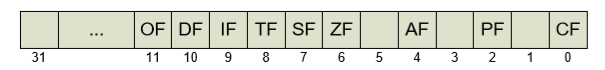
## 1.2.5. Регистр указателя инструкций (eip).

Регистр **eip** (extended instruction pointer) имеет разрядность 32 бита. К младшим двум байтам можно обратиться в отдельности – как к **ip**. Он имеет смещение относительно начала сегмента кода **cs**, указывающее на такую команду, посылаемую МКпроцессору. Прямого доступа к этому регистру **нет**, но он меняется при использовании в проге безусловных и условных переходов, а также при вызове процедур и прерываний и при возврате из них.

## 1.2.6. Регистр флагов (eflags).

В регистре флагов eflags регистрируется инфа (то есть признаки) о состоянии вычислительного процесса для принятия прогой решений. Длина регистра флагов – 4 байта.

Структура регистра eflags:



Как видно из рисунка, некоторые биты имеют свои обозначения:

− CF (carry flag) – это флаг **переноса**, принимает значение 1, если был перенос при сложении / заем при вычитании, иначе – 0;

− PF (parity flag) – флаг паритета, устанавливается в 1, если в младшем байте результата операции четное число единиц, иначе – 0;

− AF (auxiliary carry flag) – **вспомогательный** флаг переноса. 1 – если был перенос / заем с третьего в четвертый разряд младшего байта, применяется в арифметике двоично-десятичных чисел;

− ZF (zero flag) – флаг **нуля**. 1 – если результат равен нулю, иначе – 0;

− SF (sign flag) – признак знака. 1 – если результат отрицательный, иначе – 0;

− TF (trace flag) – признак трассировки. 1 - если МКпроцессор находится в пошаговом режиме, иначе – 0;

− IF (interrupt enable flag) – флаг прерывания. 1 - если МКпроцессору разрешено реагировать на прерывания от внешних устройств, если 0 – он их игнорирует;

− DF (direction flag) – флаг направления – управляет порядком обработки операнда; используется в строковых командах;

− OF (overflow flag) – флаг знакового переполнения.

---

Часть 3.

**2.1. СТРУКТУРА ПРОГИ НА ЯЗЫКЕ АССЕМБЛЕР.**

# **2.1.1. Первичные эл-ты языка Ассемблер.**

Идентификатор - это набор латинских букв (больших и малых), цифр и спец символов

\_ ? $ @

Идентификатор не может начинаться с цифры.

Идентификаторы обозначают такие эл-ты проги:

− переменные;

− метки; и

− символьные имена.

Переменные определяют хранящиеся в памяти **данные** и имеют 3 атрибута:

1) **сегмент**, соответствующий сегменту, где лежит эта переменная;

2) **смещение** относительно начала сегмента; и

3) **тип**, определяющий число байт под данную переменную.

Метка. Определяет **адрес** в памяти младшего байта команды. Метки используют для организации переходов. То есть если в адресе перехода - метка, то указан адрес команды, на которую пойдет переход. Метки имеют 2 атрибута: 1) сегмент; и 2) смещение.

По умолчанию транслятор не отличает малые буквы от больших.

Оператором может быть команда, псевдооператор / директива и макрокоманда. На этапе трансляции команды преобразуются в инструкции МКпроцессора (то есть машинные команды); а директивы несут указания самому транслятору.

Команда Ассемблера может иметь до четырех полей и быть на одной строке:

**[метка : ] мнемокод [операнд] [; комментарий]**

Квадратные скобки говорят, что эл-т команды в скобках, необязателен. Пример команды:

**pov: mov ax, 5 ; занесение в регистр ax значения 5**

; pov – это метка, mov – это мнемокод операции (то есть команда)

; ax и 5 – операнды, все, что идет после символа «;» – комментарий

Директивы имеют такой формат:

**[идентификатор] директива [операнды / параметры] [; комментарий]**

Пример директивы определения имени:

**reg equ ax** ; везде в тексте проги имя **reg** будет заменено на **ax**, а **ax** на этапе

компиляции будет пониматься как регистр.

# **2.1.2. Сегменты в Ассемблере.**

Любая прога на Ассемблере состоит из сегментов.

Сегмент – это логически выделенная область в исходном тексте проги для определенных целей. Сегмент кода имеет команды, сегмент данных имеет данные, в сегменте стека отводится место под стек. Сегмент языка Ассемблер и физический сегмент памяти – это разные вещи: прогер управляет сегментами Ассемблера с помощью спец директив и параметров, а прога-компоновщик (то есть **линкер**) определяет, как расположить в памяти сегменты Ассемблера, какую между ними установить связь и как их адресовать.

МКпроцессор все **текущие сегменты** делит на 3 вида, расположенные в памяти. Все 3 образуют **программный сегмент**:

1. текущий сегмент кода, из которого по смещению, заданному в **eip**, считывается очередная машинная команда;
2. данных – для доступа к операндам команд; смещение внутри этого сегмента задается в самой машинной команде;
3. стека, используемый для доступа к стеку; смещение задается в регистрах esp / ebp.

# **2.1.3. Структура com-проги.**

В исполняемых файлах com-формата для адресации памяти используют 16 битные регистры. То есть физический адрес ячейки памяти находится по формуле

**address = segment 16 + offset**

где segment и offset – это 16 битные регистры.

**Программный сегмент** – это блок памяти, выделяемый для размещения кода, данных и стека проги. Перед программным сегментом есть блок памяти – префикс программного сегмента (PSP – program segment prefix), его длина 256 байт. В PSP есть разная инфа для ОС (операционной системы) и самой выполняющейся проги (например, с адреса 081h до 0ffh записана часть командной строки, такая за именем проги – параметры командной строки). После загрузки com-прога занимает 1 физический сегмент памяти (= 64 Кбайта).

Com-файл готов к выполнению без настроек адресов; единственное, что делается операционной системой – это настройка некоторых регистров. Сегментные регистры установлены одинаково; они имеют адрес начала физического сегмента. Регистр указателя инструкций **ip=100 h** (первые 256 байт отведены под PSP), а регистр указателя вершины стека **sp** = 0 f f f e h. Соответствующая схема - на рис. 2.1.

PSP

Данные и код

Стек

CS

,

DS

,

ES

,

SS

0

h

0

ffh

0100

h

0

fffeh

0

ffffh

SP

=

0

fffeh

IP

=

0100

h

6

4

К

б

а

й

т

а

Рис. 2.1. Настройка регистров после загрузки com-проги

# **2.1.4. Структура exe-проги.**

Прога в exe-формате позволяет вводить неограниченное количество адресуемых раздельно областей кода и данных (стек 1, и больше не надо). Такие области называют логическими сегментами. Инфа о логических сегментах есть в заголовке exe-файла.

По данным из заголовка exe-файла ОС настраивает стек (то есть регистры ss и esp) и запускает прогу (то есть устанавливает cs и eip).

Регистры ds и es указывают на начало PSP. В проге надо предусмотреть запись в регистры ds и es нужных адресов для организации доступа к логическим сегментам данных. Смена текущего сегмента кода выполняется за счет дальних переходов. Только так можно изменить cs: eip.

Пример на рис. 2.2. - прога с двумя логическими сегментами данных (d1, d2), двумя логическими сегментами кода (c1, c2) и логическим сегментом стека.

PSP

Сегмент данных

d

1

Сегмент данных

d

2

Стек

Сегмент кода

c

1

begin

:

Сегмент кода

c

2

DS

,

ES

0100

h

SS

CS

esp

eip

Точка входа в программу

;

после загрузки управление

передается на эту метку

(

устанавливаются cs и eip

)

Рис 2.2. Пример exe-проги

---

Часть 4.

**2.2. ДИРЕКТИВЫ СЕГМЕНТАЦИИ.**

**2.2.1. Стандартные директивы сегментации.**

Структура сегмента на Ассемблере имеет вид:

**имя segment [тип вырав.] [тип комбин.] [класс] [тип размера]** команды и директивы

имя ends

Начало логического сегмента определяет директива **segment**, а конец его – директива **ends**. Данные директивы должны иметь одно и то же **имя**, представляющее собой идентификатор.

Параметры директивы **segment**:

А) Атрибут **выравнивания** сегмента (тип выравнивания) сообщает компоновщику (то есть редактору связей), что надо обеспечить размещение начала сегмента на заданной границе, так как в зависимости от выравнивания с разной скоростью идет доступ к оперативной памяти. Допустимые значения этого атрибута такие:

− byte – выравнивание не выполняется; сегмент может начинаться с любого адреса памяти.

− word – сегмент начинается по адресу, кратному 2; выравнивание на границу слова.

− dword – сегмент начинается по адресу, кратному 4; выравнивание на границу двойного слова.

− para – сегмент начинается по адресу, кратному 16; выравнивание на границу параграфа (размер параграфа – 16 байт).

− page – сегмент начинается по адресу, кратному 256; выравнивание на границу страницы (длина страницы – 256 байт).

− mempage – сегмент начинается по адресу, кратному 4 Кбайт; выравнивание на границу страницы памяти (длина страницы памяти – 4 Кбайта).

По умолчанию тип выравнивания имеет значение **para** – то есть сегмент загружается в память с адреса, кратного 16. Чем больше величина границы, на которую идёт выравнивание, тем быстрее идёт доступ к памяти.

Б) Атрибут **комбинирования** сегментов (то есть тип комбинирования) говорит редактору связей как надо объединять сегменты, имеющие одно и то же имя (это имеет смысл при модульном программировании – когда прога состоит из нескольких частей). Сегменты с разными именами объединять нельзя.

5 вариантов значений атрибута комбинирования:

1)private – сегмент не будет объединяться с другими сегментами.

2) public – заставляет компоновщик соединить все сегменты с одинаковыми именами. Новый объединенный сегмент будет целым и непрерывным. Все адреса (то есть смещения) объектов будут вычисляться относительно начала этого нового сегмента.

3) common – располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученной в результате области загрузки будет равен размеру самого большого сегмента.

4) **at xxxx** – располагает сегмент по абсолютному адресу параграфа, который задается выражением xxxx. Компоновщик располагает сегмент по заданному адресу памяти, учитывая атрибут комбинирования(например, для доступа к видеопамяти). Физически –это значит, что сегмент при загрузке в память будет расположен, начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в таком сегменте, отсчитывают относительно заданного адреса.

5) **stack** – определение сегмента стека. Он заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра **ss**. Комбинированный тип **stack** аналогичен комбинированному типу **public**, не считая того, что регистр **ss** - это стандартный сегментный регистр для сегментов стека. Регистр **sp** устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип **stack** не используется, прогер должен явно загрузить в регистр **ss** адрес сегмента (как это делается для регистра **ds**).

Неявно атрибут комбинирования принимает значение **private**.

В) Атрибут **класса** сегмента (то есть тип класса) – это заключенный в кавычки идентификатор (то есть имя класса). Сегменты с одним и тем же именем класса, будут располагаться в памяти друг за другом. Типичный пример использования атрибута класса - объединение в группу всех сегментов кода проги (для этого используется имя класса “code”).

Атрибут размера сегмента (режим адресации) может принимать 2 значения:

1. **use16** – для адресации инфы внутри сегмента используется 16 разрядное смещение; размер такого сегмента не может превышать 64 Кбайта (то есть реальный режим работы МКпроцессора).
2. use32 – при формировании адреса внутри сегмента используется 32 разрядное смещение; размер такого сегмента не может превышать 4 Гбайта (защищенный режим работы МКпроцессора).

Если атрибут размера сегмента не указан, то неявно он определяется транслятором по типу МКпроцессора. Тип МКпроцессора задает такая директива:

**.тип\_МКпроцессора**

Пример: **.486 ; режим работы МКпроцессора 80486**

Директива переключения типа МКпроцессора указывает транслятору, для какого МКпроцессора создавать код проги. Это надо для поддержки системы команд и аппаратных особенностей данного МКпроцессора.

Все сегменты равноправны, так как директивы **segment** и **ends** не имеют инфы о функциональном назначении сегментов. Чтобы использовать их как сегменты кода, данных / стека, надо заранее сообщить транслятору об этом, для этого используют спец директиву **assume**. Ее формат такой:

**assume сегментный\_регистр : имя\_сегмента[,сегментный\_регистр :**

**имя\_сегмента…]**

Эта директива сообщает транслятору какой сегмент к какому сегментному регистру привязан; то есть, какой сегментный регистр будет использоваться для адресации внутри данного сегмента при выполнении проги.

Директива assume не загружает сегментные адреса в нужные сегментные регистры. Регистр **cs** загружается автоматически, а прогер делает загрузку данных в остальные сегментные регистры.

Последний оператор проги - директива **end** и имеет формат:

**end метка**

Здесь метка определяет команду, которая должна быть выполнена первой после загрузки проги в память.

То есть метка определяет точку входа в прогу / стартовый адрес.

Пример структуры проги на Ассемблере:

d1 segment ; объявление сегмента d1

… ; определение данных проги

d1 ends ; конец сегмента d1

st1 segment stack ; объявление сегмента st1, тип комбинирования – stack

… ; отводим место под стек

st1 ends ; конец сегмента st1

c1 segment para public “code” ; объявление сегмента c1, тип выравнивания –

; на границу параграфа, тип комбинирования –

; public, имя класса – code

assume cs:code,ds:d1,ss:st1 ; привязывание сегментных регистров

; к логическим сегментам

begin: ; точка входа в прогу

… ; команды Ассемблера

c1 ends ; конец сегмента c1

|  |  |
| --- | --- |
| c2 segment “code” | ; сегмент c2, имя класса – code, в памяти |
| assume cs:c2 | ; будет расположен после сегмента c1 |
| … | ; команды Ассемблера |
| c2 ends | ; конец сегмента |
| end begin | ; конец проги, установка стартовой метки |

В общем случае количество логических сегментов не ограничено. Их имена произвольные. Директив assume может быть несколько, а метка, определяющая точку входа в прогу, может быть в любом месте внутри сегмента. Директива end стоит в самом конце проги, и других директив end быть не может.

Пример простейшей проги со стандартными директивами сегментации:

data segment ; сегмент данных

message db ‘Hello, world!$’ ; определение строки текста,

; символ ‘$’ – признак конца печати при

;выводе на экран

data ends

stk segment stack ; сегмент стека

db 256 dup (?) ; отводим место под стек

stk ends

code segment ; сегмент кода

main: ; Метка начала проги

assume cs:code,ds:data,ss:stk

mov ax,data ; заносим в регистр ax сегментную компоненту

; адреса сегмента data

mov ds,ax ; копируем содержимое регистра ax в регистр ds, напрямую

; загрузить инфу в ds в данном случае нельзя– это следует

; из режимов адресации

lea dx,message ; загружаем в dx смещение переменной

; message, это делает команда lea

mov ah,09h ; заносим в ah число 9 – для прерывания 21h в этом регистре

; хранится номер функции, которую надо выполнить. девятая

; функция означает вывод строки, смещение, которой хранится в

; регистре dx

int 21h ; вызывается прерывание, идёт вывод строки текста

mov ah,4ch ; функция 4ch означает завершение работы проги и возврат в

; систему DOS

int 21h ; вызов прерывания с номером 21h

code ends

**end** main

Эта прога выводит строку **message** на экран. Команда **mov** делает пересылку байт из одной ячейки памяти в другую. По команде **int** вызывается подпрога, соответствующая прерыванию с данным номером; адрес этой подпроги вычисляют по значению выражения, которое является аргументом данной команды.

**2.2.2. Упрощенные директивы сегментации.**

Если прога простая (то есть в ней по одному сегменту данных, стека и кода), то ее написание упрощают с помощью упрощенных директив сегментации:

**.code [имя]** – начало / продолжение сегмента **кода**

**.data** – начало / продолжение сегмента **данных**

**.stack [размер]** – начало / продолжение сегмента **стека**

; необязательный параметр, задает размер стека в байтах.

При использовании упрощенных директив сегментации нельзя напрямую управлять размещением и комбинированием сегментов. Но можно управлять сегментами неявно: для этого вместе с упрощенными директивами применяют директиву указания модели памяти. Ее формат такой:

**.model операнд**

В операнде указывают **модель памяти**. Например, такие:

1) Модель памяти **small -**  показывает, что прога в памяти занимает 2 физических сегмента: 1 отводится для команд, второй – для данных и стека. Здесь команды адресуются только с помощью смещения относительно начала своего сегмента, и адресация данных и стека проги – тоже только используя смещение относительно общего сегмента данных и стека.

2) Модель памяти **tiny** - это использование только одного физического сегмента, относительно его начала адресуются и команды, и данные.

Вместе с размещением сегментов, директива **model** выполняет функции директивы **assume**, то есть ставит в соответствие сегментному регистру определенный сегмент проги. Поэтому директиву **assume** можно не применять при использовании упрощенных директив сегментации.

Пример проги вывода строки на экран:

.model small ; модель памяти small

.data ; сегмент данных

message db ‘Hello, world!$’

.stack 256 ; сегмент стек занимает 256 байт

.code ; сегмент кода

main:

mov ax, @data ; занесение адреса сегмента данных в регистр ax

mov ds, ax lea dx, message mov ah,09h int 21h mov ah,4ch int 21h

end main ;конец проги с точкой входа main

Физические адреса сегментов можно определить, используя спец идентификаторы. Для получения адреса сегмента данных используют **@data**, адреса сегмента кода – **@code**, адреса сегмента стека – **@stack**.

---

Часть 5.

**Тема 3.1. ФОРМАТЫ ДАННЫХ.**

# **3.1.1. Общие сведения.**

Вся инфа в компе - это нули и единицы, то есть наборы битов. Байт – это 8 бит, обрабатываемых как одно целое, отдельные биты вне байта не обрабатывают. Значит, команды и данные закодированы отдельными битами, но хранятся и обрабатываются в байтах. В МКпроцессоре есть команды, по-разному расшифровывающие инфу, записанную в байтах. Есть 4 формата представления данных. Это:

− двоичный формат;

− упакованный двоично-десятичный формат;

− неупакованный двоично-десятичный формат; и

− ASCII-формат. Разберем их подробнее.

# **3.1.2. Двоичный формат.**

В нём данные закодированы в двоичной системе счисления. Есть прямой, обратный и дополнительный коды целых чисел. Для перевода двоичного числа в прямом коде, в десятичное число надо умножить значение каждого разряда (0 либо 1) на 2 в степени, которая соответствует позиции этого разряда; причем нумерация позиций начинается с нуля. Пример:

10010111 = 1 × 20 + 1 × 21 + 1 × 22 + 0 × 23 + 1 × 24 + 0 × 25 + 0 × 26 + 1 × 27 = 151

прямой двоичный код десятичное число

Для перевода целого положительного десятичного числа в двоичное надо делить его на 2 и выписывать остаток, пока частное не станет = 1. Прямой двоичный код числа получим, выписывая справа налево остатки. Пример:

12

12

–

2

6

6

–

2

3

2

–

2

1

1

0

0

12

=

1100

десятичное число двоичное число

Одно и то же число, записанное в ячейку памяти (ячейка памяти – может быть 1, 2, 4 и так далее байта), можно определить и как знаковое, и как беззнаковое.

Суть введения **отрицательных** чисел в том, чтобы результаты арифметических операций были всегда корректны: и когда в ячейке хранится число со знаком, и когда работаем с числом без знака. Поэтому для знаковых чисел есть дополнительный код. В нём если старший бит ячейки инфы равен 0, то число положительное, и дополнительный код совпадает с прямым. (для байта – это седьмой бит, для двухбайтной ячейки – это пятнадцатый бит). Если же старший бит равен 1, то число - отрицательное, и дополнительный код отличается от прямого. Чтобы получить дополнительный код отрицательного числа, надо сначала взять модуль этого числа и все биты, кроме старшего, проинвертировать. И ещё к результату прибавить 1. Например, если в байт памяти надо положить значение -112:

112 01110000

–112 11110000 (прямой код; 1 в старшем бите – признак отрицат-го числа)

10001111 (инвертирование битов)

+ 00000001 (прибавление единицы)

10010000 = -112

дополнительный код числа -112

Допустим, в каком-то байте памяти с кодом 10010000 есть беззнаковое число, то это число равно +144 в десятичной системе счисления. Корректность арифметических операций в том, что если, например, к числу

-112 прибавить +112, то результат равен нулю:

1000 1111

+0000 0001

1001 0000 (в регистре флагов фиксируется беззноковое переполнение, флажок **CF = 1**. Но для знаковых чисел переполнения нет, и **OF = 0**)

Другой пример с **одно**-байтными ячейками памяти:

1100 1000 (беззнаковое число 200 и число со знаком -56)

+0001 0100 (число 20)

1101 1100 (результат – беззнаковое 220 и знаковое -36)

Пример с **двух**-байтной ячейкой памяти:

|  |  |
| --- | --- |
| 10000010 | 00000101 |

соответствует беззнаковому **33285** и

знаковому **-32251**

15 8 7 0

Для отличия **дво**ичные числа от **десяти**чных, в конце каждого **дво**ичного числа ставят букву b / B (бинари). Неявно любой набор цифр считают десятичным, но для явного указания этого ставят в конце букву d (D) (десымал). Для 8 ричных чисел в конце буква q (Q) (кварта), а для 16 ричных – h (H) (хватит).

Главное неудобство двоичной системы счисления – их размеры. На практике используют 16 ричную систему: цифры от 0 до 9 и буквы a, b, c, d, e, f. Номер позиции цифры в числе соответствует степени, в которую надо возвести число 16 (нумерация – с нуля). Например:

**96h = 9 × 16 + 6 = 150**

Для перевода из 16 ричной в двоичную систему и обратно, надо вместо каждой 16 ричной цифры подставить соответствующее 4 значное двоичное число:

**9h = 1001b, 6h = 0110b, 96h (хватит) = 10010110b (бинари)**

Если, например, в двух-байтную ячейку памяти надо поместить число

**1000 0010 0000 1101 b**, то в 16 ричной системе счисления это выглядит так:

|  |  |  |  |
| --- | --- | --- | --- |
| 8 | 2 | 0 | d |

15 8 7 0

Если число начинается с буквы, то в начале приписывают цифру 0, чтобы не попутать его с идентификатором.

**3.1.3. Двоично-десятичный формат (BCD формат).**

**Д**ля операций над целыми числами большой разрядности есть упакованный и неупакованный двоично-десятичный формат (BCD – binary-coded decimal).

|  |  |
| --- | --- |
| **0010** | **0101** |

В упакованном формате байт имеет две десятичные цифры в виде двоичных кодов. Число 25 здесь выглядит так:

70

В неупакованном формате каждая цифра занимает отдельный **байт**. Поэтому число 25 здесь занимает 2 байта:

|  |  |
| --- | --- |
| 0000**0010** | 0000**0101** |

15 8 7 0

# **3.1.4. ASCII формат.**

ASCII формат - для представления текстовой инфы. ASCII - American Standard Code for Information Interchange – американский стандартный код для обмена инфой. Например, код ASCII символа ‘W’ равен 87, код символа ‘#’ равен 35. Здесь каждый символ хранится в отдельном **байте**. Например, для символов ‘25’:

|  |  |
| --- | --- |
| 00110010 | 00110101 |

15 8 7 0

---

Часть 6.

**3.2. ДИРЕКТИВЫ ОПРЕДЕЛЕНИЯ ДАННЫХ.**

# **3.2.1. Общие сведения.**

Директивы определения данных служат для задания размеров и содержимого **данных** проги. При обработке директив определения данных в памяти генерируется **объектный код** – то есть отводится место под данные. Транслятор преобразует числа, строки и выражения, задаваемые в этих директивах, в отдельные образы байт, слов / других единиц данных. Эти образы копируются в **объектный файл.**

Директивы определения данных могут задавать:

1) скалярные данные - это единичное значение / набор единичных значений;

2) записи, позволяющие манипулировать с данными на уровне битов;

3) структуры, позволяющие обрабатывать совокупность данных разных типов как одно целое. Рассмотрим их подробнее.

**3.2.2. Директивы определения скалярных данных.**

Формат директив определения скалярных данных такой:

**[идентификатор] db значение [,значение…][ ;комментарий ]**

**[идентификатор] dw значение [,значение…][ ;комментарий ]**

**[идентификатор] dd значение [,значение…][ ;комментарий ]**

**[идентификатор] df значение [,значение…][ ;комментарий ]**

**[идентификатор] dq значение [,значение…][ ;комментарий ] [идентификатор] dt значение [,значение…][ ;комментарий ]**

Расшифровка мнемоники этих директив такая:

# Табл 3.1. Директивы определения скалярных данных.

Мнемоника Описание Байты

1) db **define** byte 1

2) dw **define** word 2

dd **define** doubleword 4

df **define** far pointer 6

dq **define** quadword 8

3) dt **define** ten bytes 10

1) Директива db обеспечивает распределение и инициализацию одного байта памяти. В качестве значения - целое число, строковая константа, оператор **dup**, константное выражение / оператор «?». Оператор «?» обозначает неопределенное значение, то есть отсутствие инициализации; оператор **dup** – дублирование значений. Если директива имеет имя, то создается переменная типа **byte**. Строковая константа может иметь столько символов, сколько ложится на одной строке. Символы строки хранятся в памяти в порядке их следования, то есть первый символ имеет самый младший адрес, последний – самый старший. Пример:

**b1 db -16**

**b2 db ‘abCDf’ b3 db ?**

2) Директива dw обеспечивает распределение и инициализацию слова памяти (2 байта). При использовании 16 битного режима адресации сегмента значением может быть адрес, таким образом, можно определить адрес физического сегмента и смещение определенного объекта в памяти. Пример:

**w1 dw 12, 12h, -2**

**w2 dw 4\*3**

подробнее:

**4\*3** – это выражение,

символ ‘**\***’ обозначает оператор **умножения**;

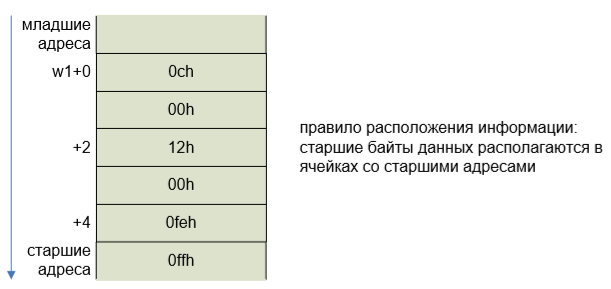
значения выражений вычисляют только на этапе транслирования, то есть значения выражений вычисляет транслятор; естественно, транслятор не может вычислить значение арифметического выражения, если в нем используется переменная, поэтому если надо выполнять арифметические и другие действия, в проге надо использовать

; нужные команды **w3 dw offset w1**

значение переменной w3 = смещению переменной w1,

**offset** – оператор, возвращающий смещение переменной / метки.

На рис. 3.1 показано, как будет располагаться в памяти переменная w1.



Пример:

**d1 dd 4294967295**

**d2 dd d1**

для 16 битного режима адресации в первое (младшее) слово памяти заносится смещение, а в следующее (старшее) слово – сегмент, в котором определена переменная d1.

3) Директива dt делает распределение и инициализацию 10 байт памяти. При обработке директивы dt значения с десятичными цифрами представляют собой не целые, а упакованные двоично-десятичные числа. Если надо определить десятибайтное целое число, то после числа надо указать спецификатор системы счисления (буква d для десятичных чисел, h для 16 ричных и т. д.). Пример:

**t1 dt 1234567890** ; в памяти это значение будет представлено в

упакованном двоично- десятичном формате.

**t2 dt 1208925819614629174706175d** ; десяти-байтное целое число.

Ниже приведены таблицы диапазонов знаковых и без-знаковых чисел в зависимости от количества байт, отводимых для представления числа.

# Таб 3.2. **Без-**знаковые целые числа (только положительные).

**Тип переменной Количество бит Диапазон**

|  |  |  |
| --- | --- | --- |
| byte | 8 | 0..255 |
| word | 16 | 0..65535 |
| dword | 32 | 0..4294967295 |
| qword | 64 | 0..18446744073709551615 |

Табл 3.3. Знаковые целые числа (старший бит определяет знак).

**Тип Количество Диапазон**

**переменной бит**

byte 7 -128 .. 127

word 15 -32768 .. 32767

dword 31 -2147483648 .. 2147483647

qword 63 -9223372036854775808 ... 9223372036854775807

Для определения массива целых чисел задают оператор дублирования **dup**. Он имеет такой формат:

**счетчик dup (значение1[, значение2, …])**

Список значений в скобках повторяется многократно в соответствии со значением счетчика. Каждое значение в скобках может быть целое число, символьная константа / другой оператор **dup** (допустимо до 17 уровней вложенности операторов **dup**). Примеры:

**val db 5 dup (4 dup (1,2,3))**

в результате под переменную **val** будет отведено 60 байт памяти. Массив val будет проинициализирован значениями ; 1,2,3,1,2,3,1,2,3 и так далее.

**mas1 dw 10 dup (0)**

задание одномерного массива слов из десяти эл-тов, эл-ты массива инициализируются нулями.

## 3.2.3. Определение BCD чисел.

Определение **не**упакованных BCD чисел (в одном байте – 1 десятичная цифра):

**a db 6, 5, 4, 3, 2, 1** ; определение числа 123456

01

02

04

03

05

06

старший байт младший байт

**mov ah, a+2** ; в ah будет помещено число 4

Определение **у**пакованных BCD чисел (в одном байте – 2 десятичные цифры):

**a db 56h, 34h, 12h**  ; определение числа 123456

12

34

56

старший байт младший байт

**mov ah, a+2** ; в ah будет помещено число 12h

## 3.2.4. Определение строк символов.

При определении строк символов одинарные ‘и двойные “” кавычки эквивалентны. Для определения строк символов использовать директиву **db**. Пример:

**str1 db “текст”**

**str2 db ‘текст’**

‘

т

’

‘

е

’

‘

с

’

‘

к

’

‘

т

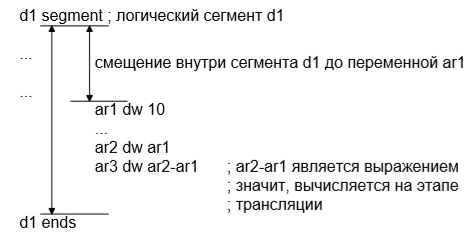
’

старший байт младший байт

**mov ah, str2+1** ; в ah будет помещен символ ‘е’

## 3.2.5. Определение адресов ячеек памяти.

Переменные используют не только для хранения данных, но и для хранения адресов ячеек памяти. На рис. 3.2 - пример определения адресов ячеек памяти для 16битного режима адресации.



# Рис. 3.2. Определение адресов ячеек памяти.

Переменная ar2 имеет значение смещения ar1; фактически, ar2 – это **указатель** на ar1. Переменная ar3 проинициализирована значением, равным разности смещений переменных **ar2 и ar1**, то есть разностью их адресов, а не разностью значений этих переменных.

3 команды, загружающие в **ax** одинаковые значения:

**mov ax, offset ar1** ; в **ax** загрузить смещение переменной ar1, offset – это

оператор, возвращающий смещение.

**mov ax, ar2** ; в **ax** загрузить содержимое **ar2**, то есть смещение переменной

**ar1 lea ax, ar1** ; в ax загрузить смещение ar1, команда lea – load effective

address – загрузить эффективный адрес (смещение).

---

Часть 7.

**3.3. ОПИСАНИЕ СИМВОЛИЧЕСКИХ ИМЕН.**

**3.3.1. Директива =.**

Директива = (знак равенства) имеет формат:

**имя = выражение**

По этой директиве создается имя, представляющее значение, равное текущему значению указанного выражения. Для его хранения память не выделяют. Вместо этого каждое вхождение этого имени в исходном файле замещают в момент транслирования значением выражения. Пример:

**a = 10**  ; определение символического имени **a**

**b = a + 5** ; имя b во время ассемблирования заменяется числом **15**

**a = 12** ; переопределение имени a, но значение b не меняется

Как показано в примере, имя может быть переопределено. В каждой директиве = именем может быть уникальный идентификатор / идентификатор, ранее использованный как имя другой директивой =.

Выражение может быть только числовым. Его значение не должно превышать величины 2 байта, а начиная с МКпроцессора 80386 –4 байта. Примеры:

|  |  |
| --- | --- |
| **a1 dw ?**  … |  |
| **MaxInt = 7fffh** | ; максимальное двухбайтное знаковое число допустимо |
| **str = ’AB’** | ; до двух символов, данные представлены в ASCII формате |
| **addr = offset a1** | ; создается символическое имя **addr**, которое по тексту замещается значением смещения переменной a1 (offset – это оператор, возвращающий смещение) |
| **.386** | ; директива переключения типа МКпроцессора, указывает  ; транслятору, для какого МКпроцессора создавать код  ; проги (это надо для поддержки системы команд и аппаратных  ; особенностей данного МКпроцессора), неявно транслятор подразумевает режим работы МКпроцессора 8086 MaxLong=7fffffffh максимальное 4 байтное знаковое число |

**3.3.2. Директива equ.**

Формат директивы equ: **имя equ операнд**

Директива **equ** создает имя, которое в момент трансляции будет заменено либо значением выражения, либо строкой текста. Значение выражения будет подставляться, если операндом будет числовое выражение, причем это значение должно помещаться в 2 / 4 байта. Во всех других случаях имя замещается строкой текста. Имя должно быть уникальным и его нельзя переопределить. Примеры:

**move equ mov** ; создается синоним для команды пересылки **mov**

**move ax, 5** ; в регистр **ax** заносится число 5

**count equ 20** ; символическое имя **count**, замещаемое в тексте числом 20

**move cx, count\*3** ; в регистр cx заносится число 60. на этапе транслирования

count заменяется на число 20, а 20\*3 – выражение, его значение вычисляет транслятор.

Директивы **equ** и **=** сходны по назначению, но различны по применению:

− директиву **equ** можно использовать не только с выражениями, значения которых можно подставить в работе, но и с текстовыми строками.

− идентификаторы, определенные директивой **equ** нельзя дальше переопределять по тексту, а с директивой = это можно.

---

Часть 8.

**4.1. ФОРМАТ МАШИННОЙ КОМАНДЫ.**

# **4.1.1. Общие сведения.**

**Машинная команда** – это закодированное указание МКпроцессору сделать определенную операцию. Машинная команда имеет инфу для определения:

− операции, которую надо выполнить;

− тип операндов, если операнды используются командой; и

− местоположение операндов; операнд может быть в самой команде / в регистре / в памяти.

Формат машинной команды МКпроцессора Pentium - на рис. 4.1.

prefix

cop

disp

data

0

–

5

1

,

2

0

,

1

0

,

1

0

,

1

,

2

,

4

0

,

1

,

2

,

4

однобайтные

префиксы

**код операции**

байт способа

адресации

байт

sib

смещение

непосред

-

ственный

операнд

mod

r

r

/

m

ss

index

base

Рис. 4.1. Формат машинной команды (над стрелками указаны возможные размеры полей в байтах)

Как видно, команда МКпроцессора состоит из нескольких частей, причем обязательное только поле кода операции. Использование остальных полей определяется выполняемым действием, типом и местоположением операндов. Подробнее о каждом поле команды:

1) Однобайтные префиксы нужны для модификации операции, которую выполняет команда; они отменяют действие, предусмотренное неявно МКпроцессором и указывают, как надо поступить в данном случае. Применение таких префиксов:

1) префикс переопределения сегмента: указывает явно, какой сегментный регистр надо использовать данной командой для адресации операнда **вместо** сегментного регистра (назначаемого неявно МКпроцессором).

2) префикс размера адреса: определяет явно, какая разрядность смещения, который используемого для адресации операнда данной командой(16 / 32 бита);

3) префикс размера операнда: явно указывает МКпроцессору размер операнда данной команды (16 / 32 бита).

4) префиксы повторения: используют со строковыми командами, позволяя организовывать быстрые циклы.

Пример: использование префикса переопределения сегмента:

**ar1 db ?** ; определение переменной ar1

…

**mov ar1, 10** ; неявно для адресации используют сегментный регистр **ds**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| c6 | 06 | 0e | 00 | 0a |

ст. мл.

**mov es:ar1, 10** ; явное указание сегментного регистра

c

6

06

0

e

00

0

a

ст

.

мл

.

26

префикс переопределения сегмента

После поля 1 байтных префиксов идет поле кода операции. Код операции определяет действие команды. Некоторые команды имеют несколько разных кодов. Размер поля кода операции – 1 / 2 байта. Если команда имеет операнды / хотя бы 1 операнд, то 2 младших бита кода операции могут быть индикаторами:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ... |  |  |  | d | w |

... 2 1 0

**w = 0**, если команда выполняет операцию над 1 байтными операндами;

**w = 1**, если размер операндов 2 / 4 байта, 2 байта – при 16 битном режиме адресации, а 4 байта – при 32 битном режиме адресации.

Бит **d** показывает, чем является регистр, указанный в поле **r** байта способа адресации:

**d = 0** – если регистр - это операнд-источник,

**d = 1** – если регистр - это операнд-приемник результата.

В некоторых командах вместо бита **d** указывают бит **s**, бит **s** определяет, надо ли расширить знаковый бит непосредственного операнда, указанного в команде. Пример:

**add bx, 5** ; прибавить к содержимому bx число 5 (=101 в 2ичной с-ме)

10000011

11000011

00000101

непосредственный операнд (число 5)

**w**

**s**

Бит **w = 1**, так как операция 2 байтная; бит **s = 1**, так как идёт расширение знакового бита непосредственного операнда (числа 5).

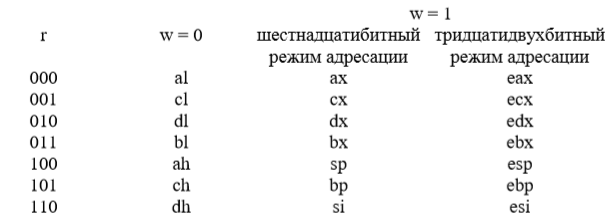
Байт способа адресации используется МКпроцессором для определения инфы об операндах: определяют конкретные регистры и правила нахождения адресов ячеек памяти.

2 старших бита называются полем **mod** (mode – режим); они задают режим адресации (например, режим “**регистр – регистр**”, “**память – непосредственный операнд**”). Неразрывно с полем **mod** используют 3 битное поле **r/m** (register/memory – регистр/память); в **r/m** определяют правила, по которым из памяти выбирается операнд / определяют используемый в качестве операнда регистр (при mod=11).

Например, при **r/m=110** и **mod=10** в 16 битном режиме адресации для получения адреса ячейки памяти, содержащей операнд, надо к содержимому регистра **bp** прибавить 16битное смещение, заданное в команде.

Поле **r** (**register**) применяют для задания регистра, используемого в команде. В табл 4.1 - соответствие значений поля **r** регистрам МКпроцессора.

Табл 4.1. Кодирование поля r, когда бит **w** есть в машинной команде.



Пример:

**ar1 dw 5** ; определение и инициализация переменной ar1.

…

**add dx, ar1** ; прибавление значения переменной ar1 к значению в регистре dx

; результат идет в dx

00000011

mod

010

r

/

m

disp

ст

.

мл

.

второй бит кода операции (бит d) соответствует регистру dx

равен 1, значит, регистр

является приемником

Байт **sib (scale, index, base** – масштаб, индекс, база) применяют вместе с байтом способа адресации для определения эффективного адреса операнда (то есть смещения относительно некоторого сегмента), который берут из памяти. Байт sib используют только в 32 битном режиме адресации и только, когда поле **r/m = 100** и поле **mod = 00,01 / 10** в байте способа адресации. В байте **sib** 3 младших бита (поле **base**) кодируют базовый регистр, 3 бита (поле **index**) – кодируют индексный регистр, и 2 старших бита определяют масштаб (поле **ss**). Общая формула определения эффективного адреса с использованием байта **sib** такая:

эффективный\_адрес **= базовый\_регистр + индексный\_регистр × масштаб+**

**+ смещение\_записанное\_в\_команде.**

Если байт способа адресации указывает, что для вычисления адреса операнда надо смещение, то это смещение кодируют в машинной команде. **Смещение** представляется как целое знаковое число. Размер поля **disp** (displacement – смещение) может составлять 0, 1, 2 / 4 байта.

Последнее поле машинной команды – это поле непосредственного операнда, позволяет напрямую сообщить значение этого операнда МКпроцессору. Непосредственным операндом может быть 1, 2 / 4 байта.

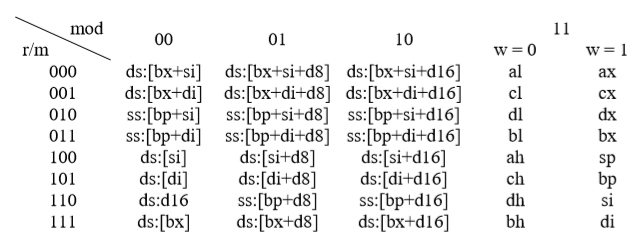
# **4.1.2. Формирование адреса операнда в 16 битном режиме адресации.**

Формат машинной команды в 16 битном режиме адресации никогда не имеет байта **sib**. Для определения адреса операнда используется только байт способа адресации. В табл 4.2 показано вычисление адреса операнда исходя из значений полей этого байта. Здесь d8 и d16 – соответственно 8 битное и 16 битное смещение, задаваемое в команде. Запись вида

**ds:[bx+si+d8]**

говорит, что операнд находится в ячейке памяти, для определения его физического адреса используется значение сегментного регистра **ds**, в котором находится адрес начала физического сегмента памяти и эффективный адрес, равный сумме значений в регистрах bx и si и смещения d8.

Таблица 4.2. Кодирование полей mod и r/m в 16битном режиме адресации



Примеры разных способов адресации операндов:

**ar1 dw ?** ; определение переменной ar1. Пусть эффективный адрес

относительно начала сегмента данных этой переменной = **000e h**

**ar2 db ?** ; определение переменной **ar2**. эффективный адрес этой переменной

равен **0010 h**

…

**add ar1, dx** ; суммирование содержимого переменной ar1 и значения в

регистре dx; результат заносится в переменную ar1.

00000001

00010110

00001110

00000000

код операции

байт способа

адресации

младший

байт

смещения

старший

байт

смещения

ст

.

мл

.

поля **w = 1** и **d = 0** означают, что операнды 2 байтные, а **регистр r** является источником; поля **mod = 00** и **r/m = 110** говорят, что для вычисления адреса операнда используют сегментный регистр **ds** и 2 байтное смещение (d16), задаваемое в команде. Поле **r = 010** соответствует регистру **dx.**

**add dh,ar2[bx][si]** ; к значению в регистре **dh** прибавляется значение, которое

есть в ячейке с эффективным адресом, равным смещение**\_ar2+bx+si**; результат остается в регистре.

00000010

10110000

00010000

00000000

код операции

байт способа

адресации

младший

байт

смещения

старший

байт

смещения

ст

.

мл

.

поля

w

=

0

и

d

=

1

соответствуют тому

,

что операнды однобайтные и регистр

является приемником результата; поле **r = 110** – используется регистр **dh**; **mod = 10, а r/m = 000** – значит, для определения эффективного адреса операнда к сумме значений регистров **bx** и **si** прибавляется 16 битное смещение, записанное в машинной команде.

**add cl, bl** ; к содержимому регистра cl прибавляется содержимое **al**, результат

остается в **cl**.

00000010

11001011

код операции

байт способа

адресации

ст. мл.

Операнды 1 байтные, результат остается в регистре (биты **dw = 10**); код в поле **r** соответствует регистру **cl**; режим адресации «регистр – регистр»

(**mod = 11**), **r/m = 011** – код регистра **bl**

**4.1.3. Формирование адреса операнда при 32 битном режиме адресации.**

При 32 битном режиме адресации для определения адреса операнда использовать 2 байта в машинной команде: 1 байт способа адресации и 1 байт **sib**. В таблице 4.3 рассмотрено как вычислить адрес операнда если нет **sib** (**d32** обозначает 32 битное смещение).

Табл 4.3. Кодирование полей mod и r/m в 32 битном режиме адресации.

mod 11

00 01 10

r/m w = 0 w = 1

000 ds:[eax] ds:[eax+d8] ds:[eax+d32] al eax

001 ds:[ecx] ds:[ecx+d8] ds:[ecx+d32] cl ecx

010 ds:[edx] ds:[edx+d8] ds:[edx+d32] dl edx

011 ds:[ebx] ds:[ebx+d8] ds:[ebx+d32] bl ebx

1. sib sib sib ah esp
2. ds:d32 ds:[ebp+d8] ds:[ebp+d32] ch ebp
3. ds:[esi] ds:[esi+d8] ds:[esi+d32] dh esi
4. ds:[edi] ds:[edi+d8] ds:[edi+d32] bh edi

Байт **sib** используют для расширения возможностей адресации операндов. Если он есть в команде, то эффективный адрес вычисляют так:

**эффективный\_адрес =**

**= базовый\_регистр + индексный\_регистр × масштаб + смещение.**

Как видно из таблицы 4.3, байт sib используют только когда **r/m = 100** и **mod = 00,01 и 10**. Стоит еще раз показать, какой вид имеет этот байт:

|  |  |  |  |
| --- | --- | --- | --- |
| ss | ind | ex | base |

7 0

Рассмотрим отдельно каждое поле байта **sib**.

Поле **ss** определяет масштаб. В табл 4.4 - соответствие значений поля **ss** масштабам для адресации.

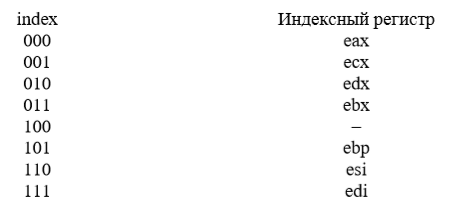
Табл 4.4. Кодирование масштабов в поле ss байта sib

**ss Масштаб**

* 1. ×1
  2. ×2
  3. ×4
  4. ×8

3 битное поле **index** определяет индексный регистр. Индексные регистры - в табл 4.5:

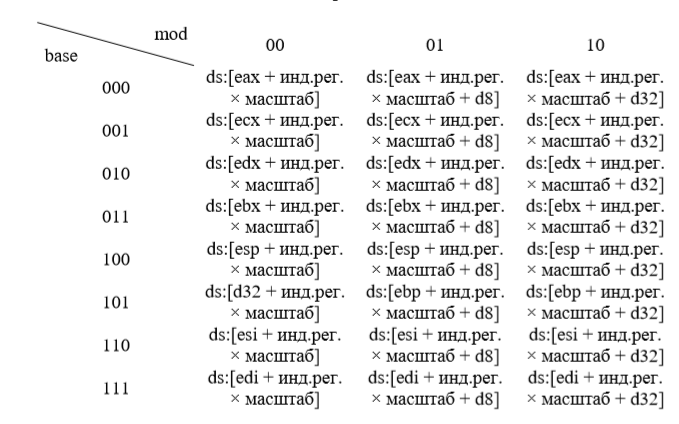
Табл 4.5. Кодирование индексных регистров в поле index байта sib



Если поле **index = 100**, то это обозначает отсутствие индексного регистра. Тогда поле **ss** должно быть равно , иначе эффективный адрес будет иметь неопределенное значение.

Поле **base** кодирует базовый регистр. В табл 4.6 показано, как вычислять адрес операнда при использовании байта **sib**.

Табл 4.6. Кодирование полей mod и base в 32битном режиме адресации



**=====================================================**

**=====================================================**

**5.1. РЕЖИМЫ АДРЕСАЦИИ.**

# 5.1.1. Общие сведения.

Операнды команд могут быть в памяти, в регистрах / в портах ввода-вывода. Где именно находится операнд и по каким правилам вычислять его физический адрес (если он в памяти) определяет режим адресации. 4 режима адресации: регистровый, непосредственный, прямой, и косвенный. Какой из них используют в данной команде, определяет транслятор по синтаксису задания этой команды в исх1 м тексте на Ассемблере.

# 5.1.2. Режимы адресации.

1) Регистрова адресация. В ней операнды находятся в регистрах МКпроцессора:

**mov ax, bx** ; содержимое **bx** **пересылают** в **ax**.

Преимущество этого режима – в том, что нет обращения к памяти – за счет этого сокращается время выполнения команды. Если бы не ограниченное количество регистров, этот способ адресации был бы удобным.

2) Непосредственная адресация. Подразумевает, что операнд лежит в самой машинной команде. Например:

**mov al, 5** ; в регистр al занесли число 5

Плюс непосредственной адресации в том, что нет обращения к памяти, недостаток – то, что значение непосредственного операнда задают только 1 раз и только перед транслированием проги, его нельзя изменить при выполнения проги.

3) Прямая адресация. В ней поле смещения в машинной команде содержит эффективный адрес ячейки памяти, где хранят запрашиваемые данные. Этот режим адресации используют, когда операнд команды - переменная, лежащая в памяти, например:

|  |  |
| --- | --- |
| **ar1 dw 10** … | ; определение переменной ar1 |
| **mov ax, ar1** | ; в регистр **ax** занесли значение переменной **ar1** |
| **mov [20h], bl** | ; в байт памяти по адресу **ds:20h** отправили содержимое  ; регистра **bl** |

Прямая адресация просто делает вычисление физического адреса операнда, но в ней нет гибкости в определении местоположения операнда при выполнении проги, которая есть у косвенной адресации (местоположение операнда фиксировано).

4) Косвенная адресация. В ней в машинной команде нет ни значения операнда, ни его эффективного адреса. А есть инфа для получения (то есть вычисления) эффективного адреса. Другими словами, МКпроцессор не может напрямую определить адрес ячейки памяти, а должен определять его косвенно. 8 видов косвенной адресации: базовая, индексная, базово-индексная, базовая со смещением, индексная со смещением, базово-индексная со смещением, базовая с масштабированным индексом и смещением.

4.1.) Базовую и индексную адресации объединяют вместе и называют косвенной регистровой адресацией. Здесь эффективный адрес операнда находится либо в базовых регистрах (для 16 битной адресации – bx, bp), либо в индексных регистрах (для 16 битной адресации – si, di). Например:

**ar1 db 5,6** ; определение переменной ar1

…

**mov si, offset ar1** ; в si загрузили смещение переменной ar1

**mov al, [si]** ; в al поместили число 5

**inc si** ; значение si увеличили на 1, команда **inc** – увеличить на 1

**mov ah, [si]** ; в ah занесли значения по адресу ar1+1, то есть 6

При косвенной адресации для явного указания размера операнда, извлекаемого из памяти, используют атрибутный оператор **ptr**. Его формат:

**тип ptr выражение**.

Явно тип надо указывать, когда неизвестно, как трактовать ячейку памяти. Пример явного указания размера операнда:

**mov byte ptr [bx], 5** ; в байт памяти по адресу, который находится в bx,

занесли число 5.

При базовой / индексной адресации со смещением эффективный адрес операнда равен сумме содержимого базового / индексного регистра и указанного в машинной команде смещения. Например:

**mas db 0,2,4,6,8,10** ; объявление массива 1 байтных чисел

…

**mov bx, 5**  ; в bx занесли значение 5 – № элемента массива

**mov al, mas[bx]** ; в **al** загрузили элемент массива с индексом 5, то есть

число 10

Тот же результат будет, если загрузить в bx не индекс, а адрес массива, а в команде указать смещение:

|  |  |
| --- | --- |
| **mov bx, offset mas** | ;в bx занесли эффективный адрес первого элемента массива |
| **mov al, 5+[bx]** | ; в **al** положили шестой элемент массива (число10) |

Последняя команда равносильна записям **mov al, [bx+5]** или **mov al, 5[bx]**.

4.2) Базово-индексная адресация возможна лишь при формировании 16 битного эффективного адреса, при 32 битном эффективном адресе вместо нее реализуют базовую адресацию с масштабированным индексом и смещением.

В базово-индексной адресации эффективный адрес операнда равен сумме содержимого базового и индексного регистров. А если в формате команды есть смещение, то его добавляют к этой сумме. Например:

**mas db 0,2,4,6,8,10** ; объявление массива 1 байтных чисел

…

**mov bx, offset mas** ; в bx загружают смещение первого элемента массива

(элемента 0)

**mov si, 5** ; число 5 занесли в регистр si, – индекс в массиве

**mov al, [bx][si]** ; в al загрузили число 10, эффективный адрес равен

сумме значений в регистрах **bx** и **si.**

Фрагмент проги с базово-индексной адресацией со смещением:

**mas db 0,2,4,6,8,10** ; объявление массива 1 байтных чисел

…

mov bx, 1 ; в bx загрузили число 1

mov si, 3 ; в si занесли 5

mov al, mas[bx][si] ; в al загрузили число 8, эффективный адрес равен

сумме значений в регистрах bx и si и смещения массива mas

4.3) Базовая с масштабированным индексом и смещением адресация идет только при 32 битной адресации сегмента. Этот режим определяется байтом **sib** машинной команды. Эффективный адрес вычисляют так:

**базовый\_регистр + индексный\_регистр × масштаб + смещение.**

Пример:

|  |  |
| --- | --- |
| **mas dw 100 dup(?)**  … | ; определение массива из 100 слов |
| **mov esi, 5** | ; в **esi** - № извлекаемого элемента массива |
| **mov ax, mas[esi\*2]** | ; в ax занесли шестой элемент массива |

Плюс косвенной адресации – это гибкость в определении адреса операнда. То есть, в момент выполнения проги можно получить доступ к произвольной ячейке памяти, а не только к описанным в исх1 м тексте Ассемблера переменным. Недосток косвенной адресации - это сложность вычисления эффективного адреса, то есть дополнительные затраты времени.

# 5.1.3. Правила использования регистров и смещений.

МКпроцессор имеет набор правил, по которым для адресации операндов неявно используют некоторые сегментные регистры. Иногда сегментные регистры можно переопределить, а иногда это сделать невозможно. Эти случаи приведены в табл 5.1.

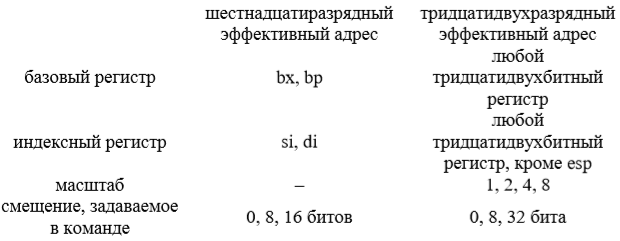
Табл 5.1. Правила комбинирования сегментных регистров и регистров, содержащих смещения.

|  |  |  |
| --- | --- | --- |
| **Регистр, содержащий смещение** | **Сегментный регистр, назначаемый по умолчанию** | **Префикс переопределения сегмента** |
| ip | cs | невозможен |
| sp | ss | невозможен |
| **bp** | **ss** | **ds, es, cs** |
| **bx** | **ds** | **ss, es, cs** |
| **si, di, не включая строковые команды** | **ds** | **ss, es, cs** |
| si для строковых команд | ds | невозможен |
| di для строковых команд | es | невозможен |

Например, в команде **mov cl, [bp]** для адресации используют сегментный регистр **ss**. Явно переопределить его можно так:

**mov cl, ds:[bp].**

Есть разница в использовании регистров и смещений при вычислении эффективного адреса в зависимости от разрядности этого адреса -см Табл 5.2.

Табл 5.2. Разница использования регистров и смещений в косвенной адресации 

--------

Часть 2

**5.2. КОМАНДЫ ПЕРЕСЫЛКИ.**

# 5.2.1. Команда пересылки mov.

Команда mov (move – перемещать) - основная команда пересылки данных:

**mov приемник, источник**

По команде mov содержимое источника пересылают в приемник.

Виды пересылки:

− из регистра в регистр; − из регистра в память;

− из памяти в регистр;

− непосредственный операнд в регистр;

− непосредственный операнд в память.

Например:

**mov ax, bx** ; содержимое bx пересылают в ax

При использовании команды **mov** запрещено пересылать из памяти в память, загружать непосредственно адресуемый операнд в регистр сегмента, пересылать содержимое 1 го сегментного регистра в другой, использовать регистр cs как приемник.

# 5.2.2. Команда замены xchg.

Команда xchg меняет местами содержимое 2 ячеек:

**xchg операнд1, операнд2**

Команда xchg меняет местами содержимое 2 регистров / регистра и ячейки памяти, кроме сегментного регистра. Например:

**ar1 db ?** ; определение переменной ar1

…

**xchg ax, bx** ; содержимое bx ложат в ax, содержимое ax ложат в bx

**xchg cl, ar1** ; данные ячейки ar1 и регистра cl меняются местами.

Команда **xchg** заменяет 3 команды пересылки и не требует промежуточной ячейки памяти.

# 5.2.3. Команда перестановки байт bswap.

Команда **bswap** (byte swap – поменять местами байты):

**bswap операнд**

Операндом может быть только 32 битный регистр. Команда **bswap** изменяет порядок следования байт на противоположный: первый байт меняется местами с 4, второй – с третьим. Например:

**.486** ; директива переключения типа МКпроцессора

…

**mov eax, 12345678h** ; занесение в регистр eax значения 12345678h **bswap eax** ; теперь значение в eax = 7856 3412h

# 5.2.4. Загрузка сегментных регистров.

Загрузку инфы в сегментные регистры делают такие команды:

lds регистр, память les регистр, память lfs регистр, память lgs регистр, память

lss регистр, память

Каждая из этих команд выполняет загрузку определенного сегментного регистра и регистра, указанного как операнд, данными из заданной ячейки памяти. Команда lds – load pointer to ds – загрузить указатель в ds, les – в es, lfs – в fs, lgs – в gs, lss – в ss.

При 16 битном режиме адресации из памяти извлекают 4 байта, старшее слово загрузили в сегментный регистр, младшее – в регистр, определенный в команде. Например:

**ar1 dd 1234**5678**h** ; определение переменной ar1

…

**lds ax, ar1**  ; в регистр ds занесли значение **1234**h (старшее

; слово), в ax – 5678h (младшее слово).

При 32 битном режиме адресации из памяти забирают 6 байт, 2 старших заносят в сегментный регистр, а остальные 4 – в регистр, указанный в команде.

**5.2.5. Загрузка эффективного адреса.**

Загрузку эффективного адреса выполняет команда lea (load effective address – загрузить эффективный адрес):

**lea регистр, память**

По этой команде в регистр пересылают смещение указанной ячейки памяти. Например:

**ar1 db ?** ; определение переменной ar1

…

**lea ax, ar1** ; в регистр ax занесли смещение относительно

; регистра ds переменной ar1

**mov bx,1000h** ; в bx занесли число 1000h

**lea cx, [bx+40h]** ; в cx загрузили смещение ячейки с адресом

; (bx)+40h, то есть число 1040h

---

Часть 3

**6.1. КОМАНДЫ РАБОТЫ СО СТЕКОМ.**

**6.1.1. Понятие стека. Адресация данных в стеке.**

Стек – это область оперативной памяти для временного хранения инфы. Программа может поместить данные в стек / забрать их оттуда. Первым из стека забирают то, что занесено последним, то есть LIFO.

Для адресации данных вну3 стека используют 2 регистра: **ss** (содержит адрес начала сегмента стека) и **esp/sp** (содержит значение указателя вершины стека). На рис. 6.1 - схема адресации данных в стеке для 16 битного режима адресации.

вершина стека

дно стека

оперативная память

64

Кбайта

направление

роста

стека

ss

:

0000

h

ss

:

sp

ss

:

0

ffffh

Рис. 6.1. Адресация данных вну3 стека

Стек растет в сторону уменьшения адресов, то есть при занесении инфы адрес вершины стека уменьшают. Для работы со стеком используются специальные команды: push, pop и производные от них. Неявно стек используют при выполнении команд **call** (вызов процедуры), **ret** (возврат из процедуры), **int** (вызов прерывания), **iret** (возврат из подпроги обработчика прерывания) и других.

Есть возможность доступа к произвольному элементу стека – с помощью регистра **ebp/bp**. Например, если поместить в него смещение определенного элемента, то пара **ss:ebp/bp** будет адресовать операнд, содержащийся в памяти.

**6.1.2. Стековые команды.**

Для занесения данных в стек используют команду **push** (push – толкать):

**push операнд**

Команда **push** сначала уменьшает значение в регистре **esp/sp** на 2, если размер заносимого операнда – 2 байта. Если размер операнда равен 4 байта, то **esp/sp** уменьшают на 4. Затем операнд заносят на новую вершину стека (то есть в ячейку, на которую указывает **esp/sp**). Операндом может быть

любой 2 байтный регистр / 2 байтная ячейка оперативной памяти. Пример:

|  |  |
| --- | --- |
| **ar1 dw ?**  … | ; определение 2 байтной переменной ar1 |
| **push ar1** | ; занесение в стек значения переменной ar1 |
| **push ax**  … | ; занесение в стек содержимого регистра ax |
| **.386** | ; директива переключение типа МКпроцессора |
| **push ecx** | ; занесение в стек содержимого ecx |
| **push 9ah** | ; занесение в стек числа 9ah |

Для извлечения данных из стека используют команду **pop** (**pop** – совать):

**pop операнд**

По команде **pop** содержимое операнда (ячейка памяти / регистр) заменяют словом / двойным словом, которое забирают с вершины стека. Вершина стека адресуется парой **ss:sp** (при 16 битном режиме адресации сегмента) **/ ss:esp** (при 32 битном). После этого значение регистра **esp/sp** уменьшают на 2 (если извлекалось 2 байта) / на 4 (если извлекалось 4 байта). Теперь этот регистр указывает на новую вершину стека. Пример:

**ar1 dw ?** ; определение 2 байтной переменной ar1

ar2 **dw ?**  ; определение 2 байтной переменной ar2

…

**mov ar1, 0ffaah** ; занесение в переменную ar1 значения 0ffaah

|  |  |
| --- | --- |
| **push ar1** | ; занесение в стек значения переменной ar1 |
| **mov ax, 12** | ; занесение в регистр ax числа 12 |
| **push ax** | ; занесение в стек содержимого регистра ax |
| **pop ar2** | ; из стека забирают 2 байта, которые заносят в  ; переменную ar2; значение ar2 = 12 |
| **pop cx** | ; в cx находится число 0ffaah |

Для сохранения в стеке значения всех регистров общего назначения, а также значения регистров базы и индексных регистров - команда **pusha** (push all – занести все):

**pusha**

**pushaw**

**pushad**

Она сохраняет регистры в таком порядке:

eax/ax, ecx/cx, edx/dx, ebx/bx, esp/sp, ebp/bp, esi/si, edi/di (для 16 битного режима адресации в стек заносят 2 байтные регистры, для 32 битного – 4 байтные). Значение регистра sp уменьшают на 16 – так как в стек занесли 8 2 байтных элементов; esp уменьшают на 32 – так как в стек занесли 8 4 байтных элементов. По времени выполнения **pusha** медленнее восьми команд **push**, но в памяти **pusha** занимает всего 1 байт.

Команды **pushaw** и **pushad** – это модификации команды **pusha**. Команда **pushaw** (pusha word) заносит в стек 16 битные регистры, **pushad** (pusha double word) – 32 битные. Например, если установлен 32 битный режим адресации, а в стек надо поместить 2 байтные регистры, то команда **pushaw** будет преобразована транслятором в команду **pusha** с префиксом размера операнда.

Для выполнения обратных по отношению к **pusha** действий (для извлечения из стека восьми элементов и помещения их в соответствующие регистры) есть команда popa (pop all – извлечь все):

**popa**

**popaw**

**popad**

Элементы заносят в регистры в порядке, обратном их записи в стек:

**edi/di, esi/si, ebp/bp, esp/sp, ebx/bx, edx/dx, ecx/cx, eax/ax.**

Извлеченное из стека значение **esp/sp** никуда не занесли – его просто отбрасили. После выполнения **popa** значение в регистре **esp/sp** увеличивается на 16 / на 32 (в зависимости от режима адресации). Команда **popaw** является 16 битной модификацией **popa**, команда **popad** – 32 битной.

Для сохранения в стеке регистра флагов – команда **pushf** (push flags – занести регистр флагов):

**pushf**

Команда pushf при 16 битном режиме адресации уменьшает значение **sp** на 2 и заносит в новую вершину стека значение регистра **flags**. При 32 битном режиме адресации **esp** уменьшают на 4 и в стек заносят **eflags**. Для извлечения данных из стека в регистр флагов применяют команду **popf** (pop **flags** – извлечь регистр флагов):

**popf**

Команда **popf** извлекает 2 байта из стека и загружает их в регистр flags (/ извлекает 4 и заносит их в eflags). Использование команд **pushf** и **popf** – единственный способ получить прямой доступ к регистру флагов.

---

Часть 4

**6.2. КОМАНДЫ ПОРТОВОГО ВВОДА-ВЫВОДА.**

# 6.2.1. Понятие порта. Адресация портов.

Каждое системное устройство ввода-вывода имеют 1 / несколько регистров, доступ к которым идет через адресное пространство ввода-вывода. Эти регистры имеют разрядность 8, 16 / 32 бита. Адресное пространство ввода-вывода физически независит от пространства оперативной памяти и имеет ограниченный объем в 65 536 адресов ввода-вывода. Поэтому, порт ввода-вывода – это восьми-, 16- / 32 разрядный аппаратный регистр, имеющий определенный адрес в адресном пространстве ввода-вывода. Адреса от **00f8h** до **00ffh** зарезервированы, их не используют для адресации портов.

Для управления устройством через порты надо знать номера этих портов и особенности представления данных, записываемых в порт и считываемых из порта. Эта инфа - служебная и приводится в технической документации по данному устройству.

# 6.2.2. Команды портового ввода-вывода

Над портами ввода-вывода определены только две операции – чтение и запись. Для чтения инфы из порта используют команда in (input from port – ввод из порта):

**in аккумулятор, порт**

По команде **in** из указанного порта в регистр-аккумулятор загрузили байт, слово / двойное слово. Байт читается в том случае, если в качестве аккумулятора указан регистр **al**, слово – если указан **ax**, двойное слово – если указан **eax**. 2 способа задания порта:

1) как непосредственный операнд команды, если адрес порта лежит в диапазоне от 0 до 255;

2) адрес порта указывают в регистре **dx**, Тогда возможен доступ ко всему адресному пространству ввода-вывода. Пример:

**in al, 64h** ; ввести байт из порта с номером 64h в регистр al **mov dx, 0ffah** ; занесение в регистр dx числа 0ffah

**in al, dx** ; ввести байт из порта с номером 0ffah в регистр al

Для записи инфы в порт применяют команду **out** (output to port

– вывод в порт):

**out порт, аккумулятор**

Данную команду заносят в указанный порт значение, которое содержится в регистре-аккумуляторе. Если аккумулятор - указан регистр **al**, то в порт записывают 1 байт, если **ax** – 2 байта, если **eax** – 4 байта. Адрес порта задают как непосредственный операнд (для диапазона 0..255) / как содержимое регистра **dx** (для диапазона 0..65 535). Пример:

**out 70h, al** ; занесение **в** порт с номером 70h содержимого регистра al

**mov dx,012ch** ; занесение в регистр **dx** значения 12ch

**out dx, al** ;занесение в порт с номером 12ch содержимого регистра al

Еще две команды для работы с портами ввода-вывода – **ins** и **outs**. Команда **ins** (input string) вводит байт, слово / двойное слово из порта, адресуемого регистром **dx**. Она имеет следующий формат:

**ins приемник, dx**

**insb**

**insw**

**insd**

Эта команда читает инфу из порта, адресуемого значением регистра **dx**, и заносит его в ячейку памяти, на которую указывает пара **es:edi/di**. Команда **insb** читает 1 байт, **insw** – 2 байта, **insd** – 4 байта; операнд команды **ins** (то есть приемник) используют только для определения размера передаваемых данных. После записи значения в память содержимое регистра **edi/di** меняется: если флаг **DF** установлен в 1, то **edi/di** уменьшают, если **DF=0**, то **edi/di** увеличивается. Значение **edi/di** меняется на размер сохраняемого операнда.

Для вывода инфы в порт применяют команду **outs** (outs – output string):

**outs dx, источник**

**outsb outsw outsd**

Эта команда передает в порт байт, слово / двойное слово памяти, которая адресуется **ds:esi/si**. Адрес порта находится в **dx**. После записи значения в порт содержимое **esi/si** меняется на размер операнда.

ins и outs используют с префиксом повторения **rep** – это 1 байтный префикс, который выполняет повторение команды столько раз, сколько указано в регистре **ecx/cx** (ecx – при 32 битном режиме адресации, cx – при 16 битном). Пример:

|  |  |
| --- | --- |
| **str1 db 768 dup (?)**  … | ; отведение места для строки из 768  ; 1 байтных элементов |
| **mov ax, seg str1** | ; в регистр ax занесли сегментную компоненту  ; адреса переменной str1 |
| **mov es, ax** | ; инициализация регистра **es** |
| **mov di, offset str1** | ; в регистр **di** занесли смещение переменной str1 |
| **mov dx, 03c9h** | ; в **dx** занесли номер порта |
| **mov cx, 768** | ; счетчик числа повторений |

**cld**  ; установка в 0 флага DF – значение di будет увеличиваться

**rep insb** ; ввод инфы из порта в строку

# 6.2.3. Пример проги, работающей с портами ввода-вывода.

Рассмотрим прогу, реализующую работу с CMOS – это энергозависимая память ограниченного объема (256 байт), в которой хранится системная инфа (например, текущие дата и время, тип винчестера, размер оперативной памяти и т.д.). Эта инфа нужна для правильной загрузки и работы компа. Так как CMOS-память важна, то при включении питания вначале идет проверка CMOS. 2 байта CMOS отводятся для хранения контрольной суммы – суммы значений 1 байтных ячеек памяти, которые не должны изменяться. При загрузке компа вычисляется сумма значений в 1 байтных ячейках по адресам от 10h до 2dh (их значения постоянны во время загрузки), а затем эта сумма сравнивается со значением контрольной суммы, хранимой в 2 байтной ячейке CMOS. Если совпадения нет, то система предлагает сконфигурировать настройки компа.

Работа с CMOS-памятью идет через порты **70h** и **71h**. Сначала в порт **70h** помещается номер байта, к которому пойдет обращение, а затем через порт **71h** идет чтение / запись этого байта. Пример проги, устанавливающей текущее значение минут = 45 и получающей значение контрольной суммы, хранимой в CMOS:

stack1 segment stack ; сегмент стека

db 100h dup(?) ; под стек отводится 256 байт

stack1 ends

code segment ; сегмент кода assume cs:code,ss:stack1

begin:

|  |  |
| --- | --- |
| mov al, 2 ; во втором байте CMOS хранится текущая минута  out 70h, al ; в порт 70h помещается номер байта  mov al, 45h ; задают значение минут – данные представляют | |
|  | ; в упакованном BCD формате  ; (4 бита кодируют одну десятичную цифру) |
| out 71h, al | ; запись нового значения минут, получение значения ; контрольной суммы CMOS |
| mov al, 2eh    out 70h, al | ; контрольная сумма хранится в байтах с адресами 2eh и 2fh |
| in al, 71h | ; в al – старший байт контрольной суммы |
| mov ah, al mov al, 2fh out 70h, al | ; занесение старшего байта в регистр ah |
| in al,71h | ; теперь в al - младший байт контрольной суммы  ; таким образом, в ax - контрольная сумма CMOS |
| mov ah, 4ch | ; корректный выход из проги в DOS |
| int 21h | ; номер функции помещают в регистр ah, затем  ; вызывается прерывание DOS – прерывание с  ; номером 21h |

code ends

end begin

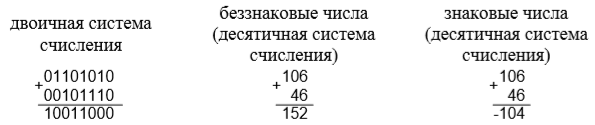
---

Часть 5

**7.1. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ НАД ЦЕЛЫМИ ДВОИЧНЫМИ ЧИСЛАМИ.**

# 7.1.1. Команды сложения.

Содержимое каждой ячейки памяти / регистра можно расценить и как число со знаком, и как беззнаковое число. После выполнения арифметической операций МКпроцессор устанавливает весь спектр флажков, и только исходя из значений флажков можно определить правильность результата операции. Например, если данные хранят в 1 байтных ячейках:



Устанавливаемые флаги: **CF = 0, OF = 1, ZF = 0, SF = 1.**

При сложении беззнаковых чисел флаг **CF** - это признак переполнения; **CF = 0** – нет без-знакового переполнения, **CF = 1** – есть без-знаковое переполнение. При сложении знаковых чисел флаг **OF** – это признак переполнения; **OF = 0** – нет знакового переполнения, **OF = 1** – есть знаковое переполнение.

В МКпроцессоре есть несколько команд сложения. Самая распространенная – это команда add:

**add приемник, источник**

По команде add сумма значений источника и приемника заносят на место операнда-приемника. В соответствии с результатом операции изменяются такие флаги регистра **eflags/flags**: **CF, PF, AF, ZF, SF, OF**. Приемником может быть регистр / ячейка памяти, источником – регистр, ячейка памяти / непосредственный операнд.

Для учета значения флага **CF** в операции сложения есть команда **adc** (add with carry – прибавить с переносом):

**adc приемник, источник**

Команда **adc** суммирует операнды и прибавляет к этой сумме 1, если к началу сложения флаг **CF** был установлен в 1. Если к началу сложения **CF** был сброшен в 0, то выполнение **adc** аналогично выполнению команды **add**. Так как **adc** использует перенос от предыдущей операции, эта команда может применяться для реализации сложения чисел неограниченной разрядности. В соответствии с результатом операции после выполнения **adc** модифицируют такие флаги: CF, PF, AF, ZF, SF, OF.

Приемником может быть регистр / ячейка памяти, источником – регистр, ячейка памяти / непосредственный операнд.

Пример корректного сложения двух 1 байтных чисел:

**op1 db ?** ; определение переменной op1 op2 **db ?** ; определение переменной op1

…

**xor ah, ah** ; обнуление регистра ah – команда xor эффективнее

; для обнуления, чем команда mov ah,0

**mov al, op1** ; в al занесли переменную op1 – так как напрямую

; выполнить сложение “память – память” невозможно,

; используют регистр

**add al, op2** ; к содержимому al прибавляют значение op2,

; результат заносят в al

**adc ah, 0** ; прибавление к содержимому **ah** (**ah** был обнулен в

; самом начале) значения 0 и флага CF; таким образом, в

; регистре ax лежит корректный результат сложения чисел op1 и op2.

inc – это 1 операндная команда сложения (increment – увеличить), для увеличения значения операнда на единицу.

Команда xadd (exchange and add – поменять и сложить):

**xadd приемник, источник**

Она сначала в источник заносит значение приемника, затем сумму приемника и источника до изменения (настоящего источника) сохраняет в приемнике. Например:

|  |  |
| --- | --- |
| .486 | ; для использования **xadd** нужен МКпроцессор 80486 и выше |
| mov al, 8 | ; занесение в регистр al числа 8 |
| mov cl, 2 | ; занесение в регистр cl числа 2 |
| **xadd** al, cl | ; после выполнения xadd в al - число 10, содержимое cl = 8 |

После выполнения **xadd** флаги изменяются в соответствии с результатом операции: **CF, PF, AF, ZF, SF, OF.**

# 7.1.2. Команды вычитания.

При вычитании чисел, хранящихся в 1 байтных ячейках, будут результаты:

|  |  |  |
| --- | --- | --- |
| **двоичная система счисления** | **беззнаковые числа (десятичная система счисления)** | **знаковые числа(десятичная система счисления)** |

11101010 234 -22

– – –

00101110 46 46

10111100 188 -68

Устанавливаемые флаги: **CF=0, OF=0, ZF=0, SF=1**.

Результат вычитания корректен и для беззнаковых чисел, и для чисел со знаком. Флаг **CF** устанавливают в 1, если был заем из старшего разряда (например, для 1 байтной операции: (- 86) - (- 46)). Флаг **OF** устанавливают в 1, если произошло переполнение в старшем бите (например, для 1 байтной операции: (200) - (- 120)).

Команда вычитания **sub** (subtract – вычитать):

**sub приемник, источник.**

**sub** вычитает из приемника источник и заносит эту разность в операнд-приемник. В соответствии с результатом выполнения операции устанавливаются флаги **CF, PF, AF, ZF, SF, OF**. Приемником может быть регистр / ячейка памяти, источником – регистр, ячейка памяти / непосредственный операнд.

Для учета заема из старшего бита используют команда **sbb** (subtract with borrow – вычесть с заемом):

**sbb приемник, источник.**

Команда **sbb** вычитает из приемника источник, а затем из результата вычитает 1, если к началу операции флаг **CF** был установлен в 1. Если к моменту начала выполнения sbb флаг CF = 0, то выполнение этой команды аналогично выполнению **sub**. Команду sbb использовать при вычитании многоразрядных чисел. Выполнение **sbb** модифицирует такие флаги:

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

Приемник - регистр / ячейка памяти, источник – регистр, ячейка памяти / непосредственный операнд.

Пример вычитания двух 2 байтных чисел:

**op1 db 12h, 0abh** ; определение переменной op1, младший байт

|  |  |
| --- | --- |
|  | ; 12h, старший – 0abh |
| **op2 db 0fah, 45h**  … | ; определение переменной op2, в младшем  ; байте содержится число 0fah, в старшем – 45h |
| **mov dl, op1** | ; в регистр **dl** занесли младший байт op1 |
| **sub dl, op2** | ; из dl вычли значение младшего байта переменной op2 ; (от числа 12h отнимается число 0fah) |
| **mov dh, op1+1** | ; в dh занесли старший байт op1 |
| **sbb dh, op2+1** | ; из регистра **dh** вычли старший байт op2 с учетом  ; заема (от числа 0abh отнимается 45h и 1); теперь в dx –  ; корректный результат вычитания двух 2 байтных чисел |

Команда **dec** уменьшает значение операнда на 1.

Команда **neg** изменяет знак операнда – вычитает операнд из 0.

Команда сравнения **cmp** вычитает 1 операнд из другого, но никуда не заносит результат – только устанавливает флаги.

# 7.1.3. Команды умножения.

Две команды операции умножения. Команда **mul** умножает без-знаковые числа, получаемый результат – также беззнаковое число. Команда **imul** умножает числа со знаком, результат – знаковое число.

Команды без-знакового умножения **mul** (multiply – умножать):

**mul источник**

Источником может быть только регистр / ячейка памяти. Если указан 1 байтный операнд, то на него умножили содержимое регистра **al**, и результат идет в регистр **ax**. Если указан 2 байтный источник, то он перемножается со значением в регистре ax, и результат идет в пару **dx:ax** (старшая часть результата – в dx, младшая – в ax). Если источник 4хбайтный, то его значение умножили на **eax**, и результат идет в **edx:eax** (edx – старшая часть результата, eax – младшая часть результата). Команда **mul** модифицирует флаг переноса **CF** и флаг переполнения **OF**. Флаги CF и OF устанавливаются в 1 тогда, когда после команды mul старшая часть результата (ah, dx и eax соответственно для 1 байтного, 2 байтного и 4хбайтного умножения) имеет хотя бы 1 отличный от 0 бит, иначе (когда старшая часть равна 0) флаги CF и OF сбрасывают в 0. Пример:

**mov al, 5** ; в регистр al занесли число 5

**mov cl, 240** ; в регистр cl занесли число 240

**mul cl** ; содержимое **al умножили на значение в cl**,

; результат занесли в **ax** (теперь в **ax** 1200)

После выполнения команды **mul** значения флагов PF, AF, ZF и SF не определены. Но это не значит, что данные флаги не модифицировались в процессе умножения.

Команда умножения чисел со знаком **imul** (integer multiply – умножить целые знаковые) имеет 3 формата:

**imul источник**

**imul приемник, источник**

**imul приемник, операнд1, операнд2**

Источник в 1 операндном формате команды **imul** (imul источник) – это байт, слово / двойное слово, размещенные в ячейке памяти / в регистре. Если размер источника – 1 байт, то значение источника умножили на **al**, результат идет в **ax**. Если – 2 байта, то значение источника умножили на **ax**, результат идет в **dx:ax**. Если – 4 байта, то умножили на **eax**, результат идет в **edx:eax**. После выполнения команды **imul** изменяются флаги CF, PF, AF, ZF, SF, OF. Флаги **CF** и **OF** сбрасывают в 0, когда старшая часть результата равна 0 / заполнена единицами (ah=0ffh, dx=0ffffh, edx=0ffffffffh – то есть результат отрицательное число, которое идет в младшую части ячейки для его хранения). Иначе (если результат не помещается в младшей части ячейки) CF и OF устанавливаются в 1.

Пример:

mov al, -5 ; в al занесли число -5

mov cl, 120 ; в cl занесли число 120

imul cl ; значение в al умножили на значение в cl результат

; занесли в ax; после умножения значение в ax = -600

При выполнении команд **mul** и **imul** в 1 операндном формате, получаемый результат всегда верен – то есть размер ячейки, отводимой для него, всегда достаточен (например, при умножении байт 255 × 255=65025, это значение помещается в 2 байта). В таблице 7.1 - соответствие между операндами и результатом команд mul и imul.

Табл 7.1. Соответствие между операндами и результатом для команд умножения **mul** и **imul** (1 операндный формат).

|  |  |  |
| --- | --- | --- |
| **размер источника** | **аккумулятор** | **результат** |
| **байт** | al | **ax** (ah – старшая часть результата, al – младшая) |
| **слово** | ax | **dx:ax** (dx – старшая часть результата, ax – младшая) |
| **двойное слово** | eax | **edx:eax** (edx – старшая  часть, eax – младшая) |

Начиная с МКпроцессора 80186 вводятся 2 операндный и 3 операндный форматы команды знакового умножения.

Команда **imul** в 2 операндном формате (**imul приемник, источник**) умножает значение приемника на значение источник и заносит результат в приемник. Операнд-приемник может быть только регистр. Источник – это непосредственный операнд, ячейка памяти / регистр. Пример:

|  |  |
| --- | --- |
| **op1 dw ?**  …  **.186** | ; определение 2 байтной ячейки памяти |
| **mov cx, 1000** | ; в cx занесли значение 1000 |
| **imul cx, -5**  …  **.386** | ; содержимое **cx** умножили на -5, результат занесли в ; **cx** (после выполнения умножения в cx лежит -5000). |
| **mov bx, 1000** | ; в bx занесли значение 1000 |
| **mov op1, -5** | ; инициализация переменной op1 значением -5 |
| **imul bx, op1** | ; умножение значения регистра bx на значение  ; переменной op1 (после умножения в bx -5000) |

Команда imul в 3 операндном формате (**imul приемник, операнд1, операнд2**) перемножает значения операнд1 и операнд2, а затем заносит результат в приемник. Приемником может быть только регистр. Операнд1 может быть определен как регистр / как ячейка памяти. Операнд2 – всегда непосредственный операнд. Пример:

.186

mov cx, -5 ; в cx занесли число -5

imul ax ,cx, -5 ; значение **cx** умножили на -5, результат занесли в ax = 25.

Команда imul в 2 операндном и в 3 операндном форматах изменяет флаги CF, PF, AF, ZF, SF, OF. Флаги CF и OF устанавливаются в 1 тогда, когда результат умножения не помещается в отведенный приемник. Иначе (когда разрядности приемника достаточно для представления результата) флаги **CF** и **OF** сбрасывают в 0.

# 7.1.4. Команды деления.

Две команды: **div** (беззнаковое деление) и **idiv** (деление чисел со знаком). Результат представляется в виде частного и остатка.

Деление над без-знаковыми числами - это команда div (divide – делить):

**div источник**

Деления над знаковыми числами – это команда **idiv** (integer divide – целые):

**idiv источник**

Источник в обеих командах деления – это делитель; им может быть только операнд в памяти / регистр. Делимое всегда подразумевается неявно.

Размер источника определяет, какие регистры образуют делимое. Связь между делителем, делимым, частным и остатком - в таблице 7.2.

Табл 7.2. Связь между делителем, делимым и результатом команд div и

|  |  |  |  |
| --- | --- | --- | --- |
|  | idiv |  |  |
| **Делимое**  **(аккумулятор)**  ax (ah – старшая | **Размер делителя (источник)** | **Частное** | **Остаток** |
| часть, al – младшая)  dx:ax (dx – старшая часть, ax– младшая) edx:eax | 1 байт  2 байта | al  ax | ah  dx |
| (edx – старшая часть, eax– младшая) | 4 байта | eax | edx |
|  |  |  |  |

Если размер указанного источника – 1 байт, то на него делится значение в регистре **ax**, и частное от деления занесли в **al**, остаток – в **ah**. Если размер делителя – 2 байта, то делимое – **dx:ax**, частное занесли в **ax**, остаток – в **dx**. Если размер делителя – 4 байта, то делимое – **edx:eax**, частное – в **eax**, остаток – в **edx**. При команде **div** значения делимого, делителя, остатка и частного понимаются как без-знаковые числа. В **idiv** значения делимого, делителя, остатка и частного понимаются как числа со знаком, причем знак остатка всегда совпадает со знаком делимого. Пример:

**op1 dw ?**

…

**mov ax, 20** ; делимое в ax

**mov cl, 3** ; в cl – делитель

**div cl** ; после беззнакового деления в al – частное 6, в ah – остаток 2

…

; деление числа 12ccbh = 77003d на 10

|  |  |
| --- | --- |
| **mov dx, 01h** | ; старшая часть делимого |
| **mov ax, 2ccbh** | ; младшая часть делимого. Итого: **01h 2ccbh =** 12ccbh |
| **mov op1, 10** | ; делитель |
| **div op1** | ; после без-знаков деления в ax – частное 7700, в dx – ост 3 |

…

**mov ax, -25** ; в ax – делимое **mov dl, 7** ; в dl – делитель

**idiv dl** ; знаковое деление -25 на 7, в результате в al – частное -3,

; в ah – остаток -4 (имеет тот же знак, что и делимое)

При выполнении div и idiv изменяются флаги CF, PF, AF, ZF, SF и OF, но полезную инфу о результате деления не несут. При делении возможна одна исключительная ситуация – когда результат вычислений не помещается в приемник (то есть частное не может быть сохранено в соответствующем регистре). Тогда вырабатывается прерывание с номером 0 (int 0 – деление на 0), и значения частного и остатка не определены. Эта ошибка возникает, когда делитель = 0 / когда делимое больше делителя в число раз, не меньшее увеличенного на единицу максимального модуля числа из диапазона делителя. Например, если для без-знаковой операции 1 байтный делитель равен 2, а делимое не меньше 2×(255+1).

---

Часть 6

**7.2. КОМАНДЫ КОРРЕКЦИИ ДВОИЧНО-ДЕСЯТИЧНЫХ ЧИСЕЛ.**

**7.2.1. Общие сведения.**

При представлении данных в неупакованном BCD-формате каждая десятичная цифра представляется в отдельном байте, например:

|  |  |  |
| --- | --- | --- |
| 04h | 08h | 09h |

десятичное число 489:

23 15 7 0

Данные в упакованном BCD-формате содержат одну десятичную цифру в каждой тетраде, то есть в 1м байте – 2 десятичные цифры:

|  |  |
| --- | --- |
| 04h | 89h |

десятичное число 489:

15 8 7 0

В МКпроцессоре нет команд операций сложения, вычитания, умножения и деления над данными в двоично-десятичном формате. А есть отдельно арифметическая команда над двоичными числами и отдельно корректируется результат / исходные операнды. Для этого есть команды коррекции сложения, вычитания, умножения и деления.

**7.2.2. Коррекция неупакованных BCD-чисел.**

Для коррекции неупакованного BCD-сложения есть команда **aaa** (ASCII adjust for add – ASCII коррекция для сложения):

**aaa**

Эта команда корректирует результат 1 байтного сложения неупакованных BCD-чисел, который лежит в **al**. Это значит, команда **aaa** должна следовать за командами add / adc, первый операнд которых – регистр **al**. Команда **aaa** преобразует младшие 4 бита (младшую тетраду) регистра al в неупакованное BCD-число, старшая тетрада al идет в 0. Если результат сложения был больше 9, то значение в регистре ah увеличивается на 1 и флаги AF и CF устанавливаются в 1. Иначе (< = 9) AF и CF сбрасывают в 0, а регистр ah не меняется. Значения флагов PF, ZF, SF и OF не определены. Пример:

**xor ah, ah** ; обнуление содержимого ah – эта команда работает

; быстрее, чем **mov ah, 0**

**mov al, 08h** ; в al неупакованное BCD-число 8

**mov bl, 09h** ; в bl неупакованное BCD-число 9

**add al, bl** ; двоичное сложение al и bl, после add в al число 17

**aaa**  ; коррекция неупакованного BCD-сложения, после

; выполнения aaa в al – 7, в ah – 1; таким образом, в

; регистре ax будет корректное неупакованное двоично- ; десятичное число

Коррекцию неупакованного BCD-вычитания делает команда **aas** (ASCII adjust for subtract – ASCII коррекция для вычитания):

**aas**

Команда aas корректирует результат 1 байтного вычитания неупакованных BCD-чисел, который лежит в al. Это значит, команда aas должна следовать за командами **sub / sbb**, первый операнд которых – регистр al. Команда aas преобразует младшие 4 бита (младшую тетраду) регистра al в неупакованное BCD-число, старшая тетрада al обнуляется. Если результат вычитания больше 9 (это соответствует тому, что уменьшаемое меньше вычитаемого), то значение в регистре ah уменьшают на 1 и флаги AF и CF устанавливают в 1. Иначе (< = 9) AF и CF сбрасывают в 0, а регистр ah не меняется. Значения флагов PF, ZF, SF и OF не определены. Пример:

mov al, 08h ; в al – **уменьшаемое**

mov bl, 09h ; в bl – вычитаемое

sub al, bl ; из al вычли значение bl

aas ; коррекция неупакованного BCD-вычитания, в результате в

; al – 9 (с учетом заема 18-9=9), значение в ah уменьшится

; на 1, CF=1.

Команда **aam** (ASCII adjust for multiply – ASCII коррекция для умножения) выполняет коррекцию неупакованного BCD-умножения:

**aam**

Она корректирует результат 1 байтного умножения двух неупакованных BCD-чисел; она должна следовать за командой **mul**. Так как максимальное значение результата = 9×9=81, то он полностью помещается в регистр al. Команда **aam** преобразует этот результат в неупакованное BCD-число путем деления содержимого al на 10, частное занесли в **ah**, остаток – в **al**. Состояние флагов PF, ZF и SF определяется значением в al, состояние флагов CF, AF и OF не определено. Пример:

**mov al, 07h** ; в al – число 7

**mov bl, 09h** ; в bl – число 9

|  |  |
| --- | --- |
| **mul bl** | ; перемножаются значения в al и в bl; результат - в ax (63=3fh) |
| **aam** | ; коррекция неупакованного BCD-умножения, после  ; выполнения в al – 03h, в ah – 06h |

Для коррекции неупакованного BCD-деления – команда aad (ASCII adjust for divide – ASCII коррекция для деления):

**aad**

Она выполняется до команды **div**. Она преобразует неупакованное двоично-десятичное делимое в **ax** в двоичное число (содержимое ah умножили на 10, это произведение прибавляется к **al**, затем значение в **ah** обнуляется). После 1 байтного деления (содержимое **ax** делится на 1 байтное число) в **al** будет верное двоичное значение. В соответствии со значением в al изменяют флаги PF, ZF и SF; состояние флагов CF, AF и OF не определено. Пример:

|  |  |
| --- | --- |
| **mov ax, 0702h** | ; в ax занесли число 72 в неупакованном BCD формате |
| **mov bl, 08h** | ; в bl – делитель 8 |
| **aad** | ; коррекция деления, теперь в ax лежит 72 = 48h |
| **div bl** | ; содержимое ax делится на значение в bl, результат:  ; частное 9 занесли в al, остаток 0 – в ah. |

**7.2.3. Коррекция упакованных BCD-чисел.**

Результат пакованных BCD-чисел корректируют только после команд сложения и вычитания. Для коррекции результата сложения используют команду **daa** (decimal adjust for add – десятичная коррекция для сложения):

**daa**

Команда **daa** делает коррекцию предшествующего результата 1 байтного сложения двух упакованных BCD-чисел. Этот результат должен быть в регистре al. После выполнения **daa** в al лежит корректное упакованное двоично-десятичное число. В соответствии со значением в al изменяют флаги CF, PF, ZF и SF; состояние флагов AF и OF не определено. Если корректируемый результат сложения > 99, то после выполнения **daa** флаг **CF** устанавливают в 1, а в al лежит усеченный (без старшей цифры) результат в упакованном BCD формате. Пример:

**mov al, 72h** ; в al занесли число 72 в упакованном BCD формате **mov dl, 43h** ; в dl занесли число 43 в упакованном BCD формате **add al, dl** ; двоичное сложение двух упакованных двоично-

; десятичных чисел; теперь в al – значение 0b5h=181

**daa**  ; в al находится число 15 в упакованном BCD формате, флаг

; CF равен 1

Для коррекции результата вычитания – команда das (decimal adjust for subtract – десятичная коррекция для вычитания):

**das**

Она корректирует предыдущий ей результат 1 байтного вычитания двух упакованных BCD-чисел. Этот результат должен быть в регистре al. После выполнения **das** в al – корректное упакованное двоично-десятичное число. Изменяются флаги CF, PF, AF, ZF, SF и OF, но значения AF и OF не определено. Если уменьшаемое меньше вычитаемого, то флаг CF устанавливают в 1. Пример:

**mov al, 72h** ; в al занесли число 72 в упакованном BCD-формате **mov dl, 23h** ; в dl занесли число 23 в упакованном BCD-формате **sub al, dl** ; из al вычли dl, результат – число 4eh=79

das ; коррекция упакованного BCD-вычитания, теперь в al число 49h

…

**mov al, 10h** ; в al – упакованное двоично-десятичное число 10 **mov dl, 25h** ; в dl – упакованное двоично-десятичное число 25 **sub al, dl** ; вычитание, в al – 0ebh

**das**  ; коррекция упакованного BCD-вычитания, теперь в al –

; число 85 в упакованном двоично-десятичном

; формате (с учетом заема 110-25 = 85); флаг CF = 1.

---

Часть 7

**7.3. КОМАНДЫ ПРЕОБРАЗОВАНИЯ ТИПОВ.**

**7.3.1. Команды расширения знакового бита.**

Для решения преобразования типов операндов – несколько команд. Преобразование типа в командах расширения знакового бита – это увеличение размера операнда с учетом знака. То есть, если в старшем бите преобразуемого операнда находится 1, то в расширяемую часть заносят также единицы (так как числа хранятся в дополнительном коде), иначе – нули. Команда cbw (convert byte to word – преобразовать байт в слово):

**cbw**

Команда **cbw** преобразует 1 байтное знаковое число в al в 2 байтное знаковое число в **ax**. Это значит, что cbw заполняет регистр **ah** значениями старшего (знакового) бита регистра **al**. Пример:

**mov al, -30**  ; в al занесли число -30 = 0e2h

**cbw** ; в **ax** будет число -30 = 0ffe2h

Команда **cwd** (convert word to dword – преобразовать слово в двойное слово):

**cwd**

Команда **cwd** преобразует знаковое слово в регистре **ax** в знаковое слово в паре регистров **dx:ax** (старшая часть – в **dx**), то есть во все биты dx занесли значение старшего (знакового) бита ax.

Команда **cwde** (convert word to dword extended – преобразовать слово в двойное слово расширенное):

**cwde**

Команда cwde преобразует знаковое слово в ax в двойное слово в **eax**. Машинный код команды **cwde** точно такой же, как и код **cbw**. При компиляции транслятор анализирует режим адресации сегмента, и если этот режим 16 битный, то перед **cwde** транслятор создает 1 байтный префикс размера операнда, и при выполнении cwde выполняется именно расширение ax в eax. Аналогично, если режим адресации 32 битный и в программе используют **cbw**, то транслятором перед **cbw** формируется префикс размера операнда, и при выполнении cbw идет преобразование **al** в **ax**.

Команда **cdq** (convert dword to quadword – преобразовать двойное слово в учетверенное слово):

**cdq**

Команда cdq преобразует знаковое 32 битное число в **eax** в знаковое 64хбитное число в **edx:eax** (**edx** – старшая часть, eax – младшая). Формат машинной команды **cdq** совпадает с машинным кодом **cwd**, поэтому при трансляции учитывается режим адресации сегмента.

Ни одна из команд расширения знакового бита (cbw, cwd, cwde, cdq) не модифицирует регистр флагов. Для преобразования типов беззнаковых чисел достаточно обнулять расширяемую часть (например, для беззнакового преобразования al в ax достаточно выполнить xor ah,ah).

**7.3.2. Команды пересылки с учетом и без учета знакового бита.**

Две команды, реализующие преобразование типов. Преобразование типа – это увеличение размера операнда с учетом знака (movsx) / без учета знака (movzx).

Для пересылки с учетом знака – команда movsx (move with sign extension – переместить с расширением знака):

**movsx приемник, источник**.

Эта команда расширяет с учетом знакового бита значение источника до размера приемника и заносит полученное число в приемник. Размер источника – 8 / 16 бит, источник – это только ячейка памяти / регистр. Размер приемника – 16 бит / 32 бита, приемник – это только регистр. Выполнение команды movsx не влияет на состояние флагов. Пример:

|  |  |
| --- | --- |
| **.386** |  |
| **op1 db ?** … | ; определение 1 байтной переменной op1 |
| mov op1,-10 | ; занесение в op1 значения -10 |
| movsx eax,op1 | ; в регистр eax занесли знаково расширенное  ; значение переменной op1; после выполнения  ; movsx в eax будет находится значение  ; 0fffffff6h = -10 = 4294967286 |

Для без-знакового расширения значения операнда – команда movzx (move with zero extension – переместить с расширением нуля):

**movzx приемник, источник**

Эта команда расширяет значение источника до размера приемника

путем занесения в старшую часть **нулей**. Полученное число кладут в приемник. Размер источника – 8 / 16 бит, источник – это только ячейка памяти / регистр. Размер приемника – 16 бит / 32 бита, приемник – это только регистр. Выполнение команды movzx не влияет на состояние флагов. Пример:

**.386**

**op1 db ?** ; определение 1 байтной переменной op1

…

**mov op1, -10** ; в op1 занесли число -10 = 0f6h = 246

**movzx eax, op1** ; после выполнения movzx в регистре eax число 0f6h=246

---

Часть 8

**7.4. ОПЕРАЦИИ НАД ОДНИМ ОПЕРАНДОМ И КОМАНДЫ**

**СРАВНЕНИЯ.**

**7.4.1. 1 операндные команды инкремента, декремента и изменения знака операнда.**

Для увеличения значения операнда на 1 – команда **inc** (increment – увеличение):

**inc приемник**

После выполнения inc содержимое приемника (приемником может быть ячейка памяти / регистр) увеличивается на 1 и меняются флаги PF, AF, ZF, SF, OF. Флаг CF командой inc не меняется.

Для уменьшения значения операнда на 1 – команда dec (decrement – уменьшение):

**dec приемник**

После выполнения dec содержимое приемника - ячейка памяти / регистр. уменьшается на 1 и флаги изменяются PF, AF, ZF, SF, OF. Флаг CF командой dec не меняется.

Команда **neg** (negate – отрицать) меняет знак операнда на противоположный:

**neg приемник**

Команда **neg** вычитает значение приемника из нуля и результат вычитания записывает обратно в приемник. Приемник – ячейка памяти / регистр. Выполнение **neg** модифицирует флаги CF, PF, AF, ZF, SF, OF. Флаг CF сброшен в 0, когда операнд равен 0, иначе CF = 1. При попытке изменения знака у минимального отрицательного числа диапазона (например, для байта – число -128, для слова – число -32768) флаг OF устанавливают в 1.

**7.4.2. Команды сравнения.**

Команда **cmp** (compare – сравнивать) сравнивает значения двух операндов между собой:

**cmp приемник, источник**

Команда **cmp** вычитает из приемника значение источника, но результат никуда не заносит – только модифицирует все арифметические **флаги** (CF,

PF, AF, ZF, SF, OF). На основании состояния этих флагов можно определить результат сравнения, поэтому после команды cmp используют какую-либо команду условного перехода. Пример:

**cmp cl, 17**  ; сравнение значения в регистре cl с числом 17. Если

; операнды равны, то при вычитании флаг нуля ZF

; будет установлен в 1

**jz l1** ; перейти на метку l1, если флаг ZF установлен в 1

…

l1: ; метка, на которую передается управление

Пример механизма проверки условия. Пусть сравниваются 2 знаковых числа и надо выяснить, что первый операнд больше второго (op1 > op2):

**cmp op1, op2**

Чтобы определить какое число больше (op1 / op2), надо 3 флага: OF (знакового переполнения), ZF (нуля) и SF (знака). Флаг нуля: чтобы op1 был строго больше op2, результат вычитания должен быть отличен от 0; значит, **ZF** должен быть сброшен в 0. Установка **ZF** в 1 говорит, что в результате вычитания двух чисел получился нулевой результат – значит, операнды равны.

Если **SF** = 0, то результат вычитания будет положительное число. Если из большего числа (op1) вычесть меньшее (op2) и результат – положительный, то op1 и op2 одновременно либо больше, либо меньше нуля. Значит, OF всегда сброшен в 0. Например, если op1 = -5, op2 = -7:

-5 - (-7) = 2 (SF = 0, OF = 0)

/ если op1 = 15, op2 = 9:

15 - 9 = 6 (SF=0, OF=0)

Флаг знакового переполнения OF будет установлен в 1, например, в таком случае: op1 = -5, op2 = 125. Значит, сочетание SF = 0, OF = 1 не соответствует тому, что первое число больше второго.

Если SF = 1, то после вычитания будет отрицательное число. Но если от большего (op1) отнималось меньшее (op2) и результат меньше нуля, значит, в вычитании есть ошибка знакового переполнения. Таким образом, OF должен быть установлен в 1. Получается, есть только две комбинации флагов для истинности условия op1 > op2: 1) OF=0, ZF=0, SF=0 и 2) OF=1, ZF=0, SF=1.

Команда замены по результату сравнения cmpxchg (compare and exchange – сравнить и заменить):

**cmpxchg приемник, источник**

Она сравнивает значение приемника со значением аккумулятора (al, ax, eax). Если приемник равен аккумулятору (флаг ZF = 1), то в приемник загрузили значение источника. Иначе (ZF = 0), то в al/ax/eax загрузили содержимое приемника. В зависимости от результата вычитания из приемника значения аккумулятора изменяются флаги CF, PF, AF, ZF, SF, OF. Приемником может быть регистр / ячейка памяти, источником – только регистр. Пример:

**.486**

**op1 db ?** ; определение 1 байтной переменной op1

…

mov al, 0ffh ; в аккумуляторе число 255

mov op1, 10 ; занесение в op1 (приемник) значения 10

mov bl, 2 ; занесение в регистр bl (источник) значения 2

cmpxchg op1, bl ; сначала значения op1 и al сравниваются; так как

; ZF = 0 (числа не равны), то в al загрузили содержимое

; приемника (op1), то есть после выполнения cmpxchg

; в al будет число 10

…

mov al, 5 ; занесение в аккумулятор числа 5

mov cl, 5 ; занесение в приемник числа 5

mov bl, 2 ; занесение в источник числа 2

cmpxchg cl, bl ; содержимое cl сравнивают с al, так как результат

; вычитания = 0, в приемник cl занесли значение

; источника bl (cl = 2).

Команда **cmpxchg8b** (compare and exchange 8 bytes – сравнить и заменить 8 байт):

**cmpxchg8b приемник**

Она сравнивает 8 байтное значение, лежащее в **edx:eax** с 8 байтным приемником (приемником может быть только ячейка памяти). Если эти значения равны, то устанавливают флаг ZF в 1 и в приемник загрузили 8 байт из пары регистров **ecx:ebx**. Иначе (если приемник и **edx:eax** не равны) флаг ZF сбрасывается в 0 и в пару **eax:edx** занесли содержимое приемника. В регистре флагов меняется только **ZF**, остальные флаги не меняются.

---

Часть 9

**8.1. КОМАНДЫ РАБОТЫ С РЕГИСТРОМ ФЛАГОВ.**

**8.1.1. Команды установки и сброса отдельных флагов.**

Выполнение арифметических команд, команд сравнения и сдвигов, строковых команд непосредственно связано с установкой / получением определенных флажков из регистра флагов. Хотя регистр флагов программно недоступен, можно установить и сбросить в нем отдельные биты.

Команда **clc** (clear carry flag–очистить флаг переноса) сбрасывает флаг CF в 0:

**clc**

Команда stc (set carry flag – установить флаг переноса) устанавливает флаг **CF** в 1:

**stc**

Команда **cmc** (complement carry flag – дополнить флаг переноса) инвертирует флаг **CF**:

**cmc**

Команда cld (clear direction flag – очистить флаг направления) сбрасывает флаг DF в 0:

**cld**

После выполнения команды **cld** индексные регистры (esi/si и edi/di), адресующие строку в строковых командах, будут увеличиваться.

Команда **std** (set direction flag – установить флаг направления) устанавливает флаг DF в 1:

**std**

После выполнения команды **std** индексные регистры (esi/si и edi/di), адресующие строку в строковых командах, будут уменьшаться.

Команда **cli** (clear interrupt enable flag – очистить флаг разрешения прерываний) сбрасывает флаг IF в 0:

**cli**

После выполнения команды cli МКпроцессор не будет реагировать ни на какие внешние (аппаратные) прерывания, кроме сигналов об отключении питания и ошибке памяти. Но на программные прерывания, то есть которые вызываются командой **int**, выполнение команды **cli** никак не влияет.

Команда **sti** (set interrupt enable flag – установить флаг разрешения прерываний) устанавливает флаг IF в 1:

**sti**

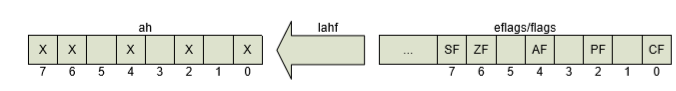
После выполнения команды sti МКпроцессору разрешается реагировать на внешние прерывания.

**8.1.2. Сохранение и загрузка регистра флагов.**

Для работы с регистром флагов не на уровне бит, а на уровне байт, есть 4 команды: pushf и popf, sahf и lahf. Команды pushf и popf соответственно заносят в стек и извлекают из стека весь регистр флагов. Команды lahf и sahf дают доступ только к младшему байту этого регистра. Команда lahf (load ah into flags – загрузить в регистр ah флаги):

**lahf**

Данная команда загружает младший байт регистра flags в регистр ah. Таким образом, становятся известны флаги SF, ZF, AF, PF и CF:



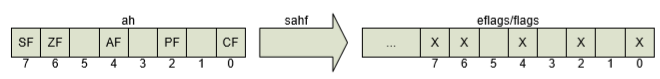
значения бит 1, 3, 5 после выполнения команды lahf в регистре ah не определены.

Для установки набора бит в регистре flags – команда sahf

(store ah into flags – сохранить регистр ah в регистре флагов):

**sahf**

Эта команда копирует биты 7, 6, 4, 2 и 0 из регистра ah в младший байт регистра флагов. Таким образом, заменяются предыдущие значения флагов SF, ZF, AF, PF и CF:



---

Часть 10

**8.2. КОМАНДЫ УСТАНОВКИ БАЙТА ПО УСЛОВИЮ.**

**8.2.1. Команды установки байта по условию setcc.**

Команды setcc (set byte on cc – установить байт в зависимости от условия cc):

**setcc приемник**

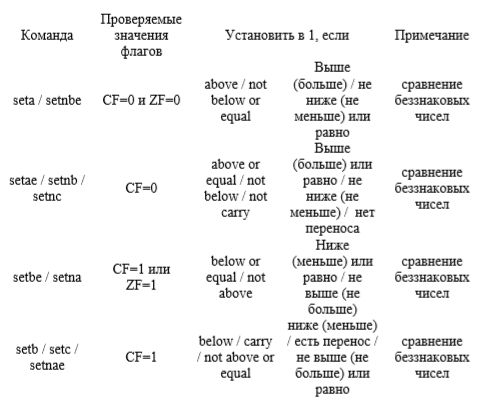
**setcc** – это обобщенное задание данных команд. Вместо окончания команды cc подставляется требуемое проверяемое условие.

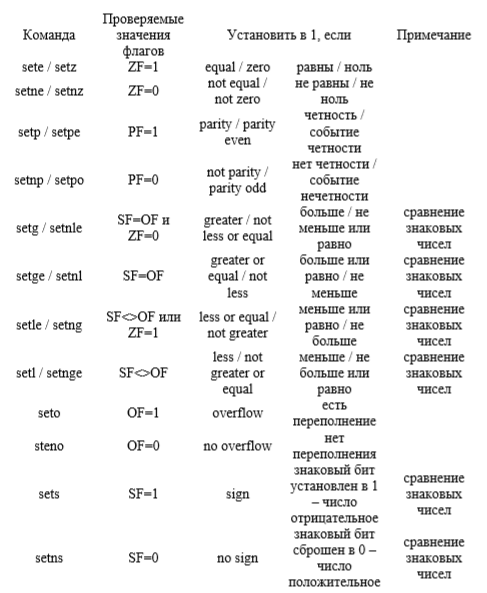
Например, **setge** (больше / равно) / **setnz** (флаг ZF=0). Команда установки байта по условию проверяет свое условие, и если оно истинно, то заносит в приемник значение 1, иначе (проверяемое условие ложно) – 0. Приемником может быть только 8 битный регистр / 8 битная ячейка памяти. Выполнение команды не влияет на состояние регистра флагов.

В табл 8.1 – полный перечень команд **setcc** (через символ ‘/’ приводятся

команды-синонимы).

Табл 8.1. Команды установки байта по условию setcc.





Пример:

|  |  |  |
| --- | --- | --- |
| mov ax, -500 | ; в регистр ax занесли -500 |  |
| mov cx, 300 | ; в регистр cx занесли 300 |  |
| cmp ax, cx | ; сравнение ax и cx |  |
| setg bl | ; так как число со знаком в ax не больше, чем число со  ; знаком в cx, то в bl занесли 0 (setg – установить, если больше | |

---

Часть 11

**8.3. СПЕЦИАЛЬНЫЕ КОМАНДЫ.**

**8.3.1. Команда пустой операции nop.**

Для реализации пустой операции – команды nop (no operation

– нет операции):

**nop**

Эта команда не делает никаких действий (команда холостого хода), кроме увеличения счетчика адреса (значение регистра eip/ip). Её размер – 1 байт; она применяется в двух случаях:

1) организация циклов задержки (используют очень редко); и

2) резервирование места в памяти (используют транслятором, если заранее неизвестно, какой будет размер формируемой команды, / для выравнивания в памяти на какую-то границу).

Синоним команды **nop** – это команда **xchg** **ax, ax** (машинный код команд nop и xchg ax, ax 1 и тот же).

**8.3.2. Команда останова hlt.**

Для останова МКпроцессора - команда hlt (halt – остановка):

**hlt**

Выполнение hlt приводит к прекращению выполнения всех команд и МКпроцессор переводится в состояние останова. Но если прерывания не были запрещены в момент останова МКпроцессора, то они продолжают восприниматься и управление передается обработчику прерываний, выполняется подпрограмма обработчика. После выхода из этой подпроги управление передается на следующую за **hlt** команду, и МКпроцессор возобновляет нормальную работу. Команда hlt – последняя в последовательности, которая завершает работу системы.

---

Часть 12

**9.1. ЛОГИЧЕСКИЕ КОМАНДЫ.**

**9.1.1. Общие сведения.**

Логические команды нужны для реализации логических операций – то есть операций над значениями “истина” (1) и “ложь” (0). Логические команды МКпроцессора позволяют реализовать такие операции:

− логическое умножение (и);

− логическое сложение (/);

− логическое исключающее / (сложение по модулю 2); − логическое отрицание (не).

**9.1.2. Логические команды (команды логических операций).**

Команда **and** (and – и) реализует побитое логическое умножение:

**and приемник, источник**

Команда **and** выполняет поразрядное логическое умножение над значениями приемника и источника и заносит результат в приемник. Биты результата устанавливаются в соответствии с табл 9.1. Команда **and** устанавливает результат бит в 1, если биты источника и приемника установлены в 1.

Таблица 9.1. Выполнение команды and

|  |  |  |
| --- | --- | --- |
| бит a | бит b | a and b |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Приемник - ячейка памяти / регистр, источник – ячейка памяти, регистр / непосредственный операнд. После выполнения команды **and** в регистре флагов изменяются флаги: CF и OF всегда сбрасывают в 0, значение AF не определено, флаги PF, ZF и SF устанавливают в соответствии с результатом. Пример:

mov al, 10000111b ; в al занесли число 135 = 10000111b

and al, 01000110b ; после выполнения and в al будет число 6 = 00000110b

Команда or реализует побитое логическое сложение:

**or приемник, источник**

Команда **or** выполняет поразрядное логическое сложение над значениями приемника и источника и заносит результат в приемник. Биты результата устанавливают в соответствии с табл 9.2. Команда **or** устанавливает в 0 бит результат только, если одновременно биты обоих установлены в 0.

Табл 9.2. Выполнение команды or

|  |  |  |
| --- | --- | --- |
| бит a | бит b | a or b |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Приемник - ячейка памяти / регистр, источник – ячейка памяти, регистр / непосредственный операнд. После выполнения команды or в регистре флагов изменяют флаги: CF и OF всегда сбрасывают в 0, значение AF не определено, флаги PF, ZF и SF устанавливают в соответствии с результатом. Пример:

mov al, 10000111b ; в al занесли число 135 = 10000111b

or al, 01000110b ; после выполнения or в al будет число 199=11000111b

Команда **xor** (exclusive or – исключающее /) - побитое логическое исключающее /:

**xor приемник, источник.**

Команда **xor** выполняет поразрядное логическое исключающее / над значениями приемника и источника и заносит результат в приемник. Биты результата устанавливаются в соответствии с таблицей 9.3. Команда xor устанавливает в 1 бит результат только в тех случаях, когда в 1 установлен бит только 1 го из операндов.

Табл 9.3. Выполнение команды xor

|  |  |  |
| --- | --- | --- |
| бит a | бит b | a xor b |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Приемник - ячейка памяти / регистр, источник – ячейка памяти, регистр / непосредственный операнд. После выполнения команды xor в регистре флагов изменяются такие флаги: CF и OF всегда сбрасывают в 0, значение AF не определено, флаги PF, ZF и SF устанавливаются в соответствии с результатом. Команду xor удобно использовать для обнуления значения операнда. Пример:

|  |  |
| --- | --- |
| mov al, 10000111b | ; в al занесли число 135=10000111b |
| xor al, 01000110b  … | ; после выполнения or в al будет  ; число 193=11000001b |
| xor cx, cx | ; самое быстрое обнуление значения, работает  ; быстрее, чем **and cx, 0** |

Команда **not** реализует логическое отрицание:

**not приемник**

Команда not инвертирует все биты приемника, результат остается в приемнике. Биты результата устанавливают в соответствии с табл 9.4 : 0 заменяют на 1, 1 – на 0.

Таблица 9.4. Выполнение команды not

|  |  |
| --- | --- |
| Бит a | not a |
| 0 | 1 |
| 1 | 0 |

Приемник - ячейка памяти / регистр. Выполнение команды not не влияет на состояние флагов. Пример:

mov al,10000111b ; в al занесли число 135=10000111b

not al ; после выполнения not в al будет число 120=01111000b

Кроме команды **and** еще одна команда логического умножения – команда test (test – проверить):

**test приемник, источник**

Команда test выполняет поразрядное логическое умножение, но результат никуда не заносит – только модифицирует флаги (CF и OF всегда сбрасывают в 0, значение AF не определено, флаги PF, ZF и SF устанавливают в соответствии с результатом). Приемник - ячейка памяти / регистр, источник – ячейка памяти, регистр / непосредственный операнд. Пример:

test bl,01000000b ; побитое логическое умножение bl и 64 jnz l1 ; если в bl установлен в 1 шестой бит (нумерация – с

; нулевого), то произойдет переход на метку l1 (так

; как флаг ZF будет равен 0)

… l1:

…

---

Часть 13

**9.2. КОМАНДЫ СДВИГОВ.**

# 9.2.1. Общие сведения.

Команды сдвигов нужны для сдвига (перемещения) всех бит операнда на некоторое число бит; это число называется **счетчиком** сдвигов. Биты можно сдвигать влево / вправо. 3 вида команд сдвигов:

1) арифметические сдвиги (для деления и умножения знаковых чисел на числа, кратные 2);

2) логические сдвиги (в логических преобразованиях и без-знаковых операциях умножения и деления на числа, кратные 2; не учитывают знакового бита числа);

3) циклические сдвиги (выдвигаемые биты не теряются, как в логических и арифметических сдвигах, а задвигаются с другой стороны).

# 9.2.2. Арифметические сдвиги.

Команда sal (shift arithmetic left – сдвинуть арифметически влево) реализует арифметический сдвиг влево:

**sal приемник, счетчик**

CF приемник

0

ст. мл.

Содержимое приемника сдвигают влево (в сторону старших бит) на количество бит, определяемое значением счетчика. При этом выдвигаемые (старшие) биты теряются, но последний выдвинутый бит хранится во флаге CF. С правой стороны (со стороны младших бит) вдвигают 0. В качестве счетчика сдвига используют:

1) значение регистра **cl**, при этом диапазон сдвига определяется всеми восьмью битами этого регистра; также можно использовать непосредственный операнд, значение которого = 1 (реализуется сдвиг на 1 бит);

2) значение регистра **cl**, при этом для уменьшения времени выполнения команды диапазон сдвига определяется только пятью битами этого регистра (операнд-приемник можно сдвинуть максимум на 31 бит); также можно использовать непосредственный операнд.

Приемник - ячейка памяти / регистр. После выполнения **sal** изменяют флаги: CF (содержит последний выдвинутый бит), PF, AF (значение AF не определено), ZF, SF и OF (если идет сдвиг не на 1 бит, то значение OF не определено; иначе OF = 0 если знаковый бит приемника не был изменен командой **sal**, и OF=1 если знаковый бит приемника был изменен). команда sal и команда логического сдвига влево **shl** эквивалентны (команды-синонимы). Пример:

|  |  |
| --- | --- |
| mov ax, 00000011b | ; в ax – число 3 |
| mov cl, 3 | ; счетчик сдвигов |
| sal ax, cl  …  .186 | ; арифметический сдвиг влево на количество  ; бит, определяемое значением в cl после  ; выполнения sal в ax – число 11000b=24 |
| mov ax, 00000011b | ; в ax – число 3 |
| sal ax, 3  …  .386 | ; арифметический сдвиг влево содержимого ax на 3 бита |
| mov eax, 1 | ; занесение в регистр eax значения 1 |
| mov cl, 33 | ; в cl занесли число 33=100001b – счетчик sal сдвигов |
| eax, cl | ; арифметический сдвиг влево на 1 бит (так как sal |

; использует только младшие 5 бит регистра cl),

; после выполнения в eax – число 2.

Сдвиг числа на одну позицию (1 бит) влево равносилен умножению этого числа на 2. Если величина сдвига больше единицы, число умножили на 2, возведенное в степень, равную содержимому счетчика сдвигов. Например, сдвиг влево на 4 бита равносилен умножению на 24 = 16.

Команда **sar** (shift arithmetic right – сдвинуть арифметически вправо) реализует арифметический сдвиг вправо:

**sar приемник, счетчик**

приемник CF

ст. мл.

Содержимое приемника сдвигается вправо на количество бит, определяемое значением счетчика. При этом младшие биты теряются, но последний выдвинутый бит хранится во флаге **CF**. С левой стороны вдвигается знак операнда (дублируется старший бит). Таким образом, положительные знаковые числа остаются положительными, а отрицательные – отрицательными. Приемник и счетчик в команде **sar** аналогичны приемнику и счетчику команды **sal**. После выполнения sar изменяются флаги CF (содержит последний выдвинутый бит), PF, AF (значение AF не определено), ZF, SF и OF.

Выполнение команды **sar** соответствует делению числа со знаком на число 2 в степени счетчика. Например, сдвиг впаво на 3 бита равносилен делению на 23 = 8. **sar** округляет все положительные числа к 0, а отрицательные от 0 (команда idiv округляет все числа к 0). Пример:

|  |  |
| --- | --- |
| mov al, -1 | ; делимое = -1 |
| sar al, 1  … | ; в al – число -1 |
| mov ax ,-1 | ; делимое = -1 |
| mov cl, 2 | ; делитель равен 2 |
| idiv cl | ; в al – число 0 |

Реализация операций умножения и деления с помощью сдвигов намного эффективнее, чем командами mul, imul, div, idiv.

# 9.2.3. Логические сдвиги.

Команда **shl** (shift logical left – сдвинуть логически влево) реализует логический сдвиг влево:

**shl приемник, счетчик**

CF приемник

0

ст. мл.

Команда **shl** равносильна команде арифметического сдвига **sal** (синонимами).

Команда shr (shift logical right – сдвинуть логически вправо) реализует логический сдвиг вправо:

**shr приемник, счетчик**

приемник CF

0

ст. мл.

Содержимое приемника сдвигается вправо на количество бит, определяемое значением счетчика. При этом младшие биты теряются, но последний выдвинутый бит хранится во флаге **CF**. С левой стороны вдвигают нули. Приемник и счетчик в команде shr аналогичны приемнику и счетчику в команде **sal**. После выполнения **shr** изменяют флаги **CF** (содержит последний выдвинутый бит), PF, AF (значение AF не определено), ZF, SF и OF. Выполнение команды **shr** соответствует делению без-знакового числа на число 2 в степени счетчика. Пример:

mov al, 10000101b ; в al занесли 10000101b, что соответствует беззнаковому

; 133 или знаковому -123

mov cl,2 ; счетчик сдвигов

shr al,cl ; логический сдвиг вправо, после выполнения в al будет

; 00100001b=33.

Две команды для сдвига повышенной точности. Команда shld (shift left double – двойной сдвиг влево) реализует сдвиг повышенной точности влево:

**shld операнд1, операнд2, счетчик.**

CF операнд1 операнд2

0

ст. мл. ст. мл.

Команда shld сдвигает содержимое операнд1 влево на количество бит, определяемое значением счетчика. Выталкиваемые из операнд1 биты теряются, но последний вытолкнутый остается в CF. Справа в операнд1 вдвигаются биты, выдвигаемые из операнд2, но после выполнения команды содержимое операнд2 остается таким же, как и было до начала команды, то есть идет псевдо-сдвиг влево значения операнд2. В качестве операнд1 могут быть 16 битные / 32 битные регистр / ячейка памяти, в качестве операнд2 – только 16 битный / 32 битный регистр; счетчик задают / значением в cl / как непосредственный операнд. После выполнения **shld** изменяют флаги: CF (содержит последний выдвинутый из операнд1 бит), PF, ZF, SF, состояния AF и OF не определены. Команду **shld** удобно использовать для сдвига влево 32 разрядных (при объединении слов, то есть операнд1 и операнд2 2 байтные) и шестидесяти4хразрядных (при объединении двойных слов, то есть операнд1 и операнд2 4хбайтные) значений. Пример:

.386

op1 dw ? ; определение переменной op1

… ; сдвиг 32 битного числа влево на 2 бита, старшая часть которого

; находится в op1, младшая - в bx

mov op1, 1110000b ; старшая часть сдвигаемого числа

mov bx,1100000000000000b ; младшая часть сдвигаемого числа

mov cl, 2 ; счетчик сдвигов

shld op1, bx, cl ; сдвиг влево повышенной точности; в результате в op1

; будет число 111000011b, содержимое bx изменено не будет

shl bx,cl ; коррекция результата – действительный (а не псевдо) сдвиг

; значения в bx, теперь в bx лежит 0.

Команда **shrd** (shift right double – двойной сдвиг вправо) - сдвиг по-вышенной точности вправо:

**shrd операнд1, операнд2, счетчик**

операнд2 операнд1 CF

0

ст. мл. ст. мл.

Команда shrd сдвигает содержимое операнд1 вправо на количество бит, определяемое значением счетчика. Выталкиваемые из **операнд1** биты теряют, но последний вытолкнутый оставляют в **CF**. Слева в операнд1 вдвигают биты, выдвигаемые из операнд2 (псевдо-сдвиг вправо значения операнд2). Операнд1, операнд2 и модифицируемые флаги в команде **shrd** аналогичны в команде **shld**. Команду **shrd** удобно использовать для сдвига вправо 32 разрядных (при объединении слов, то есть операнд1 и операнд2 2 байтные) и 64хразрядных значений (при объединении двойных слов, то есть операнд1 и операнд2 4хбайтные).

**9.2.4. Циклические сдвиги.**

Команда **rol** (rotate left – вращать влево) реализует циклический сдвиг влево:

**rol приемник, счетчик**

CF приемник

ст

.

мл

.

Команда **rol** сдвигает содержимое приемника влево на количество бит, определяемое счетчиком. Справа (в младший бит) вдвигают старший (выдвигаемый) бит. Значения выдвигаемых бит заносят в **CF**. Приемник - регистр / ячейка памяти. В качестве счетчика сдвига используют:

1) значение регистра **cl**. Диапазон сдвига определяют всеми восьмью битами этого регистра; также можно использовать непосредственный операнд, значение которого = 1 (идет сдвиг на 1 бит);

2) значение регистра cl. Для уменьшения времени выполнения команды диапазон сдвига определяют только младшими пятью битами этого регистра (операнд приемник можно сдвинуть максимум на 31 бит); также можно использовать непосредственный операнд.

Команда **rol** модифицирует флаги CF и OF. Пример:

mov al, 10000001b ; занесение в al числа 10000001b

rol al, 1 ; циклический сдвиг влево на 1 бит, после выполнения

; rol в al – число 00000011b.

Команда **ror** (rotate right – вращать вправо) - циклический сдвиг вправо:

**ror приемник, счетчик**

приемник CF

0

ст

.

мл

.

Команда ror сдвигает содержимое приемника вправо на количество бит, определяемое счетчиком. Слева (в старший бит) вдвигают младший (выдвигаемый) бит. Значения выдвигаемых бит заносят в **CF**. Приемник и счетчик в команде ror аналогичны в команде rol. При выполнении ror изменяют флаги CF и OF.

Команда **rcl** (rotate through carry left – вращать через перенос влево) предназначена для циклического сдвига через флаг CF влево:

**rcl приемник, источник**

CF приемник

ст

.

мл

.

Команда **rcl** сдвигает содержимое приемника влево на количество бит, определяемое счетчиком. Справа (в младший бит) вдвигается бит из флага **CF**, затем значение выдвигаемого (младшего) бита приемника занесли в **CF**. Приемник и счетчик в команде **rcl** аналогичны в команде rol. При выполнении rcl изменяют флаги **CF** и **OF**. Пример:

**stc**  ; установка в 1 флага CF

**mov bl, 00111000b** ; инициализация bl; команда mov не модифицирует

|  |  |  |
| --- | --- | --- |
|  |  | ; флаги, поэтому CF не будет изменен |
| **mov cl, 3** |  | ; счетчик сдвигов |
| **rcl bl, cl** |  | ; циклический сдвиг через перенос влево. В результате в  ; bl будет число 11000100b и флаг CF будет установлен в 1 |

Команда rcr (rotate through carry right – вращать через перенос вправо) - для циклического сдвига через флаг CF вправо:

**rcr приемник, источник**

приемник CF

0

ст

.

мл

.

Команда rcr сдвигает содержимое приемника вправо на количество бит, определяемое счетчиком. Слева (в старший бит) вдвигают бит из флага CF, затем значение выдвигаемого (младшего) бита приемника заносят в CF. Приемник и счетчик в команде rcr аналогичны приемнику и счетчику в команде rol. При выполнении rcr изменяются флаги CF и OF.

------

Часть 14

**9.3. КОМАНДЫ МАНИПУЛЯЦИИ ОТДЕЛЬНЫМИ БИТАМИ.**

**9.3.1. Команды поиска установленного в единицу бита.**

Команды, позволяющие манипулировать отдельными битами: **bsf** и **bsr** - для поиска первого установленного в 1 бита.

Команда **bsf** (scan bit forward – сканирование бита вперед) - для поиска вперед первого установленного в 1 бита операнда:

**bsf приемник, источник.**

Команда **bsf** сканирует биты источника, начиная с младшего (нулевого), в поисках первого бита, установленного в 1. Если такой бит не найден (источник равен 0), то возводят флаг нуля (ZF=1); Тогда содержимое приемника не меняют. Если же источник не равен нулю, то в приемник заносят позицию первого установленного в 1 бита. После выполнения команды **bsf** значение ZF определяется источником, значения флагов CF, PF, AF, SF и OF не определены. Приемник - 2 байтный / 4хбайтный регистр; источник – 2 байтный / 4хбайтный регистр, 2 байтная / 4хбайтная ячейка памяти. Пример:

**.386**

**mov bx, 1000100b**  ; инициализация bx

**bsf ax, bx** ; после выполнения bsf в ax будет находиться число 2.

Команда **bsr** (scan bit reverse – сканирование бита назад) - для поиска назад первого установленного в 1 бита операнда:

**bsr приемник, источник**

Команда **bsr** сканирует биты источника, начиная со старшего, в поисках первого бита, установленного в 1. Если такой бит не найден (источник равен 0), то возводят флаг нуля (ZF=1); Тогда содержимое приемника не меняют. Если источник не равен нулю, то в приемник заносят позицию первого установленного в 1 бита, позицию высчитывают, начиная с младшего бита. Приемник, источник и модифицируемые флаги команды bsr аналогичны в команде bsf. Пример:

**mov dx, 0100000010001001b** ; инициализация регистра dx

**bsr ax, dx** ; после выполнения bsr в регистре ax будет число 14

**9.3.2. Команды проверки и установки отдельных бит.**

Команды bt, bts, btr, btc – для проверки заданного бита и, возможно, установки его значения.

Команда **bt** (test bit – проверить бит) – для проверки значения бита в определенной позиции:

**bt приемник, источник**

По команде bt значение бита приемника, позиция которого задана в источнике, занесли во флаг CF. Нумерация позиций с 0.

Команда **bts** (test bit and set – проверить бит и установить бит) – для проверки значения бита в определенной позиции и установки этого бита в 1:

**bts приемник, источник**

По команде bts значение бита приемника, позиция которого задана в источнике, заносят во флаг CF. Затем в этом бите приемника устанавливают 1.

Команда **btr** (test bit and reset – проверить бит и сбросить бит) – для проверки значения бита в определенной позиции и сброса этого бита в 0:

**btr приемник, источник**

По команде btr значение бита приемника, позиция которого задана в источнике, заносят во флаг CF. Затем этот бит приемника сбрасывают в 0.

Команда **btc** (test bit and compliment – проверить бит и дополнить бит) – для проверки значения бита в определенной позиции и инвертирования этого бита:

**btc приемник, источник**

По команде btc значение бита приемника, позиция которого задана в источнике, заносят во флаг CF. Затем этот бит приемника инвертируют. Пример:

.386

mov dx, 01001b ; занесение в dx числа 1001b

bt dx, 3 ; проверка третьего бита, флаг CF устанавливают в 1

…

mov dx, 01001b ; занесение в dx числа 1001b

mov bx, 2 ; номер проверяемой позиции

bts dx, bx ; после проверки флаг CF=0 (так как второй бит сброшен в

; 0) и установка второго бита в dx, таким образом в dx

; будет число 1101b.

В командах **bt, bts, btr и btc** приемник – 2 байтный / 4хбайтный регистр, а также 2 байтная / 4хбайтная ячейка памяти. Источник – 2 байтный / 4хбайтный регистр / непосредственный операнд. Данные команды модифицируют только флаг CF.

---

Часть 15

**10.1. КОМАНДЫ ПЕРЕХОДОВ.**

# 10.1.1. Общие сведения.

Команды переходов передают управление в некоторую точку проги без сохранения адреса возврата. Осуществляя переход, данные команды изменяют значение регистра указателя инструкций eip/ip (используют ли 2 байтный ip / 4хбайтный eip – зависит от режима адресации – use16 / use32) и, возможно, значение кодового регистра cs. Будет ли изменено содержимое cs, определяется видом перехода. Есть дальний (**far**) и ближний (**near**) переходы. Far-переход изменяет значение кодового регистра и регистра указателя инструкций; такая передача управления – межсегментная. Near-переход влияет лишь на содержимое eip/ip; ближний переход делает передачу управления внутри сегмента. Команды переходов никогда не изменяют регистр флагов.

# 10.1.2. Команда безусловного перехода jmp.

Команда jmp (jump – прыгать) передает управление по некоторому адресу без сохранения адреса возврата (то есть старые значения регистров eip/ip и, возможно, cs теряются):

**jmp [модификатор] операнд**

Есть несколько видов данной команды передачи управления, и какая именно команда будет порождена транслятором, определяется необязательным полем модификатор и полем операнд. Если собрать эти команды в группы по виду перехода, то команда **jmp** может реализовать:

1) короткий относительный внутрисегментный переход (относительный – потому что в формате машинной команды хранится относительное смещение, / расстояние, от команды jmp до метки);

2) относительный переход в пределах всего сегмента;

3) косвенный переход в пределах сегмента;

4) прямой межсегментный переход (прямой – потому что в формате машинной команды содержится полный адрес перехода);

5) косвенный межсегментный переход.

Для реализации короткого внутрисегментного перехода есть такая команда:

**jmp short метка**

**jmp метка**

Данная команда выполняет короткий (short) относительный переход типа **near**. Его диапазон – -128..127 байт. Операнд данной команды – метка, но транслятор преобразует ее в 1 байтное относительное смещение, и уже относительное смещение записывают в машинный формат команды. Относительное смещение вычисляют так: от значения метки (от значения смещения первого байта команды, на которую указывает метка) отнимается значение смещения команды, следующей за jmp. Относительное смещение трактуют как число со знаком – отсюда возможность перехода как вперед, так и назад. Когда произойдет выполнение команды в программе, вычисленное транслятором относительное смещение будет прибавлено к содержимому eip/ip; и таким образом, произойдет относительный переход. Пример:

|  |  |
| --- | --- |
| L1: | ; метка, на которую передается управление |
| … | ; расстояние от метки до следующей за jmp команды не  ; превышает 128 байт |
| jmp short L1  … | ; короткий переход на метку L1 |
| jmp L2 | ; короткий переход на метку L2, переход будет коротким из-за  ; того, что расстояние от команды jmp до L2 не превышает  ; 127 байт; если бы расстояние было вне диапазона -128..127,  ; то транслятор сформировал бы команду относительного  ; перехода в пределах всего сегмента |
| … | ; общий размер кода команд не превышает 127 байт |
| L2: | ; метка, на которую передается управление. |

Для реализации относительного перехода в пределах всего сегмента используют команду:

**jmp near ptr метка**

**jmp метка**

Эта команда делает относительный переход в любую точку внутри сегмента. Модификатор near ptr указывать необязательно. Как и для короткого перехода, формат машинной команды содержит относительное смещение, которое прибавляют во время выполнения проги к указателю инструкций eip/ip. Только относительное смещение теперь – 2 байтная величина при использовании 16 битной (use16) и 4хбайтной при использовании 32 битной (use32) адресации внутри кодового сегмента. В первом случае – организация переходов в диапазоне -32768..32767 байт; во втором – в пределах 4Гб. Пример:

L1:

… ; расстояние от метки до следующей за jmp команды

; превышает 128 байт

jmp L1 ; относительный переход на метку L1

…

jmp near ptr L2

… ; расстояние от метки L2 до следующей за jmp команды

; превышает 127 байт

L2 label near ; задание ближней метки (метки типа near), такая

; запись аналогична L2:

…

jmp near ptr L1 ; относительный переход на метку L1

Для реализации внутрисегментного косвенного перехода – команда:

**jmp источник**.

Источник – регистр / ячейка памяти, размер источника – 2 байта для 16 битного режима адресации и 4 байта для 32 битного режима адресации. При выполнении данной команды будет осуществлен косвенный переход типа near. Косвенным он будет потому, что адрес перехода находится не в самой команде, а в регистре / в памяти. Фактически, команда внутрисегментного косвенного перехода выполняет копирование данных, которые находятся в источнике, в регистр указателя инструкций eip/ip.

Пример для 16 битного режима адресации:

mem1 dw 1, 2, offset L1, 4, 5

…

mov ax,offset L1 ; в ax – смещение метки L1

jmp ax ; содержимое ip заменяют содержимым ax, переход на метку L1

…

L1:

…

mov si, 4

jmp mem1[si] ; в ячейке по адресу mem1+(si) лежит адрес перехода

; в данном случае идет переход на метку L1

Для реализации прямого межсегментного дальнего перехода – команда:

**jmp метка\_типа\_far.**

**jmp far ptr метка.**

Данная команда делает прямой межсегментный (/ дальний) переход. Формат машинной команды содержит полный адрес перехода – адрес начала сегмента, в котором описана принимающая управление метка, и смещение этой метки относительно начала сегмента. При выполнении команды значение смещения загрузили в регистр eip/ip, значение начала сегмента – в cs. Таким образом, управление идет на метку, заданную в команде. Если метка объявлена как имя\_метки label far, то модификатор far ptr можно не использовать. Пример:

|  |  |
| --- | --- |
| code segment  … |  |
| jmp L1  … | ; дальний переход на метку L1 |
| exit: | ; метка, на которую из другого сегмента можно |
| …  code ends  seg1 segment  … | ; передать управление, только указав модификатор far ptr |
| L1 label far  … | ; описание дальней метки (метки типа far) |
| jmp far ptr exit  …  seg1 ends | ; дальний переход на метку exit |

Для реализации межсегментного косвенного перехода – команда:

**jmp источник.**

Источник – это ячейка памяти, размер которой – 4 байта для 16 битного режима адресации и 6 байт для 32 битного режима адресации. Данная команда делает косвенный межсегментный переход. При 16 битном режиме адресации для нахождения адреса перехода 2 младших байта (младшее слово) источника загружают в регистр ip, а 2 старших (старшее слово) – в регистр cs. То есть, если по указанному в команде адресу лежит цепочка байт 12345678H (старший байт – 12h, младший – 78h), то регистр указателя инструкций станет иметь значение 5678H, в регистр cs будет записано 1234H. Например:

faraddr16 dw 0aabbh ; смещение

|  |  |
| --- | --- |
| dw 0ccddh  … | ; сегмент |
| jmp dword ptr faraddr16 | ; межсегментный переход, в ip – 0aabbh в cs – 0ccddh |

При 32 битном режиме адресации старшее слово источника загрузили в регистр cs, младшие 4 байта копируют в eip. Например:

|  |  |  |
| --- | --- | --- |
| faraddr32 dd 123 |  | ; смещение |
| dw 0bH  … |  | ; сегмент |
| jmp fword faraddr32 |  | ; дальний переход по адресу, |
|  |  | ; определяемую значением faraddr32 |

# 10.1.3. Команды переходов по условию.

Команды переходов по условию, / команды условных переходов, делают относительные переходы типа **near**. они выполняют передачу управления в зависимости установлены ли определенные флаги в регистре flags / нет. Такие команды 1 типны, поэтому для обозначения группы этих команд используют обозначение **jcc**. Это значит, что там, где запись “jcc”, вместо нее можно подставлять любую из команд, делающих переход в зависимости от состояния флагов (например, jba, jz, jpe). диапазон переходов – в промежутке от -128 до 127 байт. Сейчас возможна передача управления в пределах всего сегмента.

Также есть две команды условного перехода (jecxz и jcxz), которые выполняют передачу управления в зависимости от значения в регистре ecx/cx. Но эти переходы находятся в промежутке -128..127 байт, независимо от режима работы МКпроцессора.

Для передачи управления в зависимости от состояния определенных флагов используют команды jcc (jump on cc – перейти, если условие cc истинно):

**jcc метка**

При истинности проверяемого условия идет относительный внутрисегментный переход по адресу, на который указывает метка. Все команды данной группы анализируют содержимое регистра flags. Флаги устанавливают любыми арифметическими / логическими операциями. 3 вида команд **jcc** в зависимости от проверяемых флагов:

1) команда делает переход, если выполнено условие сравнения 2 чисел,

интерпретируемых как знаковые;

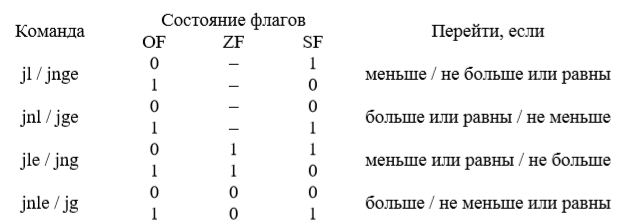
2) команда делает переход, если выполнено условие сравнения 2 чисел, интерпретируемых как беззнаковые;

3) переход по состоянию отдельных флагов.

Для проверки условия сравнения 2 чисел используют определенные сочетания флагов. Чтобы установить в регистре flags верные значения, используют команду сравнения **cmp** (реже – команду **sub**). Для установки отдельных флагов удобно использовать команду **test** (аналогична команде and, но не изменяет значение операнда).

В табл 10.1 – команды переходов по результату сравнения 2 знаковых чисел (двух чисел в дополнительном коде).

Табл 10.1. Команды переходов по результату сравнения 2 знаковых чисел (двух чисел в дополнительном коде).



Буква ‘g’ в названии команды означает “greater” (больше), ‘l’ – “less” (меньше); ‘e’ значит “equal” (равны), ‘n’ – “not” (отрицание). Через косую черту “/” обозначены эквивалентные команды; код таких команд 1аков, но для удобства используют несколько имен. Пример:

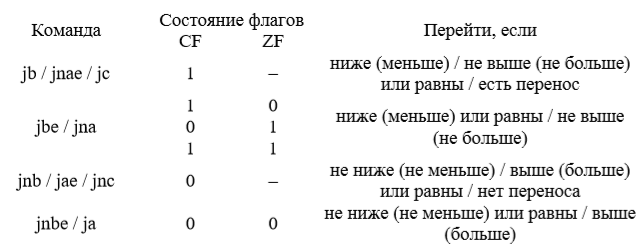
|  |  |
| --- | --- |
| **mov cl, 0feh** | ; в cl – число -2 |
| **cmp cl, 7dh** | ; значение в cl сравнивают с числом +125 |
| **jge L2** | ; перейти, если больше / =; в данном случае перехода нет |

…

L2:

В табл 10.2 – команды переходов, анализирующие флаги, установленные при сравнении без-знаковых чисел.

Табл 10.2. Команды переходов, анализирующие флаги.



Здесь буква ‘b’ в названии команды обозначает “below” (ниже), ‘a’ – “above” (выше), ‘c’ – “carry” (перенос). Пример:

|  |  |
| --- | --- |
| mov cl, 0feh | ; в cl – число 254 |
| cmp cl, 7dh | ; значение в cl сравнивается с числом +125 |
| jae L2 | ; беззнаковое число в cl > 125, поэтому идет переход на метку L2 |

…

L2:

Табл 10.3. Команды переходов по состоянию отдельных флагов.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | | | |  |
| Команда | Состояние флагов | | | | Перейти, если |
| OF | ZF | PF | SF |
| je / jz | – | 1 | – | – | = / 0 |
| jp / jpe | – | – | 1 | – | есть четность |
| jo | 1 | – | – | – | есть переполнение |
| js | – | – | – | 1 | есть знак (отрицательное число - ) |
| jne / jnz | – | 0 | – | – | Не = / не 0 |
| jnp / jpo | – | – | 0 | – | нет четности |
| jno | 0 | – | – | – | нет переполнения |
| jns | – | – | – | 0 | нет знака (положительное число + ) |

Для проверки отдельных флагов пользуются логическими командами – они работают быстрее, чем арифметические. То есть, например, вместо add al, 0 использовать test al, 0ffh. Пример:

L1:

…

mov dl, -98 ; занесение в dl числа -98=10011110b

and dl, 11111111b ; поразрядное и

js L1 ; перейти, если число в dl – отрицательное, идет переход на L1

Формат машинной команды **jcc** имеет относительное смещение, равное разности между смещением метки и смещением следующей за **jcc** команды. Для МКпроцессора относительное смещение 1 байтное, поэтому реализуют лишь короткие near-переходы. Пример:

cmp cx,3

jg L1 ; перейти, если больше

…

L1:

Если расстояние от команды jg до метки L1 больше 127 байт, то конструкция такая:

|  |  |
| --- | --- |
| cmp cx, 3 |  |
| jle continue | ; перейти, если < / = |
| jmp L1 continue:  …  L1: | ; безусловный переход |

Есть еще 1 вид команд jcc: их машинный формат имеет 2 байтное (для 16 битного режима адресации) / 4хбайтное (для 32 битного режима адресации) относительные смещения, поэтому такие команды реализуют относительные near-переходы в любую точку вну3 сегмента. Пример:

.80386

L1:

… ; расстояние от метки до следующей за jc команды не

; превышает 128 байт

jc L1 ; транслятор создает машинную команду jc

|  |  |  |
| --- | --- | --- |
| …  L2: |  |  |
| … |  | ; расстояние от метки до следующей за jc команды > 128 байт |
| jc L2 |  | ; создается команда условного перехода jc, машинный формат  ; которой имеет 2 байтное (/4хбайтное) относительное смещение |

Синтаксис таких команд такой же, как и у простых команд jcc. Транслятор сам порождает нужный код, анализируя смещение до метки.

Есть две команды перехода по состоянию регистра счетчика ecx/cx:

**jcxz метка.**

**jecxz метка.**

Команды делают относительные переходы типа **near**, их диапазон -128..127 байт. В отличие от **jcc**, для **jcxz** и **jecxz** диапазон не зависит от режима работы МКпроцессора, то есть в любом случае метка не может быть выше 127 и ниже 128 байт относительно этих команд.

Команда **jcxz** выполняет переход, если регистр **cx = 0**. Аналогично **jecxz** передает управление метке, если в **ecx** есть 0. На самом деле машинный код команд **jcxz** и **jecxz** аналогичен, просто в зависимости от режима адресации в формат машинной команды может включаться байт переопределения размера операнда, определяя, будет ли использоваться регистр cx / ecx. Пример:

|  |  |
| --- | --- |
| mov cx, 27 | ; счетчик цикла |
| startloop: | ; начало цикла |
| jcxz endloop | ; выход из цикла, если в cx находится 0 |
| sub cx, 3 | ; уменьшение значения в cx на 3 |
| … | ; тело цикла будет выполнено 9 раз |

jmp short startloop ; переход на начало цикла

endloop: ; конец цикла

---

Часть 16

**10.2. КОМАНДЫ УПРАВЛЕНИЯ ЦИКЛАМИ.**

# 10.2.1. Общие сведения.

При программировании часто возникает надость обеспечить не1 кратное выполнение некоторой последовательности команд проги, то есть организовать цикл. Есть 2 способа организации циклов:

1) использование команд переходов;

2) использование специальных команд управления циклами (/ командами организации циклов).

Независимо от режима работы МКпроцессора команды управления циклами реализуют лишь короткие относительные переходы. Это означает, что расстояние от метки до следующей за этими командами команды не должно превышать 127 байт, если идет переход вперед (метка описана ниже команды управления циклами), и 128 байт при переходе назад (метка описана выше команды управления циклами). Ни одна из команд управления циклами регистр флагов не меняет.

# 10.2.2. Команды управления циклами.

Команда **loop** (loop – петля) позволяет организовывать циклы, выход из которых идет только при обнулении значения в регистре ecx/cx (условие выхода – ecx/cx = 0):

**loop метка**

**loopw метка.**

**loopd метка.**

При выполнении loop идет декремент (уменьшение) на 1 содержимого регистра **ecx/cx** (ecx – если используют 32 битный режим адресации). Затем анализируют значение ecx/cx: если регистр содержит отличное от 0 значение, то идет переход на метку. Если же **ecx/cx** = 0, то выбирают следующую за loop команду. Команды **loopw** (loop word) и **loopd** (loop dword) – модификации команды **loop**. Они указывают явно, какой регистр должен использоваться в этой команде (loopw работает с регистром cx, loopd – с ecx). Например, если при 16 битном режиме адресации надо использовать регистр ecx, то команда loopd при трансляции будет преобразована в команду loop с префиксом размера операнда. Пример:

mov cx, 10 ; в счетчик цикла cx занесли число повторений

lp1:

… ; если в теле цикла значение cx не меняется, то

loop lp1 ; цикл будет выполнен 10 раз

Команда **loop** выполняет те же действия, что и последовательность **dec ecx/cx, jnz** метка. Но loop на 1 байт короче длины этих двух команд и выполняется быстрее, чем последовательность **dec** и **jnz**.

Команда **loope/loopz** (loop while zero/equal – цикл, пока равно) организует циклы, выход из которых идет при обнулении значения в регистре ecx/cx / как только флаг ZF будет сброшен в 0 (условие выхода – ecx/cx = 0 / ZF = 0):

**loope метка**

**loopz метка loopew метка loopzw метка looped метка loopzd метка**

Названия loope и loopz – синонимы, они определяют одну и ту же команду. По сути, **loope/loopz** – расширение команды **loop**. Как и loop, команда **loope/loopz** сначала уменьшает на 1 значение в регистре **ecx/cx**. Затем анализируют условие: если в ecx/cx отличное от 0 значение, а флаг нуля ZF установлен в 1, идет переход на указанную метку. Но если содержимое ecx/cx стало = 0, то независимо от флагов будет выход из цикла; аналогично, как только перед командой **loope/loopz** ZF будет сброшен в 0, независимо от значения в ecx/cx, будет выход из цикла. Команды **loopew** и **loopzw** работают с регистром **cx**, команды **looped** и **loopzd** – с регистром **ecx**; они являются модификациями команды **loope/loopz**. Пример:

; в программе идет многократный вызов процедуры, которая в

; случае возникновения ошибки помещает код ошибки в регистр ax;

; если процедура отработала нормально, то в ax будет возвращен 0:

…

|  |  |
| --- | --- |
| mov cx, 20 | ; счетчик числа повторений |
| lp1:  … | ; начало цикла |
| call myproc  test ax, 0ffffh | ; вызов процедуры, команда call передает управление  ; процедуре myproc, при выходе из процедуры  ; управление передается на следующую за call команду |
| loopz lp1 | ; цикл завершится, как только в ax будет значение,  ; отличное от 0, / как только счетчик cx станет = 0 |

Команда **loopne/loopnz** (loop while not zero/equal – цикл, пока не равно) позволяет организовать циклы, выход из которых идет при обнулении значения в регистре **ecx/cx /** как только флаг ZF будет установлен в 1 (условие выхода – ecx/cx = 0 / ZF = 1):

**loopne метка**

**loopnz метка loopnew метка loopnzw метка loopned метка loopnzd метка**

Команды loopne и loopnz – синонимы. Как и все команды управления циклами, **loopne/loopnz** первым делом делает декремент регистра **ecx/cx**.

Затем идет анализ значения в этом регистре на равенство 0 и сброс ZF в 0.

Если да (ecx/cx 0 и ZF = 0), то идет переход на указанную метку. Перехода нет, если ecx/cx = 0 (независимо от значения регистров флагов) / если флаг нуля ZF установлен в 1 (независимо от значения счетчика). Команды **loopnew** и **loopnzw** работают с регистром **cx**, команды **loopned** и **loopnzd** – с регистром **ecx**; это модификации команды **loope/loopz**. Пример:

mov cx, 10 ; счетчик цикла

lp1:

…

cmp al,27 ; цикл выполнится 10 раз / будет прекращен, когда

loopne lp1 ; значение в al станет равным 27.

# 10.2.3. Работа с командами управления циклами.

Во-первых, с помощью этих команд организуют циклы как с постусловием, так и с предусловием. Цикл с постусловием (сначала выполняют тело, затем анализируют условие):

mov cx, 100 ; счетчик числа повторений

L1:

…

loop L1

Цикл с предусловием (сначала анализируют условие, затем выполняют тело цикла):

mov cx, 10 ; счетчик числа повторений

|  |  |  |
| --- | --- | --- |
| L1:  loop L2 |  |  |
| jmp L3  L2: |  | ; если cx=0, то идет выход из цикла |
| … |  | ; тело цикла |
| jmp L1  L3: |  | ; возврат на оператор цикла (на команду loop) |

Во-вторых, если при первом выполнении loop счетчик cx имел 0, то тело будет выполнено 65536 раз (при работе с ecx в такой же ситуации тело будет выполнено 4294967296 раз). Например:

counter dw ?

…

mov cx, counter ; если в counter 0, то тело будет выполнено 65536 раз lp1:

…

loop lp1

Дело в том, что при выполнении команды **loop** сначала идет декремент регистра **ecx/cx**, и только затем проверка **ecx/cx** на равенство 0. А если от 0 отнять 1 и результат поместить в 2 байтную ячейку, то в нее будет помещено число 65535. Значит, в такой ситуации цикл выполнится 65536 раз, учитывая первое выполнение (когда cx = 0). Поэтому, если число повторений заранее не известно, и берется как значение какой-то переменной, то команды **jcxz/jecxz** используют перед меткой начала тела цикла. Пример:

counter dw ?

…

mov cx, counter

jcxz endloop ; переход на конец цикла при нулевом значении счетчика cx

startloop:

…

loop startloop

endloop:

Хотя каждая команда управления циклами уменьшает значение регистра **ecx/cx** на 1, регистр флагов не модифицируют. Поэтому если в цикле последняя команда изменяет флаги, то после loop можно применять условные переходы:

mov cx, 15

startlp:

…

add ax, 5 loop startlp

jp L1 ; переход на L1, ели установлен в 1 флаг четности PF

…

L1:

При использовании команд loop, loope/loopz, loopne/loopnz можно организовать вложенные циклы. Но так как все эти команды используют регистр ecx/cx, его содержимое надо временно сохранять и затем восстанавливать. Например:

|  |  |
| --- | --- |
| mov cx, 20  lp1: | ; счетчик внешнего цикла |
| … | ; \*\*\* внешний цикл |
| push cx | ; занесение в стек количества повторений внешнего цикла |
| mov cx, 10  lp2: | ; счетчик внутреннего цикла |
| …  loop lp2 | ; \*\* внутренний цикл |
| pop cx | ; извлечение из стека количества повторений внешнего цикла |

… ; \*\*\* внешний цикл

loop lp1

Значение счетчика хранить в другом регистре / в ячейке памяти. Но лучще использовать стек для временного хранения ecx/cx. Используя такую организацию циклов можно обеспечить любой уровень вложенности.

**=====================================================**

**=====================================================**

**11.1. КОМАНДЫ ОБРАБОТКИ СТРОК.**

# 11.1.1. Общие сведения.

Команды обработки строк / цепочечные команды – это отдельный класс команд МКпроцессора, которые обрабатывают последовательности одно-типных элементов, а такие последовательности называют **строками / цепочками**. Обрабатываемый элемент строки имеет размер байт, слово / двойное слово. С помощью данных команд пересылают строки (movs), сравнение строк (**cmps**), сканирование строк (**scas**), загрузка (**lods**) и сохранение (**stos**) строк, вводить и выводить строки из порта (ins) и в порт (outs).

Строка может быть источником / приемником. Всегда строка-источник адресуется парой регистров **ds:esi/si** (si – source index), причем в **esi/si** находится смещение текущего элемента строки. Приемник адресуется парой **es:edi/di** (di – destination index), в регистре di - смещение относительно es текущего элемента. Регистры si и di используют при 16 битном режиме адресации, esi и edi – при 32 битном режиме адресации. При выполнении команд обработки строк всегда идет авто-инкремент либо авто-декремент индексных регистров. Значения в этих регистрах изменяют на величину, равную размеру элемента строки в байтах. То есть, если идет обработка строк слов, то esi/si и edi/di будут изменять на 2. То, в какую сторону будут изменять значения индексных регистров, определяется флагом направления DF (direction flag). Если DF = 0, то esi/si и edi/di увеличивают; если DF = 1, то esi/si и edi/di уменьшают. Для управления этим флагом есть две команды – std (установить DF в 1) и cld (сбросить DF в 0).

При работе со строками почти всегда надо вызывать одну и ту же команду несколько раз. Для оптимизации этого процесса есть спец префиксы.

# 11.1.2. Префиксы повторения.

3 префикса повторения: **rep, repe/repz, repne/repnz**. Синтаксис построения команд с префиксами:

rep строковая\_команда

repe/repz строковая\_команда

repne/repnz строковая\_команда

Префикс **rep** (repeat – повторять) реализует повторение выполнения команды, пока значения регистра ecx/cx не = 0. Регистр **ecx** используют при 32 битном режиме адресации, cx – при 16 битном. После каждого повторения делают декремент **ecx/cx**; значения флагов при этом не изменяются.

Префикс **repe/repz** (repeat while equal/zero – повторять, пока =) обеспечивает повторение выполнения строковой команды, пока содержимое ecx/cx не = 0 и флаг ZF установлен в 1; этот флаг модифицируется некоторыми строковыми командами (в частности, командами сравнения и сканирования). После каждого повторения идет уменьшение ecx/cx на 1.

Префикс **repne/repnz** (repeat while not equal/zero – повторять, пока не =) повторяет строковую команду, пока **ecx/cx** не = 0 и флаг нуля ZF сброшен в 0. Декремент содержимого регистра ecx/cx идет после каждого выполнения команды.

Префиксы повторения аналогичны по своим функциям командам организации циклов. Например, результат выполнения **rep movsb** аналогичен результату такого фрагмента:

**jcxz l2**

**l1:**

**movsb**

**loop l1**

**l2:**

Но скорость выполнения строковой команды с префиксом намного больше, чем команды вну3 цикла loop: МКпроцессор, обрабатывая команду с префиксом, выполняет только саму команду, уменьшает счетчик и анализирует условие; он не изменяет регистр eip/ip, не производит никаких переходов. Таким образом на аппаратном уровне – максимальное быстродействие.

При повторении команд **movs, stos, lods** МКпроцессор не анализирует состояния флагов, поэтому ZF в этих командах не учитывают. Для этих команд схема работы всех префиксов такая: 1) анализ ecx/cx на равенство 0, и если да – выход из цикла; 2) выполнение строковой команды; 3) уменьшение ecx/cx на 1.

Для команд cmps и scas с префиксами схема работы другая: 1) проверка ecx/cx на равенство 0, если да – выход из цикла; 2) выполнение строковой команды; 3) декремент ecx/cx на 1; 4) проверка флага нуля – выход из цикла будет, если ZF = 0 (для префикса repe) либо ZF = 1 (для префикса repne).

**11.1.3. Строковые команды.**

Для пересылки элемента строки используют команду movs (move string – переслать строку):

**movs приемник, источник**

**movsb**

**movsw**

**movsd**

Эта команда копирует данные, адресуемые парой регистров **ds:esi/si** в область памяти, на младший байт которой указывает пара **es:edi/di**. После пересылки идет автоматический инкремент (DF = 0) либо декремент (DF = 1) содержимого регистров **esi/si и edi/di**. Операнды **movs** используют не для адресации, а для определения типа передаваемой информации (1 байт, 2 байта / 4 байта). Транслятор всегда преобразует команду **movs приемник, источник** в эквивалентную без-операндную команду. Например, если приемник и источник определены как байты (db), то транслятор породит команду **movsb**. Она используют вместе с префиксом rep. После выполнения movs регистр флагов не модифицируют. Пример:

; копирование одной строки в другую str1 db ‘abcd$’

str2 db 5 dup (?)

…

mov si, offset str1 ; смещение строки-источника

mov di, offset str2 ; смещение строки-приемника

mov cx, 5 ; пересылается 5 элементов

cld ; сброс в 0 флага направления – значения индексных

; регистров будут увеличиваться

rep movsb ; копирование строки str1 в строку str2.

Для сравнения элементов строк – команда **cmps** (compare string – сравнить строку):

**cmps источник, приемник**

**cmpsb**

**cmpsw**

**cmpsd**

Эта команда сравнивает данные, адресуемые парой регистров **ds:esi/si** с данными, на которые указывает **es:edi/di**. То есть, из элемента-источника, адресуемого **ds:esi/si**, вычитается элемент-приемник, адресуемый **es:edi/di**; но результат вычитания никуда не помещается – в соответствии с этим результатом устанавливают флаги в регистре **flags**. После сравнения идет авто-инкремент (DF = 0) либо авто-декремент (DF = 1) содержимого esi/si и edi/di. Операнды **cmps** используют для определения размера сравниваемых данных. С командой cmps используют префиксы repe/repz и repne/repnz, а также команды условных переходов. Пример:

; поэлементное сравнение 2 строк

str1 dw 5 dup (?) str2 dw 5 dup (?)

…

mov si, offset str1 ; смещение строки-источника

mov di, offset str2 ; смещение строки-приемника

|  |  |
| --- | --- |
| mov cx, 5 | ; сравнивается 5 элементов |
| cld | ; сброс флага направления |
| repe cmpsw | ; поэлементное сравнение слов |
| jz equal  …  equal: | ; переход, если все элементы строк совпадают |

Для сравнения значения аккумулятора со значением элемента строки - команда **scas** (scan string – сканировать строку):

**scas приемник**

**scasb**

**scasw**

**scasd**

Команда сравнивает значение в регистре **al, ax / eax** (в зависимости от формата команды) с данными, на которые указывает пара регистров **es:edi/di**. После сравнения идет авто-инкремент (если DF=0) либо авто-декремент (если DF=1) значения в edi/di. Операнд команды scas используют для определения размера сканируемых данных. После сравнения устанавливают все флаги в соответствии с вычитанием из регистра-аккумулятора данных по адресу es:edi/di, но сами аккумулятор и строка не изменяются. Команда scas используют вместе с префиксами **repe/repz и repne/repnz** и с командами условного перехода. Пример:

; определение позиции первого элемента, совпадающего с символом в al

str1 db ‘abc1de2$’

…

mov di, offset str1 ; смещение строки-приемника mov cx, 8 ; количество элементов строки

cld ; сброс в 0 флага направления

repne scasb ; сканирование

jcxz nofound ;если в cx – 0, то не найдено ни 1 совпадающего элемента

sub di, offset str1 ; в di – позиция совпавшего элемента

…

nofound:

Для загрузки элемента строки в аккумулятор используют команду **lods** (load string – загрузить строку):

**lods источник lodsb**

**lodsw**

**lodsd**

Эта команда загружает байт, на который указывает пара регистров ds:esi/si, в регистр al (lodsb), слово – в ax (lodsw) / двойное слово – в eax (lodsd). После этого идет авто-инкремент (DF=0) либо авто-декремент (DF=1) содержимого esi/si. Операнд lods – для определения типа загружаемых данных, но не для адресации. Выполнение команды lods не влияет на регистр флагов. Эту команду используют без префиксов. Пример:

; фрагмент, после выполнения которого в регистр al поместят первый

; несовпадающий символ строки str1, сравниваемой со строкой str2

str1 db 5 dup (?)

str2 db 5 dup (?)

…

mov si, offset str1 ; смещение строки-источника

mov di, offset str2 ; смещение строки-приемника

|  |  |
| --- | --- |
| mov cx,5 | ; сравнивается 5 элементов |
| cld | ; сброс флага направления |
| repe cmpsb | ; сравнение строк, идет, пока элементы равны и в cx –  ; значение, отличное от 0 |
| jcxz equal | ; если нет несовпадающих элементов – переход на  ; метку equal |
| dec si | ; уменьшение значения в si – теперь ds:si указывает на  ; первый не совпавший элемент |
| lodsb  …  equal: | ; загрузка этого элемента в al |

Команда lods эквивалентна команде mov аккумулятор, byte ptr [si]. Но строковая команда занимает всего 1 байт, а mov аккумулятор, память – 3 байта в 16 битном режиме адресации и 5 байт – в 32 битном. Также время выполнения МКпроцессором команды **lodsb** меньше, чем время на пересылку mov.

Для записи содержимого аккумулятора в элемент строки – команда stos (store string – сохранить строку):

**stos приемник**

**stosb**

**stosw**

**stosd**

Эта команда записывает содержимое регистра al (ax, eax) в байт (слово, двойное слово), который адресуется парой es:edi/di. После этого идет авто-инкремент (DF=0) либо авто-декремент (DF=1) значения индексного регистра. Как и во всех строковых командах, операнд **stos** используют для определения размера записываемых данных. Выполнение команды не изменяет регистр флагов. Команду **stos** используют с префиксом **rep** для заполнения строки каким-то определенным значением. Пример:

; инициализация элементов массива значением 1

.386

…

mas dd 10 dup (?)

…

mov di, offset mas ; смещение первого элемента массива

|  |  |
| --- | --- |
| mov cx, 10 | ; количество элементов = 10 |
| mov eax, 1 | ; значения, которыми инициализируется массив |
| cld | ; сброс флага направления |
| rep stosd | ; сохранение элементов строки. |

# 11.1.4. Работа с командами обработки строк.

Для адресации строки-источника используют сегментный регистр ds, строки-приемника – регистр es. Но если обе строки лежат в сегменте данных, то es и ds можно инициализировать 1аковыми значениями:

data segment

…

data ends

code segment

…

mov ax, data mov ds, ax

mov es, ax

…

code ends

Операнды, используемые в строковых командах (movs пр.,ист., stos пр. и т.д.) нужны для определения типа обрабатываемой информации, но не для адресации. Такие команды преобразуют транслятором в эквивалентные без-операндные команды.

Использование префиксов повторения для строковых команд - особый вид организации циклов. Но, в отличие от команд loop, loope, loopne, здесь счетчик числа повторений ecx/cx анализируют до выполнения тела. Таким образом, если ecx/cx = 0 перед циклом, то в строковых командах тело не будет выполнено ни разу; в такой же ситуации с командой loop цикл будет повторен 65536 раз (/ 4294967296 – при использовании ecx). То есть можно не использовать проверку счетчика на равенство 0 перед выполнением **rep строковая\_команда**:

len db 10 ; длина строки

…

mov cx, len

rep movsw ; если ecx/cx=0, то ни одной пересылки не будет

---

Часть 2

**11.2. ТАБЛИЧНОЕ ПРЕОБРАЗОВАНИЕ.**

**11.2.1. Команда табличного преобразования xlat.**

Для выборки элемента из строки по индексу используют команду xlat (translate string – перевод строки):

**xlat источник**

**xlatb**

Эта команда заменяет содержимое регистра al значением, выбираемым по индексу из таблицы, на которую указывает пара es:ebx/bx. Содержимое **al** интерпретируют как индекс выбираемого элемента. Операнд команды xlat не используют, транслятор всегда порождает команду **xlatb**. После выполнения этой команды регистр флагов не изменяют. Пример:

; прога, перекодирующая строку позиций в строку осмысленных

; символов; таблица преобразования – английский алфавит; после выполнения данной проги на экран выходит строка “assembler”

data segment ; сегмент данных

table db 'abcdefghijklmnopqrstuvwxyz' ; таблица преобразования, из которой

; выбирают значения

position\_mas db 0,18,18,4,12,1,11,4,17 ; массив позиций – строка, которая

; будет преобразована

|  |  |  |
| --- | --- | --- |
| str1 db 10 dup (?)  data ends |  | ; результирующая строка, в которую  ; преобразуется position\_mas |
| stack1 segment stack |  | ; сегмент стека |
| db 100h dup(?)  stack1 ends |  | ; под стек отводится 256 байт |
| code segment |  | ; сегмент кода |

assume cs:code,ds:data,ss:stack1

begin:

mov ax, data ; занесение в ax начала сегмента данных

mov ds, ax ; инициализация регистра ds

mov es, ax ; в es заносят адрес начала сегмента данных

mov bx, offset table ; в bx заносят смещение перекодирующей таблицы

mov si, offset position\_mas ; в si заносят смещение перекодируемой строки

|  |  |  |
| --- | --- | --- |
| mov di, offset str1 | ; в di заносят смещение результирующей строки | |
| mov cx, 9 | ; длина массива | |
| cld l1: | ; сброс в 0 флага DF, индексные регистры будут увеличены | |
| lodsb  position\_mas | ; в al заносят значение из строки | |
| xlatb | ; значение в al воспринимается как индекс в таблице table по этому индексу выбирают значение, которое загружают в al | |
|
| stosb |  | ; занесение значения al в строку str1 |
| loop l1 |  | ; цикл повторяется 9 раз – длина перекодируемой строки |
| mov byte ptr es:[di],'$' |  | ; символ конца строки - для; корректного вывода на  ; экран функцией9 прерывания 21h (прерывание DOS) |
| mov ah,09h |  | ; для прерывания 21h номер функции заносят в регистр ah |
| sub di,9 |  | ; значение di уменьшают на величину размера массива |
| mov dx, di |  | ; в dx заносят смещение строки str1 – для девятой  ; функции прерывания 21h |
| int 21h |  | ; вызов прерывания |
| mov ah, 4ch |  | ; функция 4ch прерывания 21h – завершение работы проги |
| int 21h  code ends end begin |  | ; вызов прерывания |

---

Часть 3

**12.1. ОСНОВНЫЕ СВЕДЕНИЯ О ПРЕРЫВАНИЯХ.**

**12.1.1. Понятие прерывания. Аппаратные и программные прерывания.**

Прерывание – инициируемый определенным образом процесс, временно переключающий МКпроцессор на выполнение другой проги с последующим возобновлением выполнения прерванной проги.

Обработчик прерывания – это прога, которая прерывает выполнение текущей проги и принимает управление.

К генерации (/ вызову) прерывания может привести две ситуации:

1) внешнее по отношению к выполняемой проге событие (нажатие клавиши на клаве, переход принтера в состояние готовности, деление на нуль в МКпроцессоре);

2) внутреннее событие выполняемой проги – то есть программный (с помощью специальных команд) вызов прерывания.

Аппаратные и программные прерывания.

Аппаратные прерывания инициируются внешними устройствами – внешнее устройство вырабатывает определенный сигнал, по которому МКпроцессор должен обслужить данное устройство. Программные прерывания инициируются в выполняющейся проге специальными командами вызова прерываний.

Всего - до 256 прерываний. Первые 32 из них (прерывания 0 – 1fh) зарезервированы МКпроцессором, остальные 224 - пользовательские и используются прогером.

Каждому прерыванию ставят в соответствие свой номер. Например, прерывание 8 – прерывание от системного таймера, 16 (прерывание 10h) – прерывание обслуживания видеосистемы. Номер прерывания соответствует индексу в таблице, содержащей адреса обработчиков прерываний. Эта таблица лежит в оперативной памяти, и МКпроцессор знает адрес ее начала. Вектора прерываний – это адреса обработчиков, а таблица с этими адресами, – таблица векторов прерываний. При вызове прерывания из этой таблицы по заданному индексу выбирают адрес обработчика, которому передают управление. Сами же обработчики могут быть в любом месте оперативной памяти.

В реальном режиме работы МКпроцессора эта таблица начинается с адреса **0000h:0000h**. Каждый её элемент – 4хбайтный (то есть под вектор прерывания отводят 4 байта). Старшее слово содержит адрес начала сегмента, младшее слово – смещение относительно этого сегмента. Таким образом, например, адрес обработчика восьмого прерывания находится в ячейке с адресом 0000:08h×4. В защищенном режиме работы МКпроцессора адреса обработчиков прерываний находятся в таблице **IDT** (interrupt descriptor table – таблица дескрипторов прерываний). Адрес начала этой таблицы лежит в специальном регистре МКпроцессора.

---

Часть 4

**12.2. КОМАНДЫ РАБОТЫ С ПРЕРЫВАНИЯМИ.**

**12.2.1. Вызов прерывания.**

Для вызова прерывания с заданным номером используют команду int (interrupt – прервать):

**int число**

Команда int сначала заносит в стек значения таких регистров: eflags/flags, cs, eip/ip (регистр флагов помещают в стек первым, регистр указателя инструкций – последним). Затем управление передают обработчику прерывания, номер которого равен числу, указанному в данной команде.

Значение операнда(числа) команды **int** должно быть от 0 до 255. Занесение в стек регистра флагов делают, чтобы при возврате из обработчика этот регистр был восстановлен, и вычислительный процесс не был нарушен. Значения cs:eip/ip - адреса возврата, они нужны для правильного возврата из проги обработчика. при занесении в стек значения eip/ip этот регистр указывает на команду, следующую за командой int.

Обработчики прерываний многофункциональны и выполняют разные действия в зависимости от передаваемых им параметров. Например, для установки режима дисплея используют прерывание BIOS 10h (обслуживание видеосистемы), причем в регистре ah указывают номер функции 0 (установка режима дисплея), а в al задают сам режим:

xor ah, ah ; функция 0 прерывания 10h – установка режима дисплея

mov al, 3 ; в al задают режима дисплея, третий режим – текстовый

; режим, 80 столбцов 25 строк

int 10h ; вызов прерывания, после выполнения команды

; устанавливают требуемый режим дисплея.

Для вызова третьего прерывания в МКпроцессоре есть специальная 1 байтная команда (любая другая команда int – 2 байтная):

**int 3**

Для прерывания **int 3** по умолчанию устанавливают пустой обработчик, то есть вызов int 3 не приводит ни к каким последствиям. Прогер может написать свой обработчик третьего прерывания. Многие проги отладчики используют прерывание 3 для установки точки прекращения проги. Отладчик заменяет 1 байт кода команд в точке прекращения командой **int 3**, благодаря чему после достижения точки прекращения управление передают обработчику, в котором можно выполнить какие-либо проверки / установить режим пошагового выполнения проги.

Для вызова четвертого прерывания при наличии переполнения (флаг OF=1) используют команду **into** (interrupt on overflow – прерывание по переполнению):

**into**

Длина into – 1 байт. Команда into вызывает прерывание 4, если флаг переполнения OF установлен в 1. Для прерывания 4 по умолчанию устанавливают пустой обработчик прерывания, но прогер может создать свой, сигнализирующий об ошибках переполнения (флаг OF модифицируется некоторыми арифметическими, логическими и строковыми командами).

Команда вызова прерывания 5 при условии выхода значения за рамки диапазона bound (bound – предел):

**bound индекс, источник**

Команда bound проверяет, находится ли значение индекса в заданном диапазоне. Индекс – только регистр. Источник задает нижнюю и верхнюю границы диапазона, и если значение индекса вне границ диапазона, то генерируют прерывание с номером 5. В стек заносят пару cs:eip/ip, указывающую на саму команду **bound**, а не на следующую за ней команду, как при обычном выполнении **int**. Источник – только ячейка памяти, младший байт которой содержит нижнее значение диапазона, (слово, двойное слово – в зависимости от размера тестируемого регистра) а старший байт (слово, двойное слово) – верхнюю границу диапазона. Значения индекса и источника – числа со знаком. Команду **bound** применяют при работе с массивами. Пример:

|  |  |
| --- | --- |
| **min\_max dw -6, 9**  **…** | ; определение значения источника, младшее слово –  ; нижняя граница диапазона (число -6), старшее слово –  ; верхняя граница диапазона (число 9) |
| **mov bx, 10** | ; занесение в регистр bx числа 10 – проверяемый индекс |
| **bound bx, min\_max** | ; так как значение в bx лежит вне диапазона, то идет  ; вызов прерывания 5 |

**12.2.2. Выход из прерывания.**

Для корректного выхода из обработчика прерывания - команда iret (interrupt return):

**iret iretd**

Команда iret извлекает из стека значение регистра ip, значение регистра cs и значение регистра **flags** (первым извлекают регистр указателя инструкций, последним – регистр флагов). Команда **iretd** извлекает из стека значения **eip**, cs и **eflags**. Занесения в **cs:eip/ip** данных из стека реализует дальний (far) переход. Таким образом, после выполнения **iret/iretd** управление передают команде, на которую указывает пара **cs:eip/ip.**

Команды iret и iretd имеют 1 и тот же машинный код. Например, если при 32 битном режиме адресации надо восстановить 2байтные ip и flags, то транслятор поместит перед кодом операции iret/iretd префикс размера операнда.

---

Часть 6

**13.1. ОПИСАНИЕ ПРОЦЕДУР. КОМАНДЫ РАБОТЫ С ПРОЦЕДУРАМИ.**

**13.1.1. Описание процедур в Ассемблере.**

Процедура (/ подпрога) – это группа логически выделенных команд, которые могут быть многократно вызваны прогой. Вызов процедуры – это передача ей управления. Вызов процедуры требует дополнительных затрат времени на выполнение перехода и возврата из процедуры. Для описания процедур – две директивы – proc и endp:

имя\_процедуры **proc** [тип] [язык] [USES регистры] ; заголовок процедуры

… ; команды, составляющие тело процедуры

имя\_процедуры **endp**

Здесь идентификатор **имя\_процедуры** – это метка и указывает на первую команду процедуры. Директива **endp** определяет место завершения процедуры. Здесь модификаторы в квадратных скобках – необязательные. Модификатор тип может принимать значения **near** и **far**, соответственно создаваемые процедуры будут определяться как ближние и дальние. При наличии модификатора **uses** транслятор создаст последовательность команд **push** в самом начале тела процедуры, таким образом, в стек будут заноситься значения указанных в **uses** регистров. Перед командой ret будет создана последовательность команд **pop**. Если тип процедуры не задан, то неявно он выбирается исходя из используемой модели памяти (для моделей tiny, small и compact – тип near).

Процедура может лежать в любом месте проги, но надо учитывать, что после компиляции ее код не отделим от остального кода проги (то есть в запускаемой на выполнение проге все команды расположены линейно друг за другом, как будто в исходном тексте не было директив proc и endp). Если в своем выполнении прога дойдет до первой команды процедуры без ее явного вызова, то это приведет к неожиданным последствиям. Например:

|  |  |
| --- | --- |
| mov ax, bx | ; некоторый код |
| mov bx, cx | ; после этой команды будет выполнено xor si, si, так, будто |
|  | ; процедура myproc на самом деле была вызвана |
| myproc proc |  |

xor si, si

…

myproc endp

Чтобы избежать такой ошибки надо либо размещать процедуры в начале / конце проги (до первой исполняемой команды / после команды, возвращающей управление операционной системе), либо в теле проги предусма3вать обход процедуры с помощью команд перехода. Пример корректного описания процедуры:

.model small .data

message db "Hello!!!$"

.code

show\_message proc mov dx, offset message

mov ah, 9

int 21h ; вывод строки

ret ; возврат управления вызвавшей проге

show\_message endp

start:

mov ax, @data

mov ds, ax

call message\_proc ; вызов процедуры

mov ax,4C00h

int 21h ; выход из проги

end start

**13.1.2. Команды работы с процедурами.**

Для передачи управления процедуре используют любую команду перехода, в том числе и условного. Лучше команду вызова call (call – звонить):

**call [модификатор] операнд**.

Команда call сохраняет в стеке адрес возврата и передает управление по адресу, который определяется операндом этой команды. Операндом может быть:

1) имя процедуры / имя метки: для ближнего (вну3сегментного) перехода в формат машинной команды записывают относительное смещение; для дальнего (межсегментного) перехода машинная команда содержит полный адрес ячейки, на которую передают управление (то есть сегмент и смещение);

2) ячейка памяти / регистр: извлекаемые из памяти / регистра значения используют как адрес перехода (для ближнего перехода извлеченное значение модифицирует только регистр eip/ip, для дальнего – регистры cs и eip/ip).

Адрес возврата, заносимый в стек командой call, – это адрес младшего байта команды, следующей за call. Для ближнего перехода в стек заносят только значение регистра eip/ip, для дальнего – сначала cs, затем eip/ip. Модификатор – near ptr и far ptr, соответствующие ближнему и дальнему вызовам. Если тип процедуры указан явно в директиве proc, то использование модификаторов необязательно. Пример:

; ближние переходы call Proc1

call near ptr Proc1 mov dx, offset Proc1

call dx

…

Proc1 proc near

…

Proc1 endp

…

; дальние переходы

addr16 dw ? ; смещение addr16 dw ? ; сегмент

…

call Proc2

call far ptr Proc2

mov addr16, offset Proc2

mov addr16+2, seg Proc2

call dword ptr addr16

…

Proc2 proc far

…

Proc2 endp

Для возврата из процедуры - команда ret (return – возвращаться):

**ret [число]**

**retn [число]**

**retf [число]**

Команда **ret** передает управление по адресу, который лежит на вершине стека. Команда **retn** - для возврата из ближней процедуры, retf – из дальней. Команда ret всегда преобразуется транслятором либо в **retn,** либо в **retf** в зависимости от типа процедуры. Команда retn для 16 битного режима адресации извлекает из стека 2 байта и загружает их в регистр **ip** (для 32 битного режима адресации из стека извлекают 4 байта, которые загружаются в **eip**). Команда **retf** для 16 битного режима адресации сначала извлекает из стека 2 байта и загружает их в регистр **ip** (для 32 битного режима адресации из стека извлекают 4 байта, которые загружают в eip); затем из стека берут еще 2 байта, которые загружают в cs. Если в команде retn / retf указано число, то после забора из стека адреса возврата, значение указателя вершины стека esp/sp уменьшают – чтобы уничтожить передаваемые процедуре через стек параметры. При 16 битном режиме адресации сегмента число, указанное в команде, определяет байты, которые надо удалить из стека (то есть от esp/sp отнимают это число), при 32 битном режиме адресации указанное в команде число определяет слова (то есть от esp/sp отнимают число×2). Пример:

|  |  |
| --- | --- |
| call Proc1  inc bx  … | ; вызов ближней процедуры Proc1 |
| call far ptr Proc2  dec bx  … | ; дальний вызов процедуры Proc2 |
| push ax | ; передаваемый параметр |
| push cx  call near ptr Proc3  …  Proc1 proc near  … | ; передаваемый параметр |

ret ; ближний возврат (retn) управления команде, следующей за

; call, в данном случае – команде inc bx

Proc1 endp

Proc2 proc

…

retf ; дальний возврат управления команде, следующей за

; call, в данном случае – команде dec bx Proc2 endp

Proc3

…

retn 4 ; удаление из стека переданных процедуре параметров, от

; sp отнимают число 4

Proc3 endp.

Две команды, специально предназначенные для создания и удаления локальных (внутренних) переменных процедур: в области стека выделяют участок памяти заданного размера для хранения каких-то данных. Причем эти данные не доступны из внешнего блока, и живут они столько, сколько живет подпрога.

Команда **enter** - для выделения участка памяти в области стека:

**enter размер, вложенность.**

Эта команда выполняет действия, аналогичные тем, что приведены на рис. 13.1. В качестве операндов размер и вложенность могут использоваться только целые без-знаковые числа. Операнд принимает значение от 0 до 31.

push ebp

/

bp

temp

=

esp

/

sp

вложенность

>

1

ebp

/

bp

=

temp

push ebp

/

bp

esp

=

esp

-

размер

вложенность

=

вложенность

-

1

ebp

/

bp

=

ebp

/

bp

-

n

push

[

ebp

/

bp

]

да

нет

выделение

участка памяти

указанного

размера

n

=

2

для 16 битного

режима адресации

,

n

=

4

для

32 битного режима

адресации

данные

,

на которые указывает

регистр

Рис. 13.1. Схема работы команды enter.

Команда enter сначала кладет в стек текущее значение ebp/bp, чтобы по команде **leave** (освобождение области) регистр ebp/bp содержал значение, которое находилось в нем до вызова **enter**. Затем содержимое ebp/bp уменьшают на 2 / на 4 (определяется режимом адресации) и в стек кладут данные, на которые указывает регистр ebp/bp. Такое уменьшение значения в ebp/bp и занесение в стек косвенно адресуемых через этот регистр данных повторяется вложенность-1 раз.

После этого в регистр ebp/bp заносят значение, которое имел указатель вершины стека esp/sp после самого первого занесения в стек данных – то есть после занесения в стек значения ebp/bp. Это надо для корректного срабатывания команды **leave**. Затем содержимое esp/sp уменьшают на значение, определяемое операндом размер. Таким образом, после выполнения enter регистр esp/sp указывает на младший байт только что выделенного участка памяти. Пример:

.186

; пусть в этой точке содержимое sp = 00fah, значение по адресу

; ss:[1232h] = 0fc76h, значение по адресу ss:[1230h] = 0ff06h – эти

; предположения сделаны для определенности mov bp,1234h enter 10,3

; вид стека после выполнения команды enter - на рис. 13.2, значение в

; регистре bp = 00f8h, в регистре sp – 00e8h

1234

h

fc

76

h

ff

06

h

00

f

8

h

ss

:

00

f

8

h

ss

:

00

f

6

h

ss

:

00

f

4

h

ss

:

00

e

8

h

ss

:

00

f

2

h

выделенные 10 байт

Рис. 13.2. Содержимое стека после выполнения команды enter (пример).

Если значение вложенности = 0, то в стек кладут только текущее значение ebp/bp, и сразу после этого esp/sp уменьшают. Перед выполнением leave регистр ebp/bp должен содержать то же самое значение, что и сразу после выполнения enter. Если размер, указанный в команде enter, не кратен 2 (для 16 битного режима адресации) / 4 (для 32 битного), то выделяемая область выравнивается на границу слова / двойного слова соответственно.

Например, для команды enter 9,0 под локальные переменные при 16 битном режиме адресации стекового сегмента выделяют 10 байт, что соответствует пяти 2 байтным элементам стека; а при 32 битном – 12 байт, что соответствует трем 4 байтным элементам стека.

Для освобождения участка памяти, отведенного командой enter – команда **leave** (leave – покидать):

**Leave**.

Сначала команда leave копирует содержимое ebp/bp в регистр esp/sp. Затем эта команда берет с вершины стека, на которую указывает обновленный регистр esp/sp, значение, загружаемое в ebp/bp. Таким образом, после выполнения leave выделенный участок памяти в стеке свободен, а регистр ebp/bp содержит значение, которое он имел до вызова enter. Пример:

.186

; пусть регистр sp содержит число 00fah – для определенности

mov bp, 1234h

enter 10, 3

… ; перед leave содержимое bp должно соответствовать

; содержимому bp сразу после выполнения enter

leave ; после выполнения leave в sp находится 00fah, в bp – 1234h.

Пример проги как идет доступ к локальным переменным:

; в проге вызывается процедура, создающая 3 локальных1 байтных

; переменных. процедура вычисляет сумму этих элементов и заносит ее в al

.186

stack1 segment stack use16 ; сегмент стека

|  |  |
| --- | --- |
| db 100h dup(?) stack1 ends | ; под стек отводится 256 байт |
| code segment use16 assume cs:code, ss:stack1 begin: | ; сегмент кода |
| call near ptr calc\_sum | ; вызов процедуры |

cmp al, 60 ; если после вызова процедуры значение в регистре al

; = 60, то на экран выводится символ ‘!’

jnz exit

mov ah, 02h

mov dl, '!'

int 21h ; вывод символа ‘!’

exit:

mov ah, 4ch

int 21h ; выход из проги

calc\_sum proc near

enter 3, 0

; инициализация переменных значениями 10, 20 и 30 mov byte ptr [bp-2], 10

mov byte ptr [bp-3], 20

mov byte ptr [bp-4], 30

xor al, al

; суммирование значений локальных переменных add al, [bp-2]

add al, [bp-3]

add al, [bp-4]

leave

retn

calc\_sum endp

code ends

end begin.

Выполнение команд enter и leave не влияет на состояние флагов в регистре eflags/flags.

---

Часть 7

**13.2. ПЕРЕДАЧА ПАРАМЕТРОВ ПРОЦЕДУРАМ.**

# 13.2.1. Общие сведения.

Важный момент организации работы процедур на Ассемблере – передача им параметров двумя способами: 1) через общую область памяти, используя стек / глобальные переменные; 2) через регистры.

# 13.2.2. Передача параметров через регистры.

Использование регистров МКпроцессора – самый быстрый и простой метод передачи параметров процедурам. Суть метода – в сохранении нужных параметров в регистрах, и их последующем использовании в процедуре. Это обеспечивается тем, что команда call не меняет никаких регистров (кроме eip/ip и, возможно, cs). Пример:

.model small

.data

msg1 db "First", 10, 13, '$'

msg2 db "Second", 10, 13, $'

msg3 db "Third", 10, 13, '$'

.stack 100h

.code

begin:

mov ax, @data

mov ds, ax

|  |  |
| --- | --- |
| mov dx, offset msg1 | ; передаваемый процедуре параметр |
| call show | ; вызов процедуры |
| mov dx, offset msg2 | ; передаваемый процедуре параметр |
| call show | ; вызов процедуры |
| mov dx, offset msg3 | ; передаваемый процедуре параметр |
| call show  mov ah,4ch | ; вызов процедуры |
| int 21h | ; выход из проги |

show proc near

; в регистре dx процедуре передают эффективный адрес строки

; для вывода mov ah, 9

int 21h ; вывод строки

ret

show endp

end begin

«+» метода передачи параметров через регистры: использование регистров наиболее быстрый и простой метод, не требует лишней памяти.

3 недостатка: 1) ограниченное количество регистров МКпроцессора и 2) ограничение размера передаваемых параметров. 3) регистр надо сохранять так как значение регистра не должно меняться до того места в процедуре, где оно должно быть использовано по назначению.

# 13.2.3. Передача параметров через глобальные переменные.

Использование глобальных переменных для передачи параметров предполагает, что вызывающая и вызываемая проги используют некоторую область памяти как общую. Пример:

.model small

.data

switcher dw ? ; общая переменная, будет иметь эффективный адрес строки

msg1 db "First", 10, 13, '$'

msg2 db "Second", 10, 13, '$'

msg3 db "Third", 10, 13, '$'

.stack 100h

.code begin:

mov ax, @data

mov ds, ax

|  |  |
| --- | --- |
| mov switcher, offset msg1 | ; глобальная переменная |
| call show | ; вызов процедуры |
| mov switcher, offset msg2 | ; глобальная переменная |
| call show | ; вызов процедуры |
| mov switcher, offset msg3 | ; глобальная переменная |
| call show | ; вызов процедуры |

mov ah, 4ch

int 21h

show proc near mov ah, 9

mov dx, switcher ; смещение строки

int 21h ; вывод строки ret

show endp

end begin

«+» передачи параметров через общую область памяти: имеет большую гибкость и избавлен от недостатков, характерных для способа передачи параметров через регистры.

«-»: процедуру становится невозможно использовать вне текущего контекста проги. То есть, процедура не может быть использована в проге, в которой не описана нужная ей глобальная переменная. Этот менее быстрый метод по сравнению с передачей параметров через регистры.

# 13.2.4. Передача параметров через стек.

Суть метода передачи параметров через стек: перед вызовом процедуры в стек заносят параметры, которые надо ей передать. Затем в вызванной процедуре идет обращение к этим параметрам, но после выполнения call на вершине стека всегда лежит адрес возврата. Пример:

|  |  |  |
| --- | --- | --- |
| op1 dw ?  op2 dw ?  … |  | |
| mov op1, 300 | ; инициализация переменной op1 | |
| mov op2, 200 | ; инициализация переменной op2 | |
| push op1 | ; занесение в стек значения op1 | |
| push op2 | ; занесение в стек значения op2 | |
| call near ptr calc\_sum  … | ; вызов ближней процедуры | |
| calc\_sum proc near | ; процедура, вычисляющая сумму переданных ей | |
| ; параметров, эта возвращается в регистре ax | |
|  | |
| mov bp, sp | ; так как регистр sp нельзя использовать для косвенной | |
|  | ; адресации, то вместо sp используют bp | |
|  |  | |
| mov ax, [bp+2] | ; в ax будет число 200 | |

add ax, [bp+4] ; в ax будет находится сумма 200+300=500

retn 4 ; возврат из процедуры, переданные параметры удаляют

calc\_sum endp.

Для дальнего вызова процедур на вершине стека лежит пара cs:eip/ip, поэтому на последний переданный через стек параметр указывает значение sp+2 (16 битный режим адресации) / sp+6 (32 битный).

При завершении процедуры надо выровнять стек, скомпенсировав выполненные перед вызовом команды push, используя команду ret с операндом (ret число) / прибавив к esp/sp нужное значение. Для выравнивания стека можно также выполнять команды pop после возврата из процедуры.

Используя стек, можно реализовать 2 вида передачи параметров:

1) передача по значению (в стек заносят непосредственно значения переменных, и процедура работает с этими значениями; таким образом, в вызывающей проге сами переменные не изменяются);

2) передача по адресу (в стек заносят адрес переменной, и процедура может изменять значение данной переменной). Пример:

; прога, передающая через стек адрес переменной; процедура exec\_proc

; извлекает из стека переданный ей параметр; этот параметр – указатель на

; переменную op1; в переменной op1 лежит адрес процедуры, вызов которой

; идет из exec\_proc; после вызова процедуры из exec\_proc значение op1

; изменяют (везде в exec\_proc используют переданный через стек адрес);

в результате выполнения данной проги на экран выводят две строки – ‘abc’

; и ‘xyz’

data segment

op1 dw ?

msg1 db 'abc$'

msg2 db 'xyz$'

data ends

stack1 segment stack

db 100h dup(?)

stack1 ends

code segment

assume cs:code,ds:data,ss:stack1

begin:

mov ax, data

mov ds, ax

mov op1, offset Proc1 ; в op1 – адрес процедуры Proc1

mov ax, offset op1 ; в ax – эффективный переменной op1

push ax ; занесение в стек адреса переменной op1

call near ptr exec\_proc ; вызов exec\_proc, на экран выводится строка ‘abc’ –

; выполняется Proc1, также значение op1 изменяют

push ax ; занесение в стек адреса переменной op1 (ведь изменение

; значения не модифицирует адрес)

call near ptr exec\_proc ; вызов exec\_proc, на экран выводится строка ‘xyz’ –

; выполняется Proc2

mov ah, 4ch

int 21h

exec\_proc proc near

push bp ; все регистры, значения которых изменяются в процедуре, ; лучше всегда сохранять – так процедура не изменит

; нормальный ход проги, из которой она вызывается push bx mov bp, sp

mov bx, [bp+6] ; переданный параметр лежит по адресу bp+6, так как 2 байта

; занимает адрес возврата из процедуры, и 4 байта занимают

; сохраненные в стеке значения bx и bp

call word ptr [bx] ; вызов процедуры, адрес которой лежит в ячейке,

; адресуемой bx

mov word ptr [bx], offset Proc2 ; изменение значения по адресу, который

; был передан процедуре exec\_proc

pop bx ; извлечение регистров, значения которых

; модифицировались в процедуре

pop bp retn 2 ; выход из процедуры и удаление одного элемента стека –

; переданного параметра

exec\_proc endp

Proc1 proc near push dx

push ax

mov dx, offset msg1

mov ah, 09h

int 21h ; вывод строки ‘abc’

pop ax

pop dx

retn

Proc1 endp

Proc2 proc near push dx

push ax

mov dx, offset msg2 mov ah, 09h

int 21h ; вывод строки ‘xyz’

pop ax

pop dx retn

Proc2 endp

code ends

end begin

«+» способа передачи параметров через стек: 1) самый гибкий способ передачи параметров процедурам, 2) количество передаваемых параметров практически не ограничено, 3) процедура независима от внешней проги, и, следовательно, может быть использована в других прогах без модификаций. «-»: 1) увеличивается размер кода проги, 2) уменьшается быстродействие.

---

Часть 8

**14.1. ВЫРАЖЕНИЯ И ОПЕРАТОРЫ.**

**14.1.1. Общие сведения.**

Для облегчения процесса написания программ используют выражения. **Выражение** – это совокупность переменных и констант, связанных друг с другом операторами. Операторы реализуют операции арифметические и логические, и другие. Значения всех выражений вычисляют транслятором в ходе транслирования проги, но не на этапе ее выполнения.

Арифметические операторы: сложения (+), вычитания (-), умножения (\*), целочисленного деления (/), остатка от деления (mod), унарные + и -.

Логические операторы: логического умножения (and), логического сложения (or), логического исключающего / (xor), сдвигов влево (shl) и вправо (shr).

Операторы сравнения дают 1, если проверяемое условие истинно, если ложно – 0. Операторы сравнения: равно (eq), больше / равно (ge), больше (gt), меньше / равно (le), меньше (lt) и не равно (ne).

Операторы Ассемблера не имеют ничего общего с командами. Значения выражений вычисляют только на этапе транслирования проги. То есть, например, будет неверным выражение al\*7, так как транслятору не известно содержимое регистра al ни в момент транслирования, ни в момент выполнения проги. Пример:

|  |  |
| --- | --- |
| add al, 5\*6 | ; к значению в регистре al прибавить число 30, число  ; 5 было умножено на 6 во время транслирования |
| mov bx, 1 shl 3 | ; занесение в регистр bx числа 8, здесь shl – оператор  ; число 1 сдвигают влево на этапе трансляции, а в  ; машинную команду записывают результат вычисления  ; выражения |

**14.1.2. Операторы, присваивающие а3буты.**

Оператор ptr (pointer – указатель) - для явного указания типа (/ размера) выражения:

**тип ptr выражение.**

Параметр тип может иметь одно из следующих значений / имен:

Имя Значение

byte 1

word 2

dword 4

qword 8

tbyte 10

near 0FFFFh

far 0FFFEh

Типы byte, word, dword, qword, tbyte используют с операндами из памяти. Типы near и far – с метками. Пример:

op1 dw ?

…

mov al, byte ptr op1 ; в al загружается младший байт переменной op1

call far ptr Proc1 ; вызов дальней процедуры

Proc1 proc

…

Proc1 endp

Оператор short (short – короткий) устанавливает тип метки short:

**short метка.**

Если расстояние между меткой и командой перехода – в диапазоне -128..127 байт, то метка можно объявить как метку типа short. Длина машинной команды, использующей метку short, короче любой команды перехода.

Оператор this (this – этот) создает операнд, адрес которого – текущее значение смещения относительно начала сегмента:

**this тип**

Тип, указанный в операторе this, определяет то, как трактовать извлекаемый из памяти операнд. Параметр тип может быть: byte, word, dword, qword, tbyte, near, far. Пример:

mov bx, this word-2 ; занесение в регистр bx последнего слова

; предыдущей команды.

**14.1.3. Операторы, возвращающие значение.**

Оператор high (high – высокий) возвращает старшие 8 битов выражения:

**high выражение.**

Пример:

mov bl, high 1234h ; занесение в bl числа 12h

Оператор **low** (low – низкий) возвращает младший байт выражения:

**low выражение**.

Пример:

mov bl, low 1234h ; занесение числа 34h в bl.

Оператор **seg** (segment – сегмент) возвращает значение сегмента, в котором расположено выражение:

**seg выражение.**

Выражение может быть переменной, меткой / именем сегмента. Пример:

op1 dw ?

…

mov ax, seg op1 ; занесение в регистр ax сегментной

; компоненты адреса переменной op1

mov es, ax ; инициализация регистра es

Оператор **offset** (offset – смещение) возвращает число байт между выражением и началом сегмента, в котором оно определено:

**offset выражение.**

Выражение может быть меткой, именем сегмента / переменной. Если указано имя сегмента, то возвращается смещение от начала сегмента до последнего сгенерированного байта в этом сегменте. Пример:

data segment

…

str1 db 5 dup(?) data ends

…

mov dx, offset str1 ; занесение в регистр dx эффективного адреса переменной

; str1 (то есть смещения относительно сегмента data)

Оператор **type** (type – тип) возвращает число байт, нужных для хранения переменной того типа, каким является выражение; для метки near возвращает 0FFFFh, а для метки far - 0FFFEh:

**type выражение**.

Пример:

|  |  |  |
| --- | --- | --- |
| op1 dd ? op2 dw 5 dup(?)  … |  |  |
| mov al, type op1 |  | ; занесение в al числа 4 |
| mov bl, type op2 |  | ; занесение в bl числа 2. |

Оператор **length** (length – длина) возвращает число единиц типа byte, word, dword, qword / tbyte, занимаемых переменной:

**length переменная**

Тип переменной определяет, в каких единицах измерения возвращается ее длина. По сути, значение, возвращаемое этим оператором, есть число, предшествующее оператору dup в описании переменной. Поэтому только для переменных, описанных с помощью оператора dup, будет возвращено значение, отличное от 1. Строковые константы также возвратят 1. Пример:

op1 db 5 dup(?) str1 db 'abc'

…

mov al, length op2 ; в регистр al заносят число 5

mov cl, length str1 ; в регистр cl заносят число 1

Оператор **size** (size – размер) возвращает число байт, занимаемых переменной:

**size переменная**.

Значение, возвращаемое этим оператором, равно длине **length** переменной, повторенной type раз, то есть

**size переменная = (length переменная) × (type переменная)**

Пример:

op1 dw 5 dup (?)

…

mov si, size op1 ; занесение в регистр si числа 10

Оператор $ возвращает текущее значение счетчика адреса, то есть смещение адреса текущего оператора:

**$ выражение**.

Пример:

jmp $+3 ; переход на метку L1

db ? ; резервирование байта памяти

L1:

---

Часть 9

**14.2. МАССИВЫ.**

**14.2.1. Объявление массива. Работа с массивами.**

Массив – это упорядоченный набор однотипных элементов.

Для задания массивов целых чисел использовать оператор дублирования dup.

Например, для задания массива из 10 2байтных элементов:



В Ассемблере нельзя объявлять 2- и более мерных массивов. Для их организации прогеру надо самому их обрабатывать - как хранить элементы массива. Многомерные массивы - хранить по строкам (рис. 14.1).

Массив A m×n:

a[1,1] a[1,2] … a[1,n] a[2,1] a[2,2] … a[2,n]

…

a[m,1] a[m,2] … a[m,n]

Представление в памяти:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| a[1,1] | a[1,2] | ... | a[1, n] | a[2,1] | a[2,2] | ... | a[m, n] |

строка 1 строка 2 строка m

Пример поиска максимального элемента в третей строке массива 2х10 2байтных элементов.

data segment

; задание и инициализация массива arr

arr dw 438,4, 76,87, 34,67, 234, 90, 48, 0

dw 34, -15, 2451, 6125, 864,92, 238, 4769, 1237, 6281

dw 10, 59, 251, 125, 64, 2, 8, 49, 37, 81

data ends

stack1 segment stack db 100h dup(?)

stack1 ends

code segment

assume cs:code, ds:data, ss:stack1

begin:

mov ax, data

mov ds, ax ; занесение в ds адреса начала сегмента данных data

; позиционирование в массиве на первый элемент третей строки

mov ax, 2 ; нумерация начинается с 0

**mov cl, 10\*type arr** ; 10 2байтных элементов

mul cl ; в ax – позиция первого элемента третей строки mov bx, ax ; занесение в bx значения из ax, так как регистр ax не

; может быть использован для косвенной адресации

mov ax, arr[bx] ; занесение в ax первого элемента 3й строки

; поиск максимума

|  |  |
| --- | --- |
| mov cx, 10-1  L1: | ; будут сравниваться 9 элементов |
| add bx, type arr | ; увеличение на 2 индекса массива |
| cmp ax, arr[bx] | ; в ax – текущий максимум |
| jge no\_new\_max | ; если в ax число, меньшее элемента массива, то |
| mov ax, arr[bx] no\_new\_max: | ; занесение нового максимума в ax |
| loop L1  mov ah,4ch | ; цикл, после его выполнения в ax будет число 251 |
| **int 21h**  **code ends end begin** | ; выход из проги |

---

Часть 10

**14.3. СТРУКТУРЫ.**

# 14.3.1. Объявление структур. Работа со структурами.

**Структура** – это тип данных, состоящий из фиксированного числа элементов различного типа. Элементы, входящие в состав структуры, называют **полями**. Другими словами, **структура** – это объединение различных полей под одним именем.

Для задания структур - директива struc (structure – структура):

**имя\_структуры struc**

**… ; определение полей структуры имя\_структуры ends**

Поля структуры определяются с помощью директив определения данных (например, db / dw). Пример:

**date struc**

**year dw ?**

**month db 10**

**day db 28**

**other db 10 dup (?) date ends**

Директива **struc** не выделяет памяти под структуру, она служит лишь для описания шаблона. Для создания физической переменной (для выделения памяти под переменную типа структуры) - описание (в квадратных скобках - необязательные элементы):

**[имя\_переменной] имя\_структуры <[значения]>**

В угловых скобках <> - начальные значения полей по порядку. Если значения параметров не указывают (пустой список параметров, <>), то все поля инициализируются значениями, которые они имели в шаблоне структуры. Если надо инициализировать какое-то определенное поле значением, которое оно имело в структуре, то его надо просто отделить запятой, не указав значения. В памяти поля переменной (типа структуры) лежат линейно друг за другом, как и в шаблоне. Пример:

event1 date <2003, , ,> ; поле year становится = 2003, month – 10, day – 28

event2 date <> ; инициализируют только поля month и day

Для доступа к полям структур - оператор . (точка):

имя\_переменной.имя\_поля

Пример:

mov event1.month, 12 ; занесение в поле month переменной event1 значения 12

Определение массивов структур. Пример:

events date 10 dup(<>)

…

mov bx, 4

mov events[bx].day, 5 ; занесение числа 5 в поле day 4го элемента массива

; events

---

Часть 11

**14.4. ЗАПИСИ.**

**14.4.1. Объявление записей. Работа с записями.**

**Запись** – это тип данных, состоящий из фиксированного числа битовых полей. Для задания записей - директива record (record – запись):

**имя\_записи record имя\_поля : длина\_поля [= значение] [, имя\_поля :**

**длина\_поля [= значение] ] [, …].**

Директива record - для определения 2ичного (битового) набора в байте, слове / двойном слове. Общий размер записи определяется суммой длин ее полей и не может быть больше 8, 16 / 32 битов. Если суммарный размер всех полей меньше 8, 16 / 32 битов, то элемент записи выравнивается на границу байта, слова / двойного слова (например, элемент длиной 13 битов выравнивается до 2байтного значения). В директиве **record** длины полей задают в битах. Необязательное указание значения определяет значение, которым инициализируется данное поле. Пример:

bit\_set record b1 : 1, b2 : 4, b3 : 3 = 101b.

Директива **record** описывает шаблон записи. Поэтому для использования записей надо объявлять переменные:

**[имя\_переменной] имя\_записи <[значения]>**

Объявленные переменные имеют в проге тот же смысл, что и числовые константы. В угловых скобках - значения, которыми инициализируют поля данной переменной. Если надо проинициализировать все поля значениями, совпадающими тем, что описаны при задании записи в директиве record, то следует просто опустить все значения в угловых скобках (просто <>). Значения полей перечисляют через запятую, при этом если надо задать значение, совпадающее с определенным в описании record, то такое значение просто выделяют запятыми. Пример:

bit\_set record b1:1, b2:4, b3:3 = 101b

…

set1 bit\_set <0, 0000b, > ; поле b1=0, b2=0000b, b3 = 5 = 101b

set2 bit\_set < , 1001b> ; поле b1=0, b2=1001b, b3 = 5 = 101b.

В отличие от структур, к полям записей в иных случаях, кроме инициализации переменной, обращаться нельзя. В тексте проги переменные типа записи используют везде, где возможно указание непосредственного операнда длины байт, слово, двойное слово (в зависимости от размера переменной, то есть от суммы длин всех полей). Пример:

bit\_set record b1:1, b2:4 = 0, b3:3 = 101b set1 bit\_set <1, , 011>

…

mov bl, set1 ; занесение в регистр bl числа 10000011b = 131

Для работы с записями в Ассемблере есть 2 спец оператора. Оператор **width** (width – ширина) возвращает длину в битах поля записи / всей записи:

**width имя\_поля**

**width имя\_записи.**

Пример:

bit\_set record b1:1, b2:4, b3:3 = 101b

…

mov bl, width bit\_set ; в регистр bl заносят значение 8

mov dl, width b2 ; в регистр dl заносят значение 4

Оператор **mask** (mask – маска) возвращает число из установленных в 1 битов в позициях, соответствующих данному полю в записи; все остальные биты сбрасывают в 0:

**mask имя\_поля**.

Пример:

bit\_set record b1:1, b2:4, b3:3 = 101b

…

mov bl, mask b2 ; в регистр bl заносят 01111000b

mov ax, mask b3 ; в регистр ax заносят 0000000000000111b

При использовании имен полей в командах транслятор в формат машинной команды заносит непосредственный операнд, его значение - число, на которое надо сдвинуть данное поле, чтобы выровнять его по правому краю. Пример:

bit\_set record b1:1, b2:4, b3:3 = 101b

…

mov ax, b3 ; в регистр ax заносят 0, так как маска поля b3 = 00000111b

|  |  |  |
| --- | --- | --- |
|  |  |  |
| add dl, b2 |  | ; к содержимому dl прибавляют число 3, так как  ; маска поля b2 = 01111000b |
| mov cl, b1 |  | ; в регистр cl заносят 7, так как маска поля b2 = 10000000b |

---

Часть 12

**15.1. ОБЩИЕ СВЕДЕНИЯ.**

# 15.1.1. Понятие о макросредствах Ассемблера.

Для облегчения работы при программировании задач на Ассемблере разработчики компиляторов включили в язык аппарат макросредств.

Транслятор с Ассемблера состоит из 2 частей – непосредственно **транслятор**, формирующий объектный модуль, и **макроассемблер**, обрабатывающий макросредства. При компилировании проги сначала работает макроассемблер, а затем – сам транслятор.

Основная идея введения макросредств – использовать подстановки, которые замещают определенным образом организованную символьную последовательность другой символьной последовательностью. Создаваемая таким образом последовательность может быть как последовательность, описывающая данные, так и последовательность программных кодов.

Главное здесь то, что на входе макроассемблера может быть текст проги, далекий по виду от проги на Ассемблере, а на выходе – текст на чистом Ассемблере.

К макросредствам относят макро-определения, макро-имена и макро-директивы. Макроопределение (/ **макрос**) – это блок исходных команд / директив Ассемблера, определяемый особым образом с помощью макро-директив и имеющий имя. Макро-директивы – специальные директивы для обработки макро-определений. Макро-команда – это имя макроопределения, вместо которого в текст проги подставляют группу команд / директив Ассемблера.

# 15.1.2. Преимущества и недостатки использования макросов.

«+» – это уменьшение объема текста проги и сокращение времени разработки проги. Макросы помогают прояснить логику проги, заменяя зашифрованные мнемоники языка Ассемблер на макро-имена, имеющие удобочитаемый вид. В Ассемблере можно подключать несколько файлов с исходным текстом проги к одному файлу директивой include, так можно создавать и использовать библиотеки макросов.

«-»: макросы хранят в виде текста и, следовательно, должны транслировать отдельно для каждой отдельной проги. Поэтому обширная библиотека макросов может увеличивать время транслирования проги, особенно если используют небольшое количество макросов из одной большой библиотеки. Упрощая и проясняя Ассемблер, макросы могут скрывать результаты работы индивидуальных команд. Например, макрокоманда, изменяющая значение какого-то регистра. Макросы требуют подробного документирования использования регистров, флагов и переменных в памяти.

---

Часть 13

**15.2. СОЗДАНИЕ И ВЫЗОВ МАКРООПРЕДЕЛЕНИЙ.**

**15.2.1. Создание макроопределений.**

Макроопределение (/ макрос) - блок исходных команд и директив Ассемблера, начинается директивой **macro** и заканчивается директивой **endm**:

**имя\_макрокоманды macro [формальный\_параметр1,**

**формальный\_параметр2, …]**

**… ; команды и директивы, образующие тело макроопределения endm**

Макроопределения могут располагаться:

1) в исходном тексте проги до места, в котором их применяют (до макрокоманд); этот вариант идет, если определяемые вами макрокоманды актуальны в пределах одной этой проги.

2) в отдельном файле; чтобы сделать доступными эти макроопределения в конкретной проге, надо в начале исходного текста проги записать директиву **include имя\_файла.**

Недостаток второго способа: в исходный текст проги включают абсолютно все макроопределения. Для исправления этого используют директиву **purge**, операнд, которой через запятую перечисляет имена макросов, которые не должны включаться в текст проги. Пример:

message macro msg1

|  |  |
| --- | --- |
| lea dx, msg1 | ; в dx загрузить эффективный адрес ячейки msg1,  ; передаваемой как параметр макроопределению |
| mov ah, 9 | ; функция вывода строки на экран |
| int 21h  endm. | ; прерывание DOS |

**15.2.2. Вызов макроопределений.**

Чтобы в текст проги был вставлен блок из макроопределения, используют макрокоманды:

**имя\_макрокоманды [фактический** параметр1**,** фактическийпараметр2**,** …**]**

Результат применения такой синтаксической конструкции в исходном тексте проги - ее замещение строками из тела макроопределения. Но это не простая замена. Макрокоманда имеет некоторый список фактических параметров, которыми корректируется макроопределение. Места в теле макроопределения, которые будут замещаться фактическими параметрами из макрокоманды, обозначают с помощью **формальных параметров.**

Таким образом, в результате применения макрокоманды в проге формальные аргументы в макроопределении замещают соответствующими фактическими аргументами. Процесс такого замещения называется **макрогенерацией**, а результатом этого процесса - макрорасширение. Пример:

sum macro op1, op2

add op1, op2

xor op2, op2

endm

…

mov ax, 5

mov cx,10

sum ax, cx ; занесение в регистр ax суммы cx и ax, обнуление cx

; (после этого данного фрагмента в ax будет 15, в cx – 0)

Каждый фактический параметр - это стока символов, для их формирования применяют правила: 1) строка не может содержать символы пробел, точка, запятая, точка с запятой, 2) чтобы указать, что некоторый символ вну3 строки, представляющей фактический параметр, является символом (а не чем-то иным вроде разделителя / ограничивающей скобки), применяют спец оператор ! (восклицательный знак). Его ставят перед описанным выше символом. Заключение группы символов в угловые скобки <> эквивалентно действию оператора ! на каждый из них, 3) если надо вычислить в строке константное выражение, то в его начале ставят знак ‘%’ (% константное\_выражение – значение вычисляется и подставляется в текстовом виде).

---

Часть 14

**15.3. МАКРООПЕРАТОРЫ И МАКРОДИРЕКТИВЫ.**

# 15.3.1. Макрооператоры &, <>, !, %, ;;.

Макрооператор **&** (амперсант) нужен, чтобы параметр, переданный как операнд макроопределению / макро-директиве, заменялся значением до обработки строки макроассемблером. Пример:

; макрос выполнит команду push eax, если его вызвать как pushreg pushreg macro letter

push e&letter&x

endm

Макрооператор **<>** (угловые скобки) действует так, что весь текст, заключенный в эти скобки, рассма3вают как текстовая строка, даже если он имеют пробелы / другие разделители. Этот оператор используют при передаче текстовых строк как параметры для макросов.

Макрооператор **!** (восклицательный знак) используют аналогично угловым скобкам, но действует только на 1 следующий символ, если этот символ – запятая / угловая скобка, он все равно будет передан макросу как часть параметра.

Макрооператор **%** (процент) указывает, что находящийся за ним текст – выражение и должен быть вычислен, чтобы передавать в качестве параметра в макрос не само выражение, а его результат.

Макрооператор **;;** (2 точки с запятой) – начало макро-комментария. В отличие от обычных комментариев, текст макро-комментария не попадает в листинг и в текст проги при подстановке макроса.

# 15.3.2. Макро-директивы.

С помощью макросредств Ассемблера можно не только частично изменять входящие в макроопределение строки, но и модифицировать сам набор этих строк и даже порядок их следования с помощью 2х групп макро-директив:

1) макро-директивы повторения (**while, rept, irp, irpc**) – для создания макросов, содержащих несколько идущих подряд 1аковых последовательностей строк; возможна частичная модификация этих строк;

2) макро-директивы управления процессом генерации макрорасширения (**exitm, goto**) – можно как исключать отдельные строки из макрорасширения, так и прекращать процесс генерации.

Макро-директива **local** (local – локальный) – для локализации набора меток вну3 макроопределения:

**local формальное\_имя1[, формальное\_имя2,…]**

Её используют для организации корректного многократного вызова макроопределений, в которых используют метки. Для обеспечения уникальности имен меток, определенных в local, для любого формального имени списка, порождается имя **??xxxx**. То есть 2 символа ‘?’ и xxxx – 16 битное число в диапазоне 0 .. 0ffffh (при каждой макрогенерации для указания имени очередной метки выбирают число, на 1 > предыдущего).

Директиву local используют только в макроопределениях и идет сразу после директивы **macro**. Пример:

; в макросе вычисляют сумму 2 операндов и результат сравнивают с

; числом 100; если результат сложения > 100, то идет переход на метку

; L1, иначе – на L2

sum macro op1, op2

local L1, L2 ; локализация меток

add op1, op2

cmp op1, 100

jg L1

mov op2, 5

jmp L2

L1:

xor op2, op2

L2:

endm

---

**Часть 15**

**16. ПЛАТФОРМА .NET. IL ASSEMBLER**

**16.1. ОБЩАЯ СХЕМА АРХИТЕКТУРЫ .NET.**

Исходные тексты программ компилируют в спец промежуточное представление (Microsoft Intermediate Language, сокращение – IL, MSIL).

Промежуточное представление содержит всю нужную инфу о проге, но не привязано к определенному языку программирования / к машинному коду какой-то целевой платформы. Для запуска проги надо специальное окружение, исполняющее проги, и библиотеки динамической поддержки (execution engine & runtime).

Особенность трансляции в .NET – промежуточное представление не интерпретируют, а используют механизм компиляции времени исполнения, который генерирует машинный код. Большинство программ на платформе .NET, использует много стандартных классов, предоставляющие базовую функциональность (от работы с данными до встроенных механизмов безопасности).

# 16.2. ДОСТОИНСТВА ПЛАТФОРМЫ .NET

Платформа .NET основана на единой объектно-ориентированной модели; все сервисы оформлены как единая иерархия классов.

Благодаря тому, что промежуточное представление .NET не привязано к какой-то платформе, приложения, созданные в архитектуре.NET – много-платформенные.

Платформа .NET дает автоматическое управление ресурсами. Это решает много проблем: утечки памяти, повторное освобождение ресурса и т.п. В .NET нет никакого способа явно освободить ресурс!

Одна из наибольших трудностей при развертывании приложения – использование разделяемых библиотек. Из-за этого установка нового приложения может привести к прекращению работы ранее установленного приложения. В архитектуре .NET установка приложения – простое копирование всех файлов в определенный каталог. При установке используют криптографические стандарты для придания разной степени доверия модулям приложения.

Приложения .NET не используют реестр Windows за счет использования механизма метаданных.

Код для .NET можно проверить на безопасность для гарантии, что приложение не может навредить пользователю и поэтому приложения для .NET могжно сертифицировать на безопасность.

Обработка ошибок в .NET – через механизм исключительных ситуаций. Это решает неоднозначность ситуации, когда некоторые ошибки обозначают с помощью кодов ошибки платформы Win32, некоторые возвращают HRESULTS и т.п.

Самое большое обещание .NET – межъязыковое взаимодействие (language interoperability), единая модель, позволяющая на равных пользоваться разными языками для создания приложений. Так как MSIL не зависит от исходного языка / от целевой платформы, в .NET развивают новые проги на базе старых программ – причем и первый, и второй языки программирования не важны!

Для такого подхода к разработке программ обеспечена межъязыковая отладка. Visual Studio.NET поддерживает этот процесс прозрачно для пользователя и не делает различий между языками, на которых было написано исходное приложение.

# 16.3. НЕДОСТАТКИ ПЛАТФОРМЫ .NET

Самый большой – сильное замедление выполнения проги, так как между исходным языком и машинным кодом вводят дополнительный уровень, **MSIL**. Однако промежуточное представление .NET с самого начала проектировалось с прицелом на компиляцию времени исполнения (в отличие, например, от Java bytecode, который разрабатывался с прицелом на интерпретацию).

Это дает дополнительные возможности по борьбе с замедлением. Например, можно равномерно распределить замедление при запуске, так как обычно компилируют не всю библиотеку, а только метод, который вызывается, и повторной компиляции одного и того же метода не делают.

Другая проблема .NET: при ее создании основной упор был на С++/Java-подобные языки (например, конструкторы с именем, равным имени метода, запрет множественного наследования и т.п.). Это ограничивает возможности интеграции некоторых языков с более богатыми возможностями, особенно с функциональными языками (ML, Haskell, Scheme). Разработчикам компиляторов удается реализовать "проблемные" особенности исходных языков в рамках .NET,

# 16.4. ОСНОВЫ IL ASSEMBLER

MSIL – ассемблер некоторой виртуальной машины. Это нетипичный ассемблер, имеет конструкции, характерными для языков более высокого уровня: например, в нем есть инструкции для описания пространств имен, классов, вызовов методов, свойств, событий и исключительных ситуаций.

MSIL - стековая машина со статической проверкой типов для отслеживания типичных ошибок.

MSIL – это дополнительный уровень абстракции, позволяющий переносить код с одной платформы на другую даже с изменением разрядности платформы: в отличие от Java bytecode MSIL не завязан на 32 бита / какую-то другую фиксированную разрядность. Есть версии MSIL для мобильных 16-разрядных устройств (.NET Compact Framework), стандартная 32-разрядная версия и специальная версия для работы с 64-разрядными устройствами.

MSIL сохраняет много инфы об именах, использованных в исходной проге: имена классов, методов и исключительных ситуаций сохраняют и могут быть извлечены при обратном ассемблировании. Но извлечение из MSIL исходных текстов путем дизассемблирования не имеет смысла, так как имена локальных переменных, констант и параметров сохраняются только в отладочной версии.

Проги на IL Assembler можно писать в любом текстовом редакторе. Для компиляции исходного текста проги применяют **ilasm.exe**, поставляемый вместе с .NET Framework. Рассмо3м прогу hello world на языке IL Assembler.

// hello.il

/\* Prints greeting on the console \*/

.assembly extern mscorlib {}

.assembly Hello { .ver 1:0:1:0 }

.module hello.exe

.**method** static **void** main() **cil managed**

{

.maxstack 1

.entrypoint

010 ldstr "hello, world"

011 **call void** [mscorlib]System.Console::WriteLine (string)

012  **ret**

013 }

Комментарии в IL Assembler обозначают символами // (1строчный комментарий) и /\*…\*/ (блочный комментарий). В строке 3 – директива подключения внешней библиотеки **mscorlib** (директивы начинают с символа ‘.’). Затем идет дополнительная инфа об ассемблерной сборке (имя, номер версии, и т.д.).

В строке 5 – имя исполняемого модуля программы (в каждой сборке должен быть хотя бы 1 модуль). В строках 6-13 – статический метод - процедура **main()**, не имеющая входных параметров; **cil** managed указывает компилятору, что должен генерироваться **managed code** (код, в котором выделением памяти управляет сама система, а не прогер).

В первой строке процедуры (строка 8) задают максимальное количество элементов, которые будут загружать в память или в стек за 1 раз. Директива **.entrypoint** указывает, что данный метод запускают автоматически при загрузке проги. Команда **ldstr** (строка 10) загружает строку в память или стек.

**call void [mscorlib]System.Console::WriteLine (string)** вызывает метод библиотеки mscorlib для печати сообщения на экран консоли. Параметром передается не переменная, а тип данных string, сама строка считывается из стека. Любой метод завершают командой возврата **ret**. Стек освобождают автоматически сборщиком мусора платформы .NET.

Следующая прога складывает 2 числа и выводит результат на экран. Последовательность команд ldc.i4.s и ldc.i4 загружают 2 числа в память одновременно (s - simultaneously). Первая команда загружает первый байт четверки (i4), а вторая второй. Команда **add** считывает 2 числа из стека, складывает их и помещает результат в стек.

|  |  |
| --- | --- |
| 001  002  003  004  005  006 007  008  009  010  011  012  013  014 | //Add.il  //Сложение 50 и 30  .assembly extern mscorlib {}  .assembly Add { .ver 1:0:1:0 }  .module add.exe  **.method** static **void** main() **cil managed**  {  .maxstack 2 .entrypoint ldstr "50 + 30 = "  call void [mscorlib]System.Console::Write (string) ldc.i4.s 50 ldc.i4 30 add |

|  |  |
| --- | --- |
| 015  016  017 | **call void** [mscorlib]System.Console::Write (int32) **ret**  } |

---

ФСЁ.