КРАТКО СОДЕРЖАНИЕ:

МОДУЛЬ 1

Ч2. 1.2. Регистровая архитектура процессора Пентиум

# **1.2.1. Общие сведения:** реальный реж, Защищенный реж, АЛУ, рег-ы**.**

## 1.2.3. Рег-ы индексов (esi, edi) и указателей (ebp, esp – для стека).

## 1.2.4. Сегм-тные Рег-ы (cs, ss, ds, es, fs, gs).

## 1.2.5. Рег указателя инструкций (eip).

## 1.2.6. Рег Фл. (eflags).

Ч 3. 2.1. Структура проги на яз Ассемблер

# **2.1.1. Первичные эл-ты языка Ассемблер:** Идентификатор, Переменные, Метка, Оператор, Команда, Директива**.**

# **2.1.2. Сегм-ты в Ассемблере: стека, данных, кода.**

# **2.1.3. Структура com-проги: PSP, ip, sp.**

Ч 4. 2.2. Директивы сегментации.

**2.2.1. Стандартные директивы сегм-ации имя segment тип выравнив, тип комбин, класс, тип размера.**

**2.2.2. Упрощенные директивы сегм-ации .model операнд .stack р-р .code имя .data. @**

Ч 6. 3.2. Директивы определения данных

**3.2.2. Директивы определения скалярных данных db,** dw, dd, **«?», dup**.

Ч 7. 3.3. Описание символических имен.

**3.3.1. Директива “=”.**

**3.3.2. Директива equ.**

МОДУЛЬ 2.

Ч1. 5.1. Режимы адресации.

5.1.1. **Общие сведения:** 4 реж адресац (рег-ый, непоср-ый, прямой, косвен)**.**

# **5.1.2. 4 Режима адресации.**

Ч 2. 5.2. Команды пересылки **(mov, xchg, bswap).**

# **5.2.1. Команда пересылки mov** (для адресации).

# **5.2.2. Команда замены - xchg.**

**5.2.3. Команда перестановки байт bswap.**

# **5.2.4. Загрузка сегм-тных Рег-ов lds,** les, lfs, lgs, lss.

**5.2.5. Загрузка эффективного адреса lea.**

Ч 3. 6.1. Команды работы со стеком.

**6.1.1. Понятие стека. Адресация данных в стеке push**, **pop.**

**6.1.2. Стековые команды push**, **pusha, pop, popa.**

Ч 4. 6.2. Команды портового вв-выв.

# **6.2.1. Понятие порта. Адресация портов.**

# **6.2.2. Команды портового вв-выв in, out.**

Ч5. 7.1. Арифметические операции над целыми 2ичными числами.

# 7.1.1. Команды сложения: add, adc, inc, xadd.

# 7.1.2. Команды вычитания: sub, sbb, dec, neg, cmp.

# 7.1.3. Команды умножения \* mul (б/зн), imul (знаков).

# 7.1.4. Команды деления: div, idiv.

Ч 6. 7.2. Команды коррекции 2чно-10чных чисел.

7.2.1. Общие сведения.

7.2.2. Коррекция арифметики неупакованных BCD-чисел – ааа, aas, aam, aad.

7.2.3. Команды коррекции упакованных BCD-чисел daa, das.

Ч 7. 7.3. команды преобразования типов.

7.3.1. Команды расширения знакового бита (с минусом) cbw, cwd, cwde, cdq. (из al в ax)

7.3.2. Команды пересылки с- (movsx) и без- (movzx) учета знакового бита.

Ч 8. Операции над одним операндом + команды сравнения.

7.4.1. 1 операндные команды инкремента, декремента и изменения знака операнда inc, dec, neg.

7.4.2. Команды сравнения cmp, cmpxchg, cmpxchg8b.

Ч 9. Команды работы с регистром флагов.

8.1.1. Команды установки (1) и сброса (0) отдельных Фл.

8.1.2. Сохранение и загрузка Рег Фл. pushf и popf, sahf и lahf.

Ч 8. Команды установки байт по условию.

8.2.1. Команды установки байта по условию setcc, setge (>=), setg (>) setnz.

Ч 11. 8.3. Спец команды.

8.3.1. Команда пустой операции nop.

8.3.2. Команда останова Проц-ра hlt.

Ч 12. 9.1 Логические команды.

9.1.1. Общие сведения: команды и, или, исключающее или, не.

9.1.2. Логические команды (команды логических операций) and, test, or, xor, not.

Ч 13. Команды сдвгов.

9.2.1. Общие сведения sal, shl, shl, shr, shld, shrd, rol, ror, rcl, rcr.

# 9.2.2. Арифметические сдвиги sal, sar – деление, умножение\*.

# 9.2.3. Логические сдвиги shl, shr, shld, shrd.

9.2.4. Циклические сдвиги rol, ror, rcl, rcr.

Ч 9. 9.3. Команды манипуляции отдельными битами.

9.3.1. Команды поиска установленного в 1 бита bsf, bsr – поиск I бита, =1.

Ч 15. 10.1. Команды переходов far, near..

# 10.1.1. Общие сведения Far-, Near-, eip/ip, cs.

# 10.1.2. Команда безусловного перехода jmp.

1) короткий относительный внутрисегм-тный переход: относительный – потому что в формате машинной команды хранят относительное смещение/ расстояние от jmp до метки;

jmp short метка, jmp метка (-128..127)

2) относительный переход в пределах всего сегм-а;

jmp near ptr метка, jmp метка (в любую точку)

3) внутрисегм-тный косвенный переход в пределах сегм-а;

jmp источник

Меж-сегментный (между сегментами) переход:

4) прямой межсегм-тный переход; прямой – потому что в формате Маш Ком - полный адрес перехода;

jmp метка\_типа\_far, jmp far ptr метка.

5) косвенный межсегм-тный переход.

jmp источник

# 10.1.3. Команды переходов по условию (условных переходов) jecxz и jcxz, jcc.

Ч 16. 10.2. Команды упрвнения циклами loop.

# 10.2.2. Команды управления циклами loop, loopw, loopd, loope/loopz, loopew/loopzw, looped/loopzd, loopne/loopnz.

# 10.2.3. Работа с командами управления циклами.

------- ЧАСТЬ 3 -----------

11.1. Команды обработки строк.

# 11.1.1. Общие сведения movs, cmps, scas, lods, stos, ins, outs.

11.1.3. Строковые команды movs.

# 11.1.4. Работа с командами обработки строк ds, es.

Ч 2. 11.2. Табличное преобразование.

11.2.1. Команда табличного преобразования xlat.

Ч 3. 12.1 Основные сведения о прерываниях.

12.1.1. Понятие прерывания. Аппаратные и программные прерывания.

Ч 4. 12.2 Команды работы с прерываниями.

12.2.1. Вызовы прерывания int, int 3, bound.

12.2.2. Корректный выход из обработчика прерывания (из запоя ☺) iret.

Ч 6. 13.1. Описание процедур. Команды работы с ними.

13.1.1. Описание процедур в Ассемблере proc, endp.

13.1.2. Команды работы с процедурами call, ret, enter, leave.

Ч 7. 13.2. Передача параметров процедурам.

# 13.2.2. Передача параметров через Рег-ы (1й способ).

# 13.2.3. Передача параметров через глобальные переменные (2й способ).

# 13.2.4. Передача параметров через стек (3й способ).

Ч 8. 14.1. Выражения и операторы.

14.1.1. Общие сведения: арифм, логич, сравнен.

14.1.2. Операторы, присваивающие атрибуты ptr, short, this.

14.1.3. Операторы, возвращающие значение high, low, seg, offset, type, length, size, $.

Ч 9. 14.2. Массивы.

**14.2.1. Объява массива. Работа с массивами.**

Ч 10 14.3. Структуры.

# **14.3.1. Объява структур. Работа со структурами struc,** оператор **.** (точка)**.**

Ч 11. 14.4. Записи.

**14.4.1. Объява записей. Работа с записями record, width, mask.**

Ч 12. 15.1. Общие сведения о макросредствах, макросах.

# **15.1.1. Понятие о макросредствах Ассемблера.**

**15.1.2. Преимущества и недостатки использования макросов.**

Ч 13. 15.2. Создание и вызов макроопределений.

**15.2.1. Создание макроопределений macro**, **endm.**

Ч 14. 15.3. Макрооператоры и макродирективы.

# **15.3.1. Макрооператоры &, <>, !, %, ;;.**

Ч 15. 16. Платформа .Net.

16.1. Общая схема архитектуры.

Везде, где букава «е» - **32** битн режим адресации:, **e**sp, eax, eip, eflags, esi, edi, ecx, ss:esp.

Где 2 буквы и нет «е» - 16 битный sp, ax, ip, flags, si, di, cx, ss:sp.

|  |  |  |
| --- | --- | --- |
| Рег общего назначения | | |
| **Eax -**32битн | аx 16 битн реж адресации  **аккумулятор** | al + ah 8 битн реж адресации |
| **ebx** | bx - **базовый** | bh + bl |
| **ecx** | cx - **Рег-счетчик** | ch + cl |
| **edx** | dx - Рег **данных** | dh + dl |
| **eip/ip** | указатель инструкций (команд): Рег указателя инструкций |  |
| **Eflags** | указатель Фл: Рег Фл |  |
| Рег-ы сегм-тов (сегментные рег): | | |
| **Cs** | code segment (адрес начала сегм-а кода) | сегм-т имеет команды, которые отправляются на выполнение Проц-ру; |
| **Ss** | stack segment (адрес начала сегм-а стека); |  |
| **Ds** | data segment (адрес начала сегм-а данных); | сегм-т имеет данные, которые обрабатывает прога |
| **Es** | Extra Segment | адреса начал д/п-ных сегм-тов данных; использ если не хватает объема основн сегм-а данных. |
| **fs** |  |
| **Gs** |  |
| Рег-ы индексов (esi, edi) | | |
| esi/si | source index register (инд источника). | имеет текущий адрес эл-та в цепочке-источнике; |
| edi/di | destination index register (инд приемника) | имеет текущий адрес эл-та в цепочке-приемнике. |
| Рег-ы указателей | | |
| **esp/sp** | stack pointer register (Рег указателя стека) | указатель ее вершины |
| **ebp/bp** | base pointer register (Рег указателя базы). | Для произвольного доступа к данным внутри стека |
| **eip/ip** | Рег указателя инструкций |  |
| Рег Фл. eflags - инфа о сост вычислит-го процесса д/принятия прогой решен | | |
| **CF** | carry flag (Фл. переноса) | = 1, если был перенос при «+» / заем при «-», иначе – 0; |
| **ZF** | zero flag (Фл. нуля) | = 1, если рез-т = 0, иначе – 0 (=1); |
| **SF** | sign flag (признак знака) | = 1, если рез отрицательный, иначе – 0; |
| **DF** | direction flag (Фл. направления) | управляет порядком обработки операнда; в строковых командах; |
| **OF** | (overflow flag) | Фл. знакового переполнения. |
| **Rep** | префикс повторения | повтор команды сколько раз указано в ecx/cx |
| **PSP** | program segment prefix | Блок памяти, префикс программного сегмента |

**1.1. АРХИТЕКТУРА ПЕРСОНАЛЬНОГО КОМПА**

## 1.1.4. Пространство портов вв-выв.

Каждый порт имеет свой №; для дешифрации адресов портов используют только младшие 10 бит, что позволяет обращаться к портам в диапазоне адресов **0-255**. Обмен инфой м/у Проц-ром 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.

# Табл 1.1. Стандартные **порты** вв-выв

|  |  |
| --- | --- |
| **Адрес** | **Назначение** |
| 020h – 021h | контроллер прерываний #1 |

## 1.1.5. Прерывания (аппаратные и программные).

Аппаратные передаются по спец линиям системной шины, запросы от внешних устройств (нажатие клавиши). Программные - для вызова спец подпрог, есть спец команды - по команде **int 21h** вызов прерыван ОС DOS.

**Контроллер прерываний** - для выбора события, устанавливает приоритет.

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

|  |  |  |
| --- | --- | --- |
| Имя | Вектор | Описание |
| IRQ1 | 09h | клавиатура |

## 1.1.6. Прямой доступ к памяти. DMA.

Реж прямого доступа к памяти (DMA – direct memory access) –периферийное устройство связано с Опер памятью напрямую - высокая скорость обмена большого кол-ва инфы.

Ч 2 **1.2. РЕГ-ОВАЯ АРХИТЕКТУРА ПРОЦ-РА PENTIUM.**

# **1.2.1. Общие сведения:** реальный реж, Защищенный реж, АЛУ, рег-ы**.**

2 реж работы Проц-ра Pentium:

1) реальный реж – м использовать **только первый** мегабайт Опер Пам-и – прога монопольно использует все ресурсы компа;

2) Защищенный реж – адресация до 4 Гбайт физической памяти и механизм защиты (процессор может разрешает / нет выполнять действие.).

**АЛУ** - выполняет целочисленные операции, используя операнды длиной 8, 16 и 32 бита.

Ре-гы – ячейки памяти Проц-ра. 32 Рег, которые делят на 2 группы: **пользовательские и** системные **Рег-ы.**

**1) пользовательские Рег-ы**: 1. Рег-ы общего назначения; 2. указатель инструкций (команд); 3. указатель Фл.; 4. Рег-ы сегм-тов; steck, data, code segment. Рассмотрим **пользовательские Рег-ы**.

**1.2.2. Рег-ы общего назначения (eax, ebx, ecx, edx).**

расширенная eax -32bit = 4 байта. ax = ah + al, bx = bh + bl, cx = ch +cl. Размер - 1 байт (**al – low, ah – high**), 2 байта (**ax**) и 4 байта (**eax** – **extended**).

Стандартные назначения этих 4 Рег-ов:

1) eax/ax/ah/al - **accumulator** register (**аккумулятор)**. Для хранения промежуточных данных. обязательно в умножении MUL и IMUL;

2) ebx/bx/bh/bl - **base** register (**базовый**). Хранит базовый адрес объекта в памяти. адресация данных в **MUV**;

3) ecx/cx/ch/cl - **count** register (**Рег-счетчик**). Для команд с повторами. команда организации цикла **loop**, уменьшает на 1 значение Рег ecx/cx;

4) edx/dx/dh/dl - **data register** (Рег **данных)**. хранит промежуточные **данные**. Обязательно в делении **DIV** и **IDIV**.

## 1.2.3. Рег-ы индексов (esi, edi) и указателей (ebp, esp – для стека).

2 Рег для строковых / цепочечных операций – обрабатывают послед-ти 1типных эл-тов. Их структура: si, di, **sp, bp** – 2байт, esi, edi, **esp, ebp** – 4 б.

1) **esi/si** - **source index** register (индекс источника). имеет текущий адрес эл-та в цепочке-источнике;

2) **edi/di** - **destination index** register (индекс приемника (**получателя**)). имеет текущий адрес эл-та в цепочке-приемнике.

Cтек: структура – LIFO. В стеке есть прямой доступ к содержимому стеку.

1) **esp/sp - stack pointer** register (Рег указателя стека). указатель ее вершины.

2) **ebp/bp - base pointer** register **(**Рег указателя базы). Для **произвольного** доступа к данным внутри стека.

## 1.2.4. Сегм-тные Рег-ы (cs, ss, ds, es, fs, gs).

Сегм-т – область памяти, где любой байт м адресовать только с помощью смещения. Проц-р использует сегм-тную модель адресации Опер Пам-и, формулу для определения адреса:

**address = segment × 16 + offset**

где **address** – вычисляемый адрес, **segment** – сегм-т / часть адреса, **offset** – оператор, возвращающий смещение перем-ой / метки относительно данного сегм-а, segment и offset – 16 битные Рег-ы.

Прога имеет сегм-ты стека, данных и кода (cs, ss, ds). Есть неск-ко моделей памяти. Для модели памяти **small** 2 сегм-а: 1 – под коды команды, 2й – под данные + стек.

В сегм-тной организации памяти - 6 сегм-тных Рег-ов, длина каждого –16 бит (2 байта).

* **cs - code segment** (адрес начала сегм-а кода); сегм-т имеет команды, которые отправляются на выполнение Проц-ру;
* **ds - data segment** (адрес начала сегм-а данных); сегм-т имеет данные, которые обрабатывает прога;
* **ss - stack segment** (адрес начала сегм-а стека);
* **es, fs, gs** – адреса начал д/п-ных сегм-тов данных; его используют если не хватает объема основного сегм-а данных.

## 1.2.5. Рег указателя инструкций (eip).

Рег **eip** – **extended instruction pointer**, разрядность 32 бита. К младшим 2 байтам м обратиться по отдельности – **ip**. имеет смещение относительно начала сегм-а кода **cs**, указывающее на такую команду. Прямого доступа **нет**, но он меняется при использовании в проге безусловных и условных переходов, + при вызове процедур и прерываний и при возврате из них.

## 1.2.6. Рег Фл. (eflags).

В Рег Фл. eflags регистрируют инфу (**признаки**) о состоянии вычислительного процесса для принятия прогой **решений**. Длина 4 байт.

1) CF - **carry** flag (Фл. **переноса)** = 1, если был перенос при «+» / заем при «-», иначе – 0;

4) ZF - **zero** flag (Фл. **нуля) =** 1, если рез-т = 0, иначе – 0 (=1);

5) SF - **sign** flag (признак знака) = 1, если рез-т отрицательный, иначе – 0;

8) DF - **direction** flag (Фл. направления) – управляет порядком обработки операнда; в строковых командах;

9) OF (**overflow** flag) – Фл. знакового переполнения.

Часть 3. **2.1. СТРУКТУРА ПРОГИ НА ЯЗЫКЕ АССЕМБЛЕР.**

# **2.1.1. Первичные эл-ты языка Ассемблер:** Идентификатор, Переменные, Метка, Оператор, Команда, Директива**.**

1) Идентификатор - набор латинских букв, цифр + спец символов: \_, ?, $, @.

Идентификатор нельзя начинать с цифры.

2) Переменные - опред-ют **данные**, имеют 3 атрибута: а) **сегм-т -** где лежит эта переменная; б) **смещение** относительно **начала** сегм-а; в) **тип** - определяет число байт под данную переменную.

3) Метка. Определяет **адрес** в памяти младшего байта команды. для организации переходов. если в адресе перехода - метка, то указан адрес команды, на которую пойдет **переход**. имеют 2 атрибута: сегм-т и смещение.

4) Оператор – команда, псевдо-оператор / директива и макрокоманда. На этапе трансляции команды преобразуют в инструкции Проц-ра (**Маш Ком**); директивы несут указания самому транслятору.

5) **Команда** Ассемблера м иметь до 4 полей и быть на 1 строке:

**[метка :] мнемо-код [операнд]** [; комментарий]

Пр-р команды:

**pov: mov ax, 5** ; занесение в Рег ax значения 5, **pov** – метка [],

; **mov –** мнемокод операции (**команда**), ax и 5 – **операнды**,

6) **Директива** имеет формат:

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

Пр-р директивы определения имени:

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

компиляции б пониматься как Рег.

# **2.1.2. Сегм-ты в Ассемблере: стека, данных, кода.**

Сегм-т – область в тексте проги (кусок проги?). **Программный сегм-т** состоит из сегментов 3х видов:

1. **стека (**здесьотводят место под стек**) -** для доступа к стеку; смещение - в **esp/ebp**. **.stack segment**
2. **данных (**имеетданные**)** – для доступа к операндам команд; **.data segment**
3. **кода (**имеет команды), из которого по смещению, заданному в **eip**, считывают очередную машинную команду; **.code segment**

Прога-компоновщик (**линкер**) определяет положение сегм-тов Ассемблера в памяти, связь м/у ними и как их адресовать.

# **2.1.3. Структура com-проги: PSP, ip, sp.**

В файлах com-формата для адресации памяти используют 16 битн Рег-ы. физический адрес ячейки памяти находят по формуле:

**address = segment × 16 + offset**

**Программный сегм-т** – блок (в общаге) **памяти** для размещения (по койкам) стека + данных + кода проги. Перед программным сегм-том есть **блок памяти** – префикс программного сегм-а (**PSP – program segment prefix**), длина 256 байт. В PSP есть разная инфа для ОС и проги. После загрузки com-прога занимает 1 физический сегм-т памяти (= 64 Кбайта).

Рег указателя инструкций **ip=100 h** (первые 256 байт отведены под PSP), а Рег указателя вершины стека **sp** = 0fffe h.

# **2.1.4. Структура exe-проги:** логические сегм-ты **ss** и **esp, cs** и **eip.**

Прога в **exe**-формате (ехе-файл?) - для ввода любого кол-ва логических сегм-тов кода и данных. Инфа о логических сегм-ах - в **заголовке** exe-файла. По данным из этого заголовка ОС настраивает стек (**ss** и **esp**) и запускает прогу (устанавливает **cs** и **eip** (рег указат инструкций)).

**ds** (дата сег) и **es** (Extra Seg) указывают на начало PSP (префикс - сени). Смена текущего сегм-а кода - за счет дальних переходов. так м изменить **cs: eip**.

Часть 4. **2.2. ДИРЕКТИВЫ СЕГМ-АЦИИ.**

**2.2.1. Стандартные директивы сегм-ации имя segment тип выравнив, тип комбин, класс, тип размера.**

Структура сегм-а на Ассемблере:

|  |
| --- |
| **имя segment [тип выравнив.] [тип комбин.] [класс] [тип размера]**  … ; команды и директивы  **имя** ends  **а1 segment [byte, word] [stack] [**“**code**”**] [use16]**  **c1 segment para public “code”** ; (cs) объява сегм-а c1, тип выравниван – на границу параграфа, тип комбинир-ния – public, имя класса – code |

Начало логического сегм-а - директива **segment**, конец – директива **ends**. Обе д иметь 1 **имя –** **идентификатор (- определитель – общий признак**).

Параметры директивы **segment**:

А) Атрибут **выравнивания** сегм-а (тип выравнивания) говорит компоновщику («комплектовщику») (редактор связей, линкер), что надо разместить начало сегм-а на этой границе, т к в завис-ти от выравнивания с разной скоростью идет доступ к Опер Пам-и. Его допустимые значения:

− **byte** – выравнив-е не делают; сегм-т м начинаться с любого адреса памяти.

сегм-т начинается по адресу, кратному:

− **word** – 2 (выравнивание - на границу слова).

− **dword** – 4 (вырав-ие - на границу двойного слова).

− **para** – 16 (вырав-ие - на границу параграфа §, размер § – 16 байт).

По умолчанию тип выравнивания имеет значение **para** § – сегм-т загружают в память с кратного 16 адреса.

Б) Атрибут **комбинирования** **сегм-тов** (тип комбинирования) говорит компоновщику как объединять сегм-ты с 1 именем. С разными имен - нельзя.

5 вариантов значений атрибута комбинирования компоновщиком:

1)**private** – сегм-т не объединяют с другими сегм-ами. (приватный)

2) **public** – соединит все сегм-ты с 1аковыми именами. Он б целым и непрерывным. (публичный)

3) **common** – располагает все сегм-ты с 1м именем по 1у адресу. Они б перекрываться и совместно использовать память. (обычный)

В) Атрибут **класса** **сегм-а** (**тип класса**) – заключенный в кавычки идентификатор (имя класса). Сегм-ты с 1м именем класса лежат в памяти друг за другом. Типичный Пр-р - объединение в группу всех сегм-тов кода проги (имя класса “**code**”).

Г) Атрибут **размера сегм-а** (реж адресации) - 2 значения:

1. **use16** – для адресации **инфы** внутри сегм-а – 16 разрядное смещение; р-р этого сегм-а <= 64 Кбайта (**реальный** реж работы Проц-ра).
2. **use32** – при формировании **адреса** внутри сегм-а – 32 разрядное смещение; р-р сегм-а <= 4 Гбайта (**защищенный** реж работы Проц-ра).

Если р-ра сегм-а нет, то его неявно определяет транслятор по **типу** Проц-ра –для какого Проц-ра делать код проги. Тип Проц-ра задает директива:

**.тип\_Проц-ра**

Пр-р: **.486** ; реж работы Проц-ра 80486

Все сегм-ты =правны. Чтобы их использовать как сегм-ты кода, данных, стека, надо сказать транслятору об этом директивой **assume** (какой сегм-т к какому сегм-тному Рег-у привязан). Ее формат:

**assume сегм-ыйРег : имяСегм-а[, сегм-ыйРег : имяСегм-а, …]**

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

Сегм-ый Рег **cs** загружают автоматически, а в остальные - загрузку данных делает прогер.

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

**end метка** (например, **end STARN, end Begin**)

Здесь метка определяет команду, которая д б выполнена первой после загрузки проги в память, т е точку входа в прогу (стартовый адрес).

Пр-р структуры проги на Ассемблере:

d1 segment ; объява сегм-а d1 (ds)

… ; определение данных проги

d1 ends ; конец сегм-а d1

st1 segment stack ; объява сегм-а st1, тип комбинирования – stack (ss)

db 256 dup (?) ; отводим место под стек

st1 ends ; конец сегм-а st1

; ------------------------------------------------

**c1 segment para public “code” ; (cs) объява сегм-а c1, тип выравнивания –**

**; на границу параграфа, тип комбинир-ния – public, имя класса – code**

assume cs:code, ds:d1, ss:st1 ; привязывание сегм-тных Рег-ов

; к логическим сегм-ам (сегм-ам проги)

**Begin:** ; точка входа в прогу (Метка, признак – «:»)

… ; команды Ассемблера

c1 ends ; конец сегм-а c1 (cs)

; --------------------------------------------

|  |  |
| --- | --- |
| c2 segment “code” | ; сегм-т c2, имя класса – code, в памяти б располож после сегм-а c1 |
| assume cs:c2 |  |
| … | ; команды Ассемблера |
| c2 ends | ; конец сегм-а code |
| **end Begin** | ; Директива end - в конце проги, установка стартов метки |

Кол-во логических сегм-тов - любое, имена - любые. Директив assume – неск-

ко, а метка точки входа в прогу (**Begin**) - в любом месте внутри сегм-а.

;Пр-р проги со стандартными директивами сегм-ации - выводит

; строку **message** на экран:

**data segment** ; сегм-т данных

**Мessage db ‘Hello, world! $’** ; определение строки текста, символ ‘$’ –

; признак конца печати при выводе на экран

**data ends**

**stk segment stack** ; сегм-т стека

**db 256 dup (?)** ; отводим место под стек

**stk ends**

**code segment** ; сегм-т кода

**Main:** ; Метка начала проги (признак – «:»)

**assume cs:code, ds:data, ss:stk** ; привязываем сегм-тные Рег-ы (cs, ds, ss)

; к логическим сегм-там (code, d1, st1)

**mov ax, data** ; заносим в **ax** сегм-тную компоненту адреса сегм-а data

**mov ds, ax** ; копируем содержимое ax в ds, напрямую загрузить

; инфу в ds нельзя – из реж-ов адресации

**lea dx, Message** ; грузим в **dx** смещение перем-ой Message командой lea

**mov ah, 09h** ; в ah 9 – для прерывания 21h, в этом Рег - № ф-ции, кот

;надо выполнить. 9я ф-ция = вывод строки, смещение, хранят в dx

**int 21h** ; вызываем прерывание, идёт вывод строки текста

**mov ah, 4ch** ; ф-ция 4ch - завершение работы проги и возврат в с-му DOS

**int 21h** ; вызов прерывания с № 21h

**code ends**

**end Main**

Команда **mov** пересылает байт из 1 ячейки памяти в другую. **int** - вызываем подпрогу прерывания с данным №.

**2.2.2. Упрощенные директивы сегм-ации .model операнд .stack 256 .data .code имя.**

- если прога простая (в ней по 1у сегм-ту данных, стека и кода), то ее написание упрощают с помощью упрощенных директив сегм-ации:

**.model операнд**

**.stack [размер]** – начало / продолжение сегм-а **стека. [размер]** **= 256 / 100h**

**.data** – начало сегм-а **данных**

**.code [имя]** – начало / продолжение сегм-а **кода**. **[имя] = Main, Begin, Start.**

Здесь нельзя напрямую (явно) управлять размещением и комбинированием сегм-тов, а управлять неявно - директивой указания **модели** памяти, формат:

**.model операнд** (small, tiny)

Директива **model** размещает сегм-ты и выполняет ф-ции директивы **assume.** Поэтому директиву **assume** здесь м не применять. Пр-ры:

1) **small -** прога в памяти занимает 2 физических сегм-а: 1 - для команд, 2й – для данных + стека. Здесь команды адресуют только с помощью смещения относительно **начала** своего сегм-а. Адресация данных и стека – тоже только смещением относительно **общего** сегм-а (данных и стека).

2) Модель памяти **tiny** - используют 1 физический сегм-т, относительно его начала адресуют и команды, и данные. Пр-р:

; прога вывода строки на экран:

**.model small** ; модель памяти small

**.data** ; сегм-т данных

**message db ‘Hello, world! $’**

.stack 256 ; сегм-т стека занимает 256 байт

**.code** ; сегм-т кода

**Main:**

**mov ax, @data** ; занесение адреса сегм-а данных в Рег ax

**mov ds, ax** ; копируем содержимое Рег ax в Рег ds

**lea dx, message**

**mov ah, 09h**

**int 21h**

**mov ah, 4ch**

**int 21h**

**end Main** ;конец проги с точкой входа main

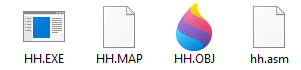
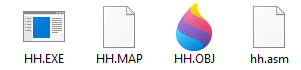
**Физические** адреса сегм-тов определять - спец идентиф-ом **@**:

Здесь д/получения адреса сегм-а данных, кода, стека – **@data**, **@code**, **@stack**

Для отличия систем счисления чисел - в конце **2**чного ставят **b** (бинари). Неявно любой набор цифр - 10чный, но для явного указания - **d** (десымал). Для 8 ричных – **q** (кварта), 16 ричных – **h** (хватит).

Часть 6. **3.2. ДИРЕКТИВЫ ОПРЕДЕЛЕНИЯ ДАННЫХ** (ДОД)**.**

# **3.2.1. Общие сведения.**

Директивы определения данных (ДОД) - для задания размеров и содержимого данных. При обработке ДОД в памяти создают **объектный код** – отводят место под данные. Транслятор преобразует числа, строки и выражения в этих директивах в отдельные **образы** байт, слов и других единиц данных. Эти образы копируют в **объектный файл****.**

ДОД задают: 1) скалярные данные, 2) записи, 3) структуры.

**3.2.2. Директивы определения скалярных данных db,** dw, dd, **«?», dup**.

Формат:

**[идентификатор** (Mess1)**] db** (dw, dd) **значение** (‘Hi! $’)[, значен,..][;комент]

|  |  |  |  |
| --- | --- | --- | --- |
| Мнемон | define | Байты |  |
| **db** | byte | 1 | Распредел-е и инициал-ция 1 байт памяти |
| **dw** | Word | 2 | Распредел-е и инициал-ция **слова** памяти |
| **dd** | double word | 4 | --//--- **двойного слова** |

1) Директива db распределяет и инициализирует 1 байт памяти. Значения: целое число, строковая константа, оператор **dup**, константное выражение, оператор «**?**».

**Оператор «?»** обозначает неопред-ое значение, отсутствие инициализации;

Оператор **dup** – дублирование значений. Если директива имеет имя, то создают переменную типа **byte**. Строковая константа имеет столько символов, сколько ложится на 1 строке (‘Hi! $’). Символы строки хранят в памяти в порядке их следования, I символ имеет младший адрес, последний –старший (нахапал адресов). Пр-р:

**b1 db -16; b2 db ‘abCDf’; b3 db ?**

2) Директива **dw** – распределение и инициализация слова памяти (2 байта). В 16 битн реж адресации сегм-а значение – адрес. Пр-р:

**w1 dw 12, 12h, -2; w2 dw 4\*3**

значения выражений вычисляют только на этапе транслирования транслятором, он не вычисляет арифметические выражения с перем-ой. Поэтому если надо выполнять арифметические и другие действия, в проге используют нужные команды

|  |
| --- |
| **w3 dw offset w1** ; значение перем-ой **w3** = смещению перем-ой **w1**, |

3) Директива **dd**, 4 байт:

Пр-р: **d1 dd 4294967295** ; вмладшее слово памяти занесли смещение

**d2 dd d1** ; в старшее слово– сегм-т, в котором определена **d1**.

Д/определения **массива** целых чисел - оператор дублирования **dup**, формат:

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

Список значений в скобках повторяют в соотв-ии со значением **счетчика**. Значения в скобках - целое число, символьная константа, др оператор **dup** (м до 17 уровней вложения опер-ов **dup**). Пр-р массива:

**val db 5 dup (4 dup (1,2,3))** ; под перемен **val** отвели 60 байт памяти =5\*4\*3

; Массив **val** б проинициализирован знач-ми 1,2,3,1,2,3,1,2,3 и т д.

**mas1 dw 10 dup (0) ;**задан 1мерн массив слов из **10** эл-тов, эл-ты инициализ **0**

Часть 7. **3.3. ОПИСАНИЕ СИМВОЛИЧЕСКИХ ИМЕН.**

**3.3.1. Директива “=”.**

Директива = (знак =ства) имеет формат:

**имя = выражение**

По директиве **“=”** создают имя – значение, =е текущему значению указанного выражения. Пр-р:

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

**b = a + 5** ; имя b заменяют числом **15**

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

Имя м переопределять. В директиве = имя – уникальный идентификатор / идентификатор, ранее использованный как имя другой директивой =. Пр-р:

|  |  |
| --- | --- |
| **a1 dw ?**  … |  |
| **MaxInt = 7fff h** | ; макс-ое 2байтное знаковое число |
| **str = ’AB’** | ; допустимо до 2 символов, данные - в ASCII формате |
| **adr = offset a1** | ; создают символическое имя adr, которое по тексту меняют  ; значением смещения перем-ой a1 |
| **.386** | ; дирек-ва перекл-ия типа Проц-ра, указывает транслятору,  ; для какого Проц-ра создавать код проги, неявно - реж  ; работы Проц-ра 8086 MaxLong = 7 fff fff f h макс-ое 4  ; байтное знак-е число. |

**3.3.2. Директива equ.**

Формат: **имя equ операнд** (число / выражение)

**equ** создает имя, которое в момент трансляции меняют на значение выражения (число) / строку. значение выражения д влазить в 2 / 4 байта. Во всех других случаях имя б строкой. Имя д б уникальным, его нельзя переопределять. Пр-ры:

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

**move ax, 5** ; в Рег **ax** занесли число 5

**Count equ 20** ; символическое имя **Count**, замещаемое в проге числом **20**

**move cx, Count\*3** ; в cx = 60. на этапе транслирования **Count** меняют на 20,

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

− директиву **equ** м использовать и с выражениями, и со строками.

− идентификаторы, определенные директивой **equ** нельзя переопределять по тексту, а с директивой = м.

**================ МОДУЛЬ 2 =============================**

**5.1. РЕЖ-Ы АДРЕСАЦИИ.**

# **5.1.1. Общие сведения:** 4 реж адресац (рег-ый, непоср-ый, прямой, косвен)**.**

Операнды (команд) м б в 1) памяти (в голове), 2) Рег-ах (4 шт.), 3) портах вв-выв (где корабли). Реж адресации определяет где лежит операнд и как вычислять его физический адрес.

4 реж адресации: 1) Рег-овый (**mov ax, bx**), 2) непосредственный (**mov al, 5**), 3) прямой (**mov [20h], bl; mov ax, ar1** ), и 4) косвенный (8 видов – опред-ет адрес ячейки памяти косвенно). Его определяет **транслятор** по синтаксису задания этой команды.

1) Косвенная Рег-овая адресация: **mov si, offset ar1 … mov al, [si]**

2, 3) При базовой / индексной адресации со смещением (base🡺, ind🡺)

**mov bx, 5;**  **mov al, mas[bx]**

**mov bx, offset mas; mov al, 5+[bx]==[bx+5]==5[bx]**.

4) Базово-индексная **(b-ind)**: **mov bx, offset mas**; **mov si, 5; mov al, [bx][si]**=6

5) **b-ind** адресация со смещением (**b-ind🡺**):

**mov bx, 1; mov si, 3**; **mov al, mas[bx][ si]**

# **5.1.2. 4 Режима адресации.**

1) Рег-овая адресация - операнды лежат в Ре-ах Проц-ра:

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

«+»: нет обращения к памяти (к голове) – сокращается время команды.

«-»: ограниченное кол-во Рег-ов.

2) Непосредственная адресация: операнд лежит в самой Маш Ком. Пр-р:

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

«+»: нет обращения к памяти, «-» – значение непосредств-го операнда задают только 1 раз, его нельзя изменить при выполнении проги.

3) Прямая адресация – когда операнд (команды) - **переменная** лежит в памяти – поле смещения в Маш Ком лежит эффективный адрес ячейки памяти, где хранят запрашиваемые данные (дом, кв). Пр-р:

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

Вычисляет физический адрес операнда. Нет гибкости в определении местоположения операнда при выполнении проги.

4) Косвенная адресация – в Маш Ком нет ни значения операнда, ни его эффективного адреса, а есть инфа для получения (вычисления) эффективного адреса. Т е, Проц-р определяет адрес ячейки памяти косвенно.

8 видов косвенной адресации:

1) Рег-овая, 2) базовая со смещением (base🡺), 3) индексная со смещением (ind🡺), 4) базово-индексная **(b-ind)**, 5) базово-индексная со смещением (**b-ind🡺**), 6) базовая с масштабированным индексом i и смещением, 7) базовая (base), 8) индексная (index).

4.1.) Косвенная Рег-овая адресация = базовая + индексная (base + index). Эффективный адрес операнда - либо в базовых (**b**ase) (**bx, bp**), либо в индексных (**i**ndex) Рег-х (**si, di**). Пр-р:

**ar1 db 5, 6** ; определение перем-ой ar1. 5, 6 –??

…

**mov si, offset ar1** ; в si загрузили смещение перем-ой ar1

**mov al, [si]** ; в al поместили число 5 -1й символ массива

**inc si** ; значение si увеличили на 1

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

Атрибутный оператор **ptr** («пусть трактуем раз») – для явного указания р-ра операнда при косвенной адресации. Его формат:

**тип ptr выражение**.

Явно тип надо указывать, когда непонятно, как трактовать ячейку памяти. Пр-р явного указания размера операнда:

**mov byte ptr [bx], 5** ; в байт памяти по адресу bx, занесли число 5.

4.2, 3) При базовой / индексной адресации со смещением (base🡺, ind🡺) эффективный адрес операнда = ∑ содержимого базового / индексного Рег + смещения в Маш Ком. Пр-р:

**mas db 0, 2, 4, 6, 8, 10** ; объява массива 1 байтных чисел (№циия – с 0)

…

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

**mov al, mas[bx]** ; == mov al, mas[5] - в **al** загрузил эл-тмас с индексом 5 =10

Тот же рез-т б, если в **bx** загрузить не индекс, а адрес массива, а в команде (**mov**) указать смещение:

|  |  |
| --- | --- |
| **mov bx, offset mas** | ;в bx - эффективный адрес (смещение) **I** эл-та массива - 0 |
| **mov al, 5+[bx]** | ; в **al** - **6й** элемент массива (число10) (5+1) |

**mov al, 5+[bx] == mov al, [bx+5] ==** **mov al, 5[bx]**.

4.4) Базово-индексная **(b-ind)** адресация м б лишь при формировании **16** битн эффективного адреса.

В **b-ind** адресации эффективный адрес операнда = ∑ содержимого базового и индексного Рег-ов. А если в формате команды есть смещение, то его прибавляют к этой ∑. Пр-р:

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

…

**mov bx, offset mas** ;в bx - эффективный адрес (смещение) **I** эл-та массива - 0

**mov si, 5** ; число 5 - в Рег si, – индекс в массиве

**mov al, [bx][si]** ; в al загрузили число 10, эффективный адрес = ∑ **bx** + **si**

4.5) Фрагмент проги с **b-ind** адресацией со смещением (**b-ind🡺**):

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

…

**mov bx, 1**  ; в bx загрузили число 1

**mov si, 3** ; в si занесли 3

**mov al, mas[bx][ si]** ; в **al** - 8, эффективный адрес = ∑ bx+si + смещ mas

«+» косвенной адресации: гибкость в определении адреса операнда: м получить доступ к произвольной ячейке памяти. «-» - сложность вычисления эффективного адреса, д/п-ные затраты времени.

# **5.1.3. Правила использования Рег-ов и смещений.**

У Проц-ра есть правила как использ-ть Рег-ы неявно для адресации операндов.

Табл 5.1. Правила комбинирования сегм-тных Рег-ов и Рег-ов со смещением.

|  |  |  |
| --- | --- | --- |
| Рег, содержащий смещение | Сегм-ый Рег по умолчанию | Префикс переопределения сегм-а |

|  |  |  |
| --- | --- | --- |
| bp | ss | ds, es, cs |
| bx | ds | ss, es, cs |
| si, di, не включая строковые команды | ds | ss, es, cs |

**bp – base pointer** register **(**Рег указателя базы)

Пр-р:

**mov cl, [bp]** ; в команде для адресации - сегм-тный Рег **ss**.  
**mov cl, ds:[bp]** ; явное переопределение **ss**.

**Разница** в использовании Рег-ов и смещений при вычислении эффективного адреса - зависит от **разрядности** (16/32) этого адреса.

Часть 2. **5.2. КОМАНДЫ ПЕРЕСЫЛКИ (mov, xchg, bswap).**

# **5.2.1. Команда пересылки mov** (для адресации).

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

**mov приемник, источник**

Виды пересылки:

− в Рег из Рег;

− в память из Рег и наоборот;

− в Рег - Непосред Оп-д (цифра);

− в память - Непосред Оп-д.

Пр-р: **mov ax, bx** ; содержимое bx пересылают в ax (в Рег из Рег).

При использовании команды **mov** **запрещено**:

1) пересылать из памяти в память, 2) загружать Непосред Оп-д в Рег сегм-а (ds, cs, ss), 3) пересылать содержимое 1го сегм-тного Рег в другой (ds🡪ss, cs), 4) использовать Рег **cs** (**code** segment) как приемник.

# **5.2.2. Команда замены - xchg.**

- меняет местами содержимое 2 ячеек 2 Рег / Рег и ЯчП, кроме сегм-ного Рег:

**xchg операнд1, операнд2**

**ar1 db ?**  ; определение перем-ой ar1

…

**xchg ax, bx**  ; содержимое bx ложат в ax, а содерж ax - в bx (меняют мест-и)

**xchg cl, ar1** ; содержимое ячейки ar1 и Рег cl меняют местами.

# **5.2.3. Команда перестановки байт bswap. (**противоположный) (**пере**-ставь!)

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

**bswap операнд**

Операнд – только 32 битный Рег. Команда **bswap** изменяет порядок следования байт на **противоположный**: 1й байт меняется местами с 4, 2й – с 3м. Пр-р:

**.486** ; директива переключения типа Проц-ра

…

**mov eax, 12345678h** ; занесение в Рег eax значения 12 34 56 **78** h

**bswap eax** ; теперь значение в eax = **78** 56 34 12 h

# **5.2.4. Загрузка сегм-тных Рег-ов lds,** les, lfs, lgs, lss.

Загрузку инфы в сегм-тные Рег-ы делают команды **lds** – **load pointer** to **ds** – загрузить указатель в ds, les – в **es**, lfs – в **fs**, lgs – в **gs**, lss – в **ss**. (это сегментные Рег-ы):

lds Рег, память

les (lfs, lgs, lss) Рег, память

Каждая из этих команд делает загрузку сегм-тного Рег и Рег (указанного как операнд) данными из (заданной) ячейки памяти.

При **16** битн реж адресац из памяти берут 4 байта: старшее слово грузят в сегм-тный Рег, младшее – в указанный в команде Рег. Пр-р:

**ar1 dd 1234**5678**h** ; определение перем-ой ar1 (**dd –** double word, 4 байта)

…

**lds ax, ar1**  ; в Рег **ds** (data segment) занесли значение **1234**h (старшее слово),

; в **ax** – 5678h (младшее слово).

При **32** битн реж адресации из памяти берут 6 байт: 2 старших идут в сегм-тный Рег, остальные 4 – в указанный в команде Рег.

**5.2.5. Загрузка эффективного адреса lea.**

Загрузку эффективного адреса выполняет команда **lea** - load effective address (загрузить эффективный адрес) - в Рег идет смещение указанной ячейки памяти:

**lea Рег, память**

**ar1 db ?** ; определение перем-ой ar1

…

**lea ax, ar1** ; в Рег **ax** занесли смещение относительно Рег **ds** пер-ой ar1

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

**lea cx, [bx+40h]** ; в **cx** загрузили смещение ячейки с адресом bx+40h= 1040h

Часть 3. **6.1. КОМАНДЫ РАБОТЫ СО СТЕКОМ.**

**6.1.1. Понятие стека. Адресация данных в стеке push**, **pop.**

Стек – обл Опер Пам-и для временного хранения инфы. LIFO.

Для адресации данных внутри стека - 2 Рег: **ss** – stack segment - адрес начала сегм-а стека и **esp/sp** – stack pointer - значение указателя вершины стека.

|  |  |  |
| --- | --- | --- |
| **Оперативная** память (16 битн реж адресац) | | 64 Кбайта |
| Ss:0000h |  |
| ss:sp | Вершина стека |
|  | Данные |
| ss:0ffffh | Дно стека |

Стек растет (вверх) в сторону уменьшения адресов (как давление в воде), при занесении инфы адрес вершины стека уменьшают. Спец команды для работы со стеком: 1) **push**, 2) **pop** и 3) производные от них. Неявно стек используют при выполнении команд:

**call -** вызов процедуры,

**ret** - возврат из процедуры (return),

**int** - вызов прерывания (interrupt),

**iret** - возврат из подпроги обработчика прерывания (int ret).

Доступ к произвольному элементу стека – с помощью Рег **ebp/bp**.

Пр-р: если поместить в него смещение определенного эл-та, то пара **ss:ebp/bp** б адресовать операнд, лежащий в памяти.

**6.1.2. Стековые команды push**, **pusha, pop, popa.**

1) Для занесения данных **в** стек - команда **push** (push – толкать):

**push операнд**

**push** 1) уменьшает значение в Рег **esp/sp** на **2**, если размер заносимого операнда – 2 байта, на 4 – если 4 байта, 2) операнд кладут на новую вершину стека (в ячейку, на которую указывает **esp/sp**). Операнд - любой 2 байтн Рег / 2 байтн ячейка Опер Пам-и. Пр-р:

|  |  |
| --- | --- |
| **ar1 dw ?**  … | ; определение 2 байтной перем-ой ar1 |
| **push ar1** | ; занесение в стек значения перем-ой ar1 |
| **push ecx** | ; занесение в стек содержимого **ecx** |

2) Для извлечения данных из стека – команда **pop** (**pop** – совать):

**pop операнд**

**pop** заменяют содержимое **операнда** (ЯчП / Рег) словом / двойным словом с вершины стека. Вершина стека адресуется парой **ss:sp** **/ ss:esp** и значение **esp/sp** уменьшают на 2 (если забрали 2 байта) / на 4 (если - 4 байта). Теперь этот Рег указывает на новую вершину стека (лежала ниже достаной). Пр-р:

**ar1 dw ?** ; определение 2 байтной перем-ой ar1

**ar2** **dw ?**  ; определение 2 байтной перем-ой ar2

…

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

|  |  |
| --- | --- |
| **push ar1** | ; занесение в стек значения перем-ой ar1 |
| **mov ax, 12** | ; занесение в Рег ax - 12 |
| **push ax** | ; занесение в стек содержимого Рег ax = 12 |
| **pop ar2** | ; из стека забирают 2 байта - заносят в ar2 = 12 |
| **pop cx** | ; в cx - 0ffaah |

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

**pusha**

**pushaw, pushad**

Она сохраняет Рег-ы в порядке (8 шт.): eax/**ax**, ecx/**cx**, edx/**dx**, ebx/**bx**, esp/**sp**, ebp/**bp**, esi/**si**, edi/**di** (для 16 битн реж адресации 2 байтные Рег-ы, для 32 битн – 4 байтн). Значение **sp** уменьшают на 16 – т к в стек занесли 8 (шт.) 2 байтных элементов и **esp** – на 32 (8 (шт.) 4 байтных эл-ов). По времени выполнения **pusha** медленнее 8 команд **push**, но в памяти **pusha** занимает всего 1 байт.

**pushaw** и **pushad** – модификации команды **pusha**. Команда **pushaw** (pusha **word**) заносит в стек 16 битн Рег-ы, **pushad** (pusha **double** word) – 32 битн. Пр-р: если установлен 32 битный (4 байт) реж адресации, а в стек надо ложить 2 байтные Рег-ы, то команда **pushaw** б преобразована в команду **pusha** с префиксом размера операнда.

**popa** - pop all (извлечь все) – для выполнения обратных к **pusha** действий - забрать из стека **8** элементов (шт.) и класть их в Рег-ы.

**popa, popaw, popad**

Эл-ты заносят в Рег-ы в обратном порядке записи их в стек.

Извлеченное из стека значение **esp/sp** никуда не занесли – его просто отбросили. После выполнения **popa** значение в Рег **esp/sp** увеличили на 16 / на 32 (зависит от реж адресации). Команда **popaw/ popad** – 16/ 32 битн модификация **popa**.

Для сохранения в стеке Рег Фл. – команда **pushf** - push flags (занести Рег Фл.):

**pushf**

**pushf** при 16 битн (2 байт) реж адресации уменьшает значение **sp** на 2 и ложит в новую вершину стека значение Рег **flags**. При 32 битном (4 ба) реж адресации **esp** уменьшают на 4 и в стек ложат **eflags**.

Забрать данные из стека в Рег Фл. – команда **popf** – pop **flags**:

**popf**

**popf** берет 2 байта из стека и ложит их в Рег flags (/4 в **e**flags). Использование команд **pushf** и **popf** – (**!**) способ получить прямой доступ к Регу Фл.

Часть 4. **6.2. КОМАНДЫ ПОРТОВОГО ВВ-ВЫВ.**

# **6.2.1. Понятие порта. Адресация портов.**

Доступ к некоторым Рег-м идет через адресное пространство вв-выв (АПрВ-В). Эти Рег-ы имеют разрядность 8, 16, 32 бита. АПрВ-В физически не зависит от пространства Опер Пам-и и имеет объем в 65536 адресов вв-выв. Поэтому, **порт** вв-выв – 8-, 16- / 32 разрядный аппаратный **Рег**, имеющий определенный адрес в АПрВ-В. Адреса 00f8h - 00ffh не используют

Для управления устройством через порты надо знать: 1) №№ портов и 2) представления данных (записываемых и считываемых).

# **6.2.2. Команды портового вв-выв in, out.**

– чтение (из порта - команда **in, ins**) и запись (команда **out**, **outs**).

**in аккумулятор, порт**

**in** – input from port (ввод из порта): в Рег-**аккумулятор** из **порта** грузят (читают) байт, если аккумулятор – Рег **al,** слово– **ax**, двойное слово – **eax**. 2 способа задать **порт**:

1) как Непосред Оп-д (число), если адрес порта – 0 - 255;

2) адрес порта указать в Рег **dx**, тогда - доступ ко всему АПрВ-В. Пр-р:

**in al, 64h** ; ввести байт в Рег **al** из порта № 64h

**mov dx, 0ffah** ; занести в Рег dx 0ffah

**in al, dx** ; ввести байт в Рег al **из порта № 0ffah**

**out порт, аккумулятор**

**out** output to port (вывод в порт) если **аккумулятор** - **al –** в порт записывают 1 байт, **ax** – 2, **eax** – 4. Адрес порта задают как Непосред Оп-д (0..255) / как содержимое **dx** (0..65 535). Пр-р:

**out 70h, al** ; занесение **в** порт № 70h содержимое **al**

**mov dx, 012ch** ; занесение в **dx** значения 12ch (**dx =** 12ch)

**out dx, al** ; занесение в порт №12ch содержимого **al**

Команда **ins (**получение эл-тов строки из порта вв-выв**)** - input string вводит байт (слово, двойное слово) из порта, адресуемого Рег-ом **dx** и заносит его в ячейку памяти, на которую указывает пара **es:edi/di** (приемник):

**ins приемник, dx**

**insb** (читает 1 байт), **insw (**2 байта**), insd** (4 байта)

Операнд команды **ins** – приемник - только для определения **размера** передаваемых данных. После записи значения в память содержимое Рег edi**/di** меняют: если **DF** **= 1**, то edi/**di** уменьшают, если **DF=0** – увелич-ют. Значение edi/**di** меняют на размер сохраняемого операнда.

Команда **outs** – output string, передает в порт байт, слово, двойное слово памяти, которая адресуется **ds:**esi**/si**. Адрес порта – в **dx**. После записи значения в порт содержимое esi/**si** меняется на размер операнда.

**outs dx, источник**

**outsb, outsw, outsd**

**ins** и **outs** используют с префиксом повторения **rep** – 1 байтный префикс, который выполняет повторение команды ск-ко раз указано в ecx/**cx**. Пр-р:

|  |  |
| --- | --- |
| **str1 db 768 dup (?)**  … …… | ; отвели места для строки из 768 1 байтных эл-в |
| **mov ax, seg str1** | ; в **ax** занесли сегм-тную компоненту  ; адреса перем-ой str1 (только адрес) |
| **mov es, ax** | ; инициализация Рег **es** (д/п рег) |
| **mov di, offset str1** | ; в Рег **di** занесли смещение перем-ой str1 |
| **mov dx, 03c9h** | ; в **dx** занесли № порта |
| **mov cx, 768** | ; счетчик числа **повторов** |

**cld**  ; сброс в 0 DF (**direction** flag) – значение **di** б увеличив

**rep insb** ; ввод инфы из порта в строку **str1 сх** раз

# **PSP** - блок памяти – program segment prefix - префикс программного сегм-та.

# 

# **6.2.3. Пр-р проги, работающей с портами вв-выв.**

Прога, работающая с CMOS – энерго-зависимая память ограниченного объема (256 байт) где хранят системную инфу (текущие дата + время, тип винчестера, размер Опер Пам-и) для правильной **загрузки** и работы компа. При включении питания - проверка CMOS. 2 байта в CMOS - для хранения **контрольной** суммы – суммы значений 1 байтных ячеек памяти - их значения постоянны во время загрузки, а затем эту сумму сравнивают со контрольной суммой, хранимой в CMOS. Если нет совпадения, то с-ма предлагает сконфигурировать настройки компа. Работа с CMOS-памятью - через порты **70h** и **71h**.

Пр-р: прога устанавливает текущее значение минут = 45 и получает значение контрольной суммы, хранимой в CMOS:

**stack1 segment stack** ; сегм-т стека - **stack1**

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

stack1 ends

code segment ; сегм-т кода

assume cs:code, ss:stack1

begin:

|  |  |
| --- | --- |
| mov al, 2 ; во втором байте CMOS хранят текущую минуту  **out 70h**, al ; в порт 70h положила № байта (=2)  mov al, 45h ; задали значение минут – данные представляют в | |
|  | ; упакованном BCD формате (4 бита кодир 1 10чную цифру) |
| **out 71h, al** | ; запись нового значения минут, получение значения ; контрольной суммы CMOS |
| mov al, 2eh  out 70h, al | ; контрольную сумму храню 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 Рег **ah** (МАРАЗМ???) |
| **in al, 71h** | ; теперь в al - младший байт контрольной суммы.  ; Т о, в **ax** - контрольная сумма CMOS |
| mov ah, 4ch | ; корректный выход из проги в DOS |
| int 21h | ; № ф-ции помещают в Рег ah, затем  ; вызываем прерывание DOS – прерывание с № 21h |

**code ends**

**end begin**

Часть 5. **7.1. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ НАД ЦЕЛЫМИ**

**2ЧНЫМИ ЧИСЛАМИ.**

# **7.1.1. Команды сложения: add, adc, inc, xadd.**

Знаковое и б/знаковое числа в ячейках памяти / Рег. После арифмет-ой операции Проц-р устанавливает все флаги, и только по значению флагов определяют правильность рез-та операции. Пр-р: если данные хранят в 1 байтн ячейках: 1101010‬ (=106) +101110‬ (=46). Ответ: 10011000 = +152 и -104

Устанавливаемые Фл.: **CF = 0** (б/зн), **ZF = 0, OF = 1** (зн), **SF = 1.**

При «+» б/знаковых чисел **CF** - признак переполнения: **CF = 0** – нет б/знакового переполнения, **CF = 1** – есть.

При «+» знаковых – **OF** – признак переполнен; **OF = 0** – нет знак переполнен, **OF = 1** – есть.

1) В Проц-ре неск-ко команд сложения. Самая частая – команда **add**:

**add приемник, источник**

В **add** сумма источника + приемника идет в приемник. В соотв с рез-том операции меняют Фл. Рег eflags/flags. Приемник – Рег/ЯчП, источник – Рег, ЯчП, Непос Оп.

2) Команда **adc** – д/учета значения **CF** - add with carry (прибав с переносом):

**adc приемник, источник (adc ah, 0 🡪 ах = al)**

**adc** суммирует операнды и **прибавляет к этой сумме 1**, если к началу сложения **CF=1** (вб/знак - переполнен**)**. Если к началу **CF=0**, то **adc** == **add**.

Т к **adc** использует перенос от предыдущей операции, ее использ для сложения чисел любой разрядности. После **adc** в соотв с рез-том модиф Фл.

Приемник, источник == в **add.**

Пр-р корректного сложения 2 1байтных чисел:

**op1 db ?** ; определение перем-ой op1 (типа «Драсте. Я – переменная **op1**»)

**op2 db ?** ; определение перем-ой op1

…

**xor ah, ah** ; обнуление Рег ah – команда **xor** эффективнее для

; обнуления, чем команда **mov ah, 0**

**mov al, op1**  ; в al - op1 – т к напрямую выполнить «+» “память – память”

; невозм, используют Рег

**add al, op2** ; к содержимому **al** «+» значение op2, рез-т – в al

**adc ah, 0** ; «+» к содержимому **ah** значения 0 (**ah** был обнулен) и Фл. CF;

; Т о, в **ax** - рез-т сложения op1 и op2

**3) inc** – 1 операндная команда сложения - increment (увеличить), для увеличения значения операнда на 1 (как в Паскале).

4) Команда **xadd** - exchange and add (поменять и сложить):

**xadd приемник, источник**

Работает **xadd**: сначала в источник идет значение приемника, затем сумма приемника + источника **до** изменения сохраняется **в приемнике**. Пр-р:

|  |  |
| --- | --- |
| .486 | ; для использования **xadd** нужен Проц-р 80486 и выше |
| mov **al, 8** | ; в al - 8 |
| mov **cl, 2** | ; в cl - 2 |
| **xadd** **al, cl** | ; после **xadd** в al (прием) - 10, **cl = 8**. источнику дали знач прием. |

После **xadd** Фл. меняют в соотв с рез-том: CF, PF, AF, ZF, SF, OF.

# **7.1.2. Команды вычитания: sub, sbb, dec, neg, cmp.**

Устанавлив Фл.: **CF = 0, OF = 0, ZF = 0, SF = 1**.

Рез-т вычитания корректен и для знаков для б/знаков чисел. **CF** **= 1**, если был заем из старшего разряда (Пр-р: - 86 - (- 46)). **OF = 1** – переполнение в старшем бите (Пр-р: 200 - (- 120)).

1) Команда **sub** - subtract (вычитать) вычитает из приемника источник и заносит эту разность в операнд-приемник.:

**sub приемник, источник.**

В соотв с рез-том операции устанавливают (т е они =ы 1??) Фл. CF, PF, AF, ZF, SF, OF. Приемник - Рег/ЯчП, источник – Рег, ЯчП/Непос Оп.

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

**sbb приемник, источник.**

Если **CF** **= 1**, то **sbb** вычитает из приемника источник, а затем из рез-та вычитает 1. Если **CF = 0**, **sbb** == **sub.** Команду **sbb** – для «-» многоразрядных чисел. **sbb** модифицирует Фл: CF, PF, AF, ZF, SF, OF.

Приемник, источник == в **sub**. Пр-р вычитания 2байтных чисел:

**op1 db 12h, 0abh** ; **определение перем-ой op1, младший байт 12h (в dl),**

|  |  |
| --- | --- |
|  | **; старший – 0abh (в dh)** |
| **op2 db 0fah, 45h**  … | ; определение перем-ой op2, младший - 0fah, старш – 45h |
| **mov dl, op1** | ; в **dl** - младший байт **op1** |
| **sub dl, op2** | ; из **dl** вычли знач младшего байта op2:op1-op2 =12h- 0fah |
| **mov dh, op1+1** | ; в dh - старший байт op1 - **0abh** |
| **sbb dh, op2+1** | ; из **dh** вычли старший байт op2 **с учетом заема** (от **0abh**  **;** отняли **45h** и **1**: (0abh+1) – (45h +1)). теперь в **dx** –  ; корректный рез-т вычитан 2 2байтных чисел = А6h. |

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

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

5) Команда сравнения **cmp** вычитает 1 операнд из другого, но никуда не заносит рез-т – только устанавливает Фл.

# **7.1.3. 2 команды умножения \* mul, imul.**

**mul** - multiply (умножать) умножает б/знаковые числа, рез-т –б/знаковый. **imul** – знаковые, рез-т – знаковый («i**so** 9000»).

**mul источник**

источник - только Рег/ЯчП. приемник:

**ax - е**сли источник1 байтн – **al**;

**dx:ax** (старш ч – в **dx**, младш – в **ax**) – если источник2 байтн – **ax**.

**edx:eax** (**edx** – старш ч, **eax** – младшая) – если источник 4 байтн– **eax**.

**mul** модифицирует **CF** и **OF** = 1, когда после **mul** старшая часть рез-та имеет хотя бы 1 бит не 0, иначе (старшая ч = 0) **CF** и **OF = 0**.

Старшая ч рез-та – **ah**, **dx** и **eax** соотв-но для 1-, 2- и 4-байтного умножения.

Пр-р:

**mov al, 5** ; в Рег al занесли 5

**mov cl, 240** ; в Рег cl занесли 240

**mul cl** ; содержимое **al** умножили на знач в cl, рез-т - в **ax**

После **mul** значения Фл. PF, AF, ZF и SF не определены.

2) Команда умножения **знак**овых чисел **imul** - integer multiply (умножить целые знаковые) имеет 3 формата:

**1) imul источник (**1операндн формат**)**

**2) imul приемник, источник (**2 операндный формат команды умнож знак чис)

**3) imul приемник, операнд1, операнд2 (**в 3операндном формате**)**

а) В 1операндном формате команды **imul** источник – байт, слово, двойное слово в Рег/ЯчП. Приемник и рез – как в **mul (al** – **ax, ax** – **dx:ax**, **eax** –**edx:eax)**

После **imul** изменяют Фл. CF, PF, AF, ZF, SF, OF.

**mov al, -5** ; в al занесли -5

**mov cl, 120** ; в cl – 120

**imul** cl ; значения в al \* cl, рез-т - в **ax**: ax = -5 \*120 = -600.

**б)** В 2 операндном формате (**imul приемник, источник**): приемник \* источник, рез-т – в приемник. Приемник - только Рег. Источник – Непоср Оп, ЯчП/Рег.

|  |  |
| --- | --- |
| **op1 dw ?**  …  **.186** | ; определение 2 байтной ячейки памяти |
| **mov cx, 1000** | ; в cx – 1000 (Непоср Оп) |
| **imul cx, -5**  …  **.386** | ; содержимое **cx** умножили на -5, рез-т в **cx** =1000\*-5= -5000. |
| **mov bx, 1000** | ; в bx занесли 1000 |
| **mov op1, -5** | ; **инициализация** перем-ой op1 значением -5 |
| **imul bx, op1** | ; значения Рег bx \* значение перем-ой op1 = в **bx** -5000. |

в) В 3операндном формате (**imul приемник, операнд1, операнд2**) умножает значения операнд1 \* операнд2, рез - в приемник. Приемник - только Рег. Операнд – Рег / ЯчП. Операнд2 – всегда Непосред Оп-д (число).

**.186**

**mov cx, -5** ; в cx занесли -5

**imul ax, cx, -5** ; значение **cx** \* -5, рез-т - в ax = -5 \* -5 = 25.

imul в 2- и 3- операндном форматах меняет CF, PF, AF, ZF, SF, OF. CF и OF = 1, когда рез-т умножения не влазит в приемник, иначе **CF** и **OF** = 0.

# **7.1.4. 2 команды деления: div, idiv.**

**div** – **б/**знаковое и **idiv** – деление знаковое. Рез-т = частное + остаток. divide (делить). **idiv** - integer divide (целые).

**div источник**

**источник** в обеих командах – делитель – только операнд в памяти / Рег. Делимое – неявное (**ax, dx:ax, edx:eax**).

Размер источника определяет Рег-ы (делимое):

если р-р источника – 1 байт, то делимое – **ax** (**ah** – старш ч, **al** – младш), частн – в **al**, остат - в **ah**,

2 байт - **dx:ax** (**dx** – старш, **ax** – младш), частн –**ах**, ост - **dx**,

4 байт - **edx:eax**, частн – **eax**, ост – **edx**.

**idiv** **источник**

В **idiv** знаки остатка и делимого совпадают.

**op1 dw ?**

…

**mov ax, 20** ; делимое

**mov cl, 3** ; **делитель**

**div cl** ; после б/знакового деления в **al** – частное 6, в **ah** – остаток 2

; деление числа **12ccb**h **= 77003d на 10:**

|  |  |
| --- | --- |
| **mov dx, 01h** | ; старшая часть делимого (**dx:ax**) |
| **mov ax, 2ccbh** | ; младшая часть делимого. Итого: **01h 2ccbh = 012ccbh** |
| **mov op1, 10** | ; делитель 10d = 0Ah |
| **div op1** | ; после деления в **ax** – частное 01e14h = 7700d, в **dx** – ост 3 |

… ……..

**mov ax, -25** ; делимое

**mov dl, 7** ; в **dl** – делитель

**idiv dl** ; знаковое деление -25/7 = в **al** – частное -3, в **ah** – остаток -4

Исключительная ситуация – делитель = 0 (х/0 - запрещено) - рез-т вычислений не лезет в приемник. Тогда – прерывание с № 0 – **int 0**, значения частного и остатка не определены.

Часть 6. **7.2. КОМАНДЫ КОРРЕКЦИИ 2НО-10ЧНЫХ ЧИСЕЛ.**

**7.2.1. Общие сведения.**

неупакованный BCD-формат – каждая 10чная цифра - в отдельном байте.

Упакованный («2 в 1ом») BCD-формат – 1 10чная цифра в каждой тетраде.

В Проц-ре нет команд арифмет-их операций над 2-10чными числами, а есть отдельно арифметическая команда над 2чными числами и отдельно корректируется рез-т / исходные операнды.

**7.2.2. Коррекция арифметики неупакованных BCD-чисел – ааа, aas, aam, aad.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Коррекция неупакованного | идет за командой | 1й опе-ранд | Процедура преобразования |
| aaa | BCD-сложения (рез > 9: **CF = 1,**  (< = 9: **CF = 0**) | add / adc | Al | преобраз младшие 4 бита в неупаков-ое BCD-ч, в старшую тетраду al идет 0 |
| aas | BCD-вычитания | sub / sbb | Al | --//-----//-----//----- |
| aam | BCD-умножения | Mul |  | преобразует рез-т в неуп BCD-ч путем деления содерж **al** на 10, частное - в **ah**, остаток – в **al**. |
| aad | BCD-деления | до коман div |  | преобраз неупак-ое 2-10чное делимое в **ax** в 2чн чис: **ah**\*10+ **al** |

1) **aaa** - **ASCII adjust** for add (ASCII коррекция для сложения):

**aaa**

Если рез-т сложения > 9, то значение в Рег ah + 1 и **AF** и **CF = 1**. Иначе (< = 9) **AF и CF = 0**, а Рег **ah** не меняют. Значения остальных Фл. не определены. Пр-р: **08h +** **09h =** al =**1**7, **aaa: al** – 7, в **ah** – **1 –** неупаков 2-10чное число

**xor ah, ah** ; обнуление содержимого ah – 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** ; 2чное «+» al и bl, после add в al =**1**7

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

; **al** – 7, в **ah** – **1**; т о, в **ax** б корректное неупаков-ое 2-10чное число.

2) **aas** - ASCII adjust for **subtract** (ASCII коррекция для **вычитания**):

**aas**

Если рез-т > 9 (уменьшаемое < вычитаемого), то значение в Рег **ah** уменьшают на 1 и **AF, CF = 1**. Иначе (< = 9) **AF, CF = 0**, а Рег **ah** не меняют. Пр-р: **08 – 09 = 9**

**mov al, 08** ; в al – **уменьшаемое**

**mov bl, 09** ; в bl – **вычитаемое**

**sub al, bl** ; из al вычли значение bl: 8 – 9= 9

**aas**  ; коррекция неупакованного BCD-вычитания, в рез-те в **al** – 9 (с

; учетом **заема** 1 8-9 = 9), значение в **ah** уменьшится на 1, CF=1.

3) Команда **aam** - adjust for multiply (коррекция для **умножения**):

**aam**

Т к макс-ое значение рез-та = 9×9=81, то он полностью влазит в Рег **al**. Пр-р:

**mov al, 07h** ; в al 7

**mov bl, 09h** ; в bl 9

|  |  |
| --- | --- |
| **mul bl** | ; al \* bl = в **ax** (3f h = 63) |
| **aam** | ; коррекция неупаков-го BCD-умножения: в **ah** – 06h, **al** – 03h. |

4) **aad** - ASCII adjust for **divide** (ASCII коррекция для **деления**):

**aad**

**aad** преобразует неупакованное 2но-10чное делимое в **ax** в 2чное число (содержимое **ah** умножают на 10 и это произведение прибавляют к **al**, и после значение в **ah** обнуляют). После 1 байтного деления (число в **ax** делят на 1 байтное число) в **al** б верное 2чное значение. Пр-р:

|  |  |
| --- | --- |
| **mov ax, 0702h** | ; в ax - 72 в неупакованном BCD формате (делимое) |
| **mov bl, 08h** | ; в bl – делитель 8 |
| **aad** | ; до операц деления. теперь в **ax** – 0702h = 07\*10+02 **=72** |
| **div bl** | ; **ax /** **bl = 72 / 8**, рез-т - частное = 9 в **al**, остаток **0** – в **ah** |

**7.2.3. Команды коррекции упакованных BCD-чисел daa, das.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Коррекция | Рез-т – в Рег | Рез-т | Корректирует рез-т 1байтного |
| **daa** | сложения | **al** | **> 99 –** усечен | сложен 2 упакован BCD-чисел |
| **das** | вычитания | **al** | уменьш < выч | вычитания 2 упаков BCD-чисел |

Рез-т упаков-ых BCD-чисел коррект только после **сложения** и **вычитания**. 1) **daa** - decimal adjust for add (10чная коррекция для **сложения**):

**daa**

После **daa** в al - корректное упакованное **2но-10чное** число. Если рез-т сложения **> 99**, то после **daa** **CF** **= 1**, а в **al** - усеченный (без старшей цифры) рез-т. Пр-р: **72h + 43h =** al = 0b5h=181d, **daa:**  **al = 51h(=81)**

**mov al, 72h** ; в al - 72 в упакованном BCD формате

**mov dl, 43h** ; в dl - 43 в упакованном BCD формате

**add al, dl** ; 2чное «+» 2 упакованных 2чно-10чных чисел; al = 0b5h=181d

**daa**  ; в **al** = 15 в упаков-ом BCD формате, Фл. CF = 1

Моё: рез-т сложения > 99, то в al – усеченный (без старшей цифры) рез-т. + в калькуляторе **81** в НЕХ = **51**, т е в **al = 51h**, а не 15)).

2) **das –** decimal adjust for **subtract** (10чная коррекция для вычитания):

**das**

Если уменьшаемое < вычитаемого, то CF = 1.

**mov al, 72h** ; в al 72 в упаков-м BCD-формате

**mov dl, 23h** ; в dl 23 в упаков-м BCD-формате

**sub al, dl** ; al – dl = **72h** – **23h** = 4Fh

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

…

**mov al, 10h** ; в al – упаков-ое 2но-10чное 10

**mov dl, 25h** ; в dl – упаков-ое 2но-10чное 25

**sub al, dl** ; «-»: 10-25= -15, в al – 0ebh = 0fbh = -21

**das**  ; коррекция упакованного BCD-вычитания, теперь в al – 85 в

; упакованном 2чно-10чном формате (с учетом **заема** 110-25 = 85); CF=1.

Часть 7. **7.3. КОМАНДЫ ПРЕОБРАЗОВАНИЯ ТИПОВ.**

**7.3.1. Команды расширения знакового бита (с минусом, 1) cbw, cwd, cwde, cdq.** (из **al в ax**)

– увеличение размера операнда с учетом знака. Т е, если в старшем бите операнда **1**, то в расширяемую часть заносят также единицы (т к числа хранят в д/п коде), иначе – **00**.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | | из | В | Из 🡪 в | |
| **Cbw** | convert **byte** to word | | 1 байтное знаков число в **al** | 2 байтное знаковое число в **ax**. | **al 🡪 ax** | |
| **Cwd** | convert **word** to dword | | Знаков слово в **ax** | знаковое слово в **dx:ax** (старш ч – **dx**) | **Ax 🡪 dx:ax** | |
| **Cwde** | convert word to dword **extended** | | знаков слово в **ax** | двойное слово в **eax** | **ax 🡪 eax** | |
| **Cdq** | convert dword to **quadword** | | двойн слово - знаков 32 битное число в **eax** | учетверенное слово - знаковое 64 битное число в **edx:eax** (**edx** – старшая ч) | **eax 🡪 edx:eax** | |
| **xor ah, ah** | | обнуление расширяемой час д/беззнак-го преобразов-ия | | | | **al 🡪 ax** |

1) **cbw** – convert **byte** to word (преобразовать байт в слово), заполняет **ah** значениями старшего (знакового) бита **al**:

**cbw**

**mov al, -30**  ; в al занесли -30 = 0 e2 h

**cbw** ; в **ax** б число -30 = 0 ff e2 h

2) **cwd** - convert **word** to dword (преобразовать слово в двойное слово), во всех битах **dx –** значение старшего (знакового) бита **ax**:

**cwd**

3) **cwde -** convert word to dword extended (преобразовать слово в двойное слово **расширенное**):

**cwde**

Машинный код **cwde** == коду **cbw**.

4) **cdq -** convert dword to quadword (преобразовать двойное слово (2) в **учетверенное слово** (4)):

**cdq**

Маш Ком **cdq ==** **cwd**, поэтому при трансляции учитывают реж адресации сегм-а.

Ни эта 1команда (cbw, cwd, cwde, cdq) не изменяет Рег Фл.

5) Преобразование типов **б/**знаковых чисел – обнулить (0000) расширяемую часть. Пр-р:

**xor ah, ah** ; обнуление расширяемой части д/б/знаков преобразован **al** в **ax**.

**7.3.2. Команды пересылки с- (movsx) и без- (movzx) учета знакового бита.**

**Преобразование типа** – увеличение размера операнда: с учетом знака - **movsx** и без учета знака - **movzx**. Сост Фл. – Не влияют

1) **movsx** - move with **sign** extension (переместить **с** расширением знака):

**movsx** **приемник (**16, 32 бит, Рег**), источник (**8, 16 бит, ЯчП/Р**)**.

|  |  |
| --- | --- |
| **op1 db ?** | ; определение 1 байтной перем-ой op1 |
| **mov op1, -10** | ; в op1 -10 |
| **movsx** **eax, op1** | ; в **eax** занесли знаков**о** расширенное значение перем op1;  ; после **movsx** в **eax** – 0 ff ff ff f6 h = -10 |

2) **movzx** - move with zero extension (переместить с расширением 0) б/знак в старшую часть не заносят:

**movzx приемник (**16, 32 бит, Рег**), источник (**8, 16 бит, ЯчП/Р**)**.

**movzx** расширяет значение источника до размера приемника: заносит в старшую часть **00**.

**op1 db ?** ; определение 1 байтной перем-ой op1

…

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

**movzx eax, op1** ; после выполнения **movzx** в Рег **eax** число 0f6h

Часть 8. **7.4. ОПЕРАЦИИ НАД ОДНИМ ОПЕРАНДОМ + КОМАНДЫ**

**СРАВНЕНИЯ.**

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

|  |  |  |  |
| --- | --- | --- | --- |
| **inc** | increment (увел-ие) | **приемник –**  ЯчП / Рег | **Ув**еличив значения операнда на 1 |
| **dec** | decrement (ум-ие) | **Ум**еньшает значения операнда на 1 |
| **neg** | negate (отрицать) | Меняет знак операнда на противоп |

**inc (dec, neg) приемник**

**neg** вычитает значение приемника из 0 и рез-т записывает обратно в приемник. **CF = 0** – если операнд = 0, иначе **CF = 1**.

Пр-р: для байта – число -128, для слова – число -32768.

**7.4.2. Команды сравнения cmp, cmpxchg, cmpxchg8b.**

1) Команда **cmp** - compare (сравнивать) сравнивает знач-ия 2 операндов м/у собой:

**cmp приемник, источник**

**cmp** вычитает из приемника источник (sub), но рез-т никуда не несет, а модифицирует все арифметические Фл**.** (CF, PF, AF, ZF, SF, OF). По ним определяют рез-т сравнения, поэтому после команды **cmp** используют 1 из команд условного перехода. Пр-р:

**cmp cl, 17** ; сравнен значения в **cl** с числом **17**. Если **=**ы, то при «-» **ZF = 1**

**jz l1** ; перейти на метку l1, если (ZF = 1) условие выполняется

…

**l1**: ; метка, на которую передается управление.

Пр-р: сравнивают 2 знаковых числа; выяснить, что первый операнд > второго (op1 > op2):

**cmp op1, op2**

Чтобы определить какое число >, надо 3 Фл.: OF, ZF и SF. ZF: чтобы op1 был строго >op2, рез вычитан д б не= 0, т е **ZF** = 0, **ZF**=1 – рез-т = 0: операнды =ы

Если **SF** = 0, то рез-т вычитания – «+» число. Если из большего числа (**op1**) вычесть меньшее (**op2**) и рез-т – «+й», то op1 и op2 либо >, либо < 0 одновременно. Значит, OF всегда = 0. Пр-р:

op1 = -5, op2 = -7: -5 - (-7) = 2 (SF = 0, OF = 0).

op1 = 15, op2 = 9: 15 - 9 = 6 (SF=0, OF=0).

**OF = 1**: Пр-р: **op1 = -5, op2 = 125.** Значит, сочетание SF = 0, OF = 1 **не** соответствует, что первое число > второго.

Если **SF = 1**, то после вычитания число б «-»ое. 2 комбинации Фл. для истинности условия **op1 > op2**: 1) OF, ZF, SF=0 (ВСЕ 0) и 2) OF, SF=1, ZF=0.

2) Команда замены по рез-ту сравнения **cmpxchg** - compare & exchange (сравнить и заменить):

**cmpxchg приемник, источник**

**cmpxchg** сравнивает значение приемника со значением аккумулятора (**al, ax, eax**). Если приемник **=** аккумулятору (**ZF = 1**), то в приемник (Рег, ЯчП) идет источник (Рег). Иначе (**ZF = 0**) – в **al/ax/eax** идет приемник (аккумулятор забирает душу приемника).

**op1 db ?** ; определение 1 байтной перем-ой op1

…

**mov al, 0ffh** ; в аккумуляторе 255

mov **op1, 10** ; в op1 приемнике 10

**mov bl, 2** ; в **bl** (источник) 2 (чтобы если «-» = 0, то в **op1** б 2)

**cmpxchg** **op1, bl** ; сравнили **op1** и **al (bl** на стреме**)**; т к **ZF = 0** (10 не = 255),

; то в **al** загрузили содержимое приемника (**op1 = 10**),

; после cmpxchg в **al** б **10** (**bl** оказался не нужен - 3й лишний)

…

mov **al**, 5 ; занесли в аккумулятор 5

mov **cl**, 5 ; занесли в приемник 5

mov bl, 2 ; занесли в источник 2

cmpxchg **cl, bl** ; **cl** сравнив с **al**. Т к **al** - **cl**=0, в приемник **cl** идет ист **bl:** cl=2

Часть 9. **8.1. КОМАНДЫ РАБОТЫ С РЕГИСТРОМ ФЛАГОВ**

**8.1.1. Команды установки (1) и сброса (0) отдельных Фл.**

Выполнение команд: арифметических, сравнения и сдвигов, строковых – связано с установкой / получением определенных Фл. из Рег Фл. Хотя Рег Фл. программно недоступен, м установить и сбросить в нем отдельные биты.

|  |  |  |  |
| --- | --- | --- | --- |
| Команда | Что знач | Перевод | значение |
| **cld** | clear direction flag | очистить Фл. направления | DF = 0 |
| **cli** | clear interrupt enable flag | очистить Фл. разрешения прерываний | IF = 0 |

После **cli** Проц-р не реагирует ни на какие внешние (аппаратные) прерывания, кроме сигналов об отключении питания.

После **sti** Проц-р м реагировать на внешние прерывания.

**8.1.2. Сохранение и загрузка Рег Фл.** pushf и popf, sahf и lahf.

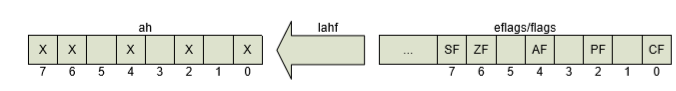
4 команды для работы с Рег-ом Фл. на уровне байт:

1, 2) Команды **pushf** (заносят) и **popf (**извлекают**)** в стеке **весь** Рег Фл.

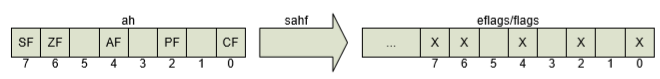
3, 4) Команды **lahf** и **sahf** дают доступ только к младшему байту Рег-а Фл.

3) Команда **lahf** - load **ah** into flags - загружает младший байт Рег **flags** в **ah**:

**lahf**



4) Команда **sahf** – store **ah** into flags - копирует биты 7, 6, 4, 2, 0 из Рег **ah** в младший байт Рег Фл. Т о, заменяют предыдущие значения SF, ZF, AF, PF, CF: **sahf**



Часть 10. **8.2. КОМАНДЫ УСТАНОВКИ БАЙТА ПО УСЛОВИЮ.**

**8.2.1. Команды установки байта по условию setcc, setge, setg,** **setnz**.

Команда **setcc** - set byte on cc (установить байт в завис-ти от условия cc):

**setcc приемник** (**=1** –истина, **=0** - ложь) (**setg bl**)

**setcc** – обобщенное задание данных команд. Вместо окончания **cc** подставляют проверяемое условие. Пр-р:

**setge** (**>=**), **setg** (**>**) / **setnz** (**ZF = 0**). Команда установки байта по условию проверяет свое условие, и если оно истинно, то заносит в приемник **1**, иначе (ложь) – **0**. Приемник – только 8 битн Рег/8 битн ЯчП. Команда не влияет на состояние Рег Фл.

|  |  |  |
| --- | --- | --- |
| **mov ax, -500** | ; в ax -500 |  |
| **mov cx, 300** | ; в cx 300 |  |
| **cmp ax, cx** | ; сравнили ax и cx: ax **g**rater cx ? – no: false, = 0. |  |
| **setg bl** | ; т к -500 <= 300, то **bl** = **0** (setg = 1, если >) | |

Часть 11. **8.3. СПЕЦ КОМАНДЫ nop, hlt.**

**8.3.1. Команда пустой операции nop.**

Для пустой операции – команда **nop (== no)** - no operation (нет операции):

**nop**

**nop** - команда холостого хода, но увеличивает счетчик адреса (значен **eip**/**ip**). Размер – 1 байт; применяют при резервировании места (в ОПере) в памяти если неизвестно, какой б размер формируемой команды, / для выравнивания в памяти на какую-то границу.

Синоним команды **nop** – команда **xchg** **ax, ax**: их машинные коды ==.

**8.3.2. Команда останова Проц-ра hlt.**

**hlt**

Выполнение **hlt**- halt (остановка, стоп!) приводит к прекращению выполнения всех команд и Проц-р переходит в состояние останова. Но прерывания продолжают восприниматься. Тогда 1) управление передается обработчику прерываний, идет подпрога обработчика. 2) После выхода из этой подпроги 3) управление переходит на команду за **hlt**, и 4) Проц-р возобновляет нормальную работу.

**hlt** – последняя в последовательности, которая завершает работу с-мы.

Часть 12. **9.1. ЛОГИЧЕСКИЕ КОМАНДЫ.**

**9.1.1. Общие сведения: команды и, или, исключающее или, не.**

Логические команды (4) – д/логических операций “истина” (1) и “ложь” (0):

− логическое умножение (и); − логическое «+» (/);

− логическое исключающее / («+» по модулю 2);

− логическое отрицание (не).

**9.1.2. Логические команды (команды логических операций) and, test,** **or, xor, not.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| бит a | бит b | a **and** b | a **or** b | a **xor** b | Бит a | **not** a |
| **0** | **0** | 0 | 0 | 0 | **0** | 1 |
| **0** | **1** | 0 | **1** | **1** | **1** | 0 |
| **1** | **0** | 0 | **1** | **1** |  |  |
| **1** | **1** | **1** | **1** | 0 |  |  |

У всех операций приемник – ЯчП/ Рег, источник – ЯчП, Рег, Непосред Оп-д. У всех операций результат идет в приемник.

**1)** Команда **and** (и\*) – побитое (поразрядное) логическое умножение\*:

**and приемник, источник**

2) Еще одна команда логического умножения – **test** (проверить):

**test приемник, источник**

**test** делает поразрядное логическое умножение, но рез-т никуда не заносит – только модифицирует Фл: CF и OF всегда = 0, PF, ZF и SF = в соотв с рез-ом.

**test bl, 01 00 00 00b** ; побитое логическое умножение bl и 64, если в **bl** 6й

**jnz l1** ; бит =**1** (№ – с 0), то б переход на метку **l1** (т к ZF б =0)

**…**

**l1:**

3) Команда **or** - побитое (поразрядное) логическое «+» значений приемника и источника:

**or приемник, источник**

4) Команда **xor** - exclusive or (исключающее /) - побитое логическое исключающее «или» над знач-ми приемника и источника:

**xor приемник, источник.**

**xor** устанавливает в 1 бит рез-т, если только 1 из операндов установлен в 1 бит. Команду **xor** м использовать для обнуления значения операнда.

|  |  |
| --- | --- |
| **xor cx, cx** | ; самое быстрое обнуление значения, быстрее **and cx, 0** |

5) Команда **not** - логическое отрицание, инвертирует все биты приемника:

**not приемник (10000111b** после **not** в al б 01111000b**)**

Часть 13. **9.2. КОМАНДЫ СДВИГОВ.**

**9.2.1. Общие сведения sal**, **shl, shr, shld, shrd, rol, ror, rcl, rcr.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Равносильно | Сдвиг | Выдвигаемые биты | пихают |
| sal == shl | Умножению на 2 знаковых чисел | Арифмет Влево – в стор старш | Теряют старшие биты, последн – в CF | Справа 00 – в старш бит |
| Sar | делению на 2 в степени счетчик | Арифмет вправо | Теряют бит младшие, последн – в CF | Слева знак операнда |
| shl == sal | Умножению б/знаковых на числа, кратные 2 | Логически влево |  | Справа 00 – в старш бит |
| shr | делению б/знак числа на 2 в степе счетчика | Логически вправо | младшие теряют, но последний выдвинут бит - в CF | Слева 00 |

У всех команд приемник - Яч П/Рег, счетчик - сl – 5 бит/Непоср Операнд.

**Команда сдвига** - сдвиг влево / вправо (перемещение) всех бит операнда на некоторое число бит; это число - **счетчик** сдвигов.

3 вида команд сдвигов:

1. арифметические сдвиги **sal** == **shl**: для деления и умножения\* знаковых чисел на числа, кратные 2;

2) **логи**ческие сдвиги **shl, shr, shld, shrd**: в логических преобразованиях и б/знаковых операциях деления и умножения на числа, кратные 2; **не** учитывают знаковый бит числа;

3) **цикл**ические сдвиги **rol, ror, rcl, rcr**: выдвигаемые биты не теряют (как в арифмет-их логич-их), а **за**двигают с другой стороны.

# **9.2.2. Арифметические сдвиги sal, sar** (1) – деление, умножение\*.

Команда **sal** - shift arithmetic **left** (сдвинуть арифметически влево) реализует:

**sal приемник, счетчик shl приемник, счетчик**

**CF приемник**  CF приемник

**000**

0

ст. мл. ст. мл.

влево (в сторону старших бит) на кол-во бит счетчика. Счетчик: 1) значение Рег **cl**, диапазон сдвига определяют **всеми 8** битами этого Рег; 2) Непосред Оп-д, значение которого =1 (сдвиг на 1 бит). Значение Рег **cl** - 5 бит этого Рег.

После выполнения **sal** изменяют Фл. Пр-р:

|  |  |
| --- | --- |
| **mov ax, 00000011b** | ; в ax – 3 |
| **mov cl, 3** | ; счетчик сдвигов |
| **sal ax, cl**  **…** | ; арифметический сдвиг влево на кол-во бит в cl, после  ; выполнения sal в ax – число **11 000 b** = 24 =3\*23 |
| **mov ax, 00000011b** | ; в ax – число 3 |
| **sal ax, 3**  **…** | ; арифметический сдвиг влево содержимого **ax** на 3 бита |
| **mov eax, 1** | ; занесение в eax 1 |
| **mov cl, 33** | ; в cl занесли 33 = **1 00001b** – счетчик **sal** сдвигов |
| **sal eax, cl** | ; арифметический сдвиг влево на 1 бит (т к sal использует |

; только младшие 5 бит **cl**), после выполнения

; в eax – 2 = 000010b

Сдвиг числа на **1** позицию (1 бит) влево == **умножению на 2**. Если сдвиг >1, число \* на 2 в степени, = содержимому счетчика сдвигов. Пр-р: сдвиг влево на 4 бита == умножению на **24 = \*16.**

2) Команда **sar** - shift arithmetic **right** (сдвинуть арифметически **вправо**):

**sar приемник, счетчик**

**приемник** **CF**

ст.- знак операнда мл.

С левой стороны вдвигают знак операнда (**дублируют старший бит**). Т о, положительное знаковое число остаются полож-ым, отрицательное – отриц-ым. После **sar** изменяют Фл.

Пр-р: сдвиг вправо на 3 бита == делению на **23 = 8**. **sar** округляет все положительные числа к **0**, а отрицательные от **0** (команда **idiv** округляет все числа к 0). Пр-р:

|  |  |
| --- | --- |
| **mov al, -1** | ; делимое = -1 |
| **sar al, 1**  **…** | ; в al – -1 |
| **mov ax, -1** | ; делимое = -1 |
| **mov cl, 2** | ; делитель = 2 |
| **idiv cl** | ; в **al** – 0: ax / cl = -1/2 = 0 |

Умножение и деление с помощью сдвигов намного эффективнее, чем командами **mul, imul, div, idiv**.

# **9.2.3. Логические сдвиги shl, shr, shld, shrd.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Приемник | счетчик | Сдвиг | Выдвиг-ые биты | пихают |
| shld | опера1 – 16, 32 битн Рег/ ЯчП;  опера2 –16, 32 битн Рег | знач сl / Непоср операнд | Повышен точности двойной влево | из **операнд1** теряют, но последн - в CF | в **опера1** биты из **операнд2,** псевдо-сдвиг влево **операнд2** |
| shrd | Повыш точности двойной вправо |

1) Команда **shl** - shift logical left (сдвинуть логически влево):

**shl приемник, счетчик**

2) Команда **shr** - shift logical **right** (сдвинуть логически **вправо** на кол-во бит счетчика): **shr приемник, счетчик**

приемник CF

0

ст. мл.

**mov al, 10000101b** ; в al - 10000101b = беззнаковому 133 / знаковому -123

**mov cl, 2** ; счетчик сдвигов

**shr shr, cl**  ; логический сдвиг вправо, после в al б **00100001b** = 33.

2 команды для сдвига повышенной точности:

3) Команда **shld** - shift left **double** (**двойной** сдвиг влево **операнд1** на кол-во бит счетчика):

**shld операнд1, операнд2, счетчик.**

CF **операнд1** **операнд2**

0

ст. мл. ст. мл.

Выталкиваемые из **операнд1** биты теряют, но последний вытолкнутый - в CF. Вначале в **операнд1** вдвигают биты, выдвинутые из **операнд2**, но потом содержимое **операнд2** оставляют как до начала команды – псевдо-сдвиг влево **операнд2**. Команда **shld** двигает влево 32 разрядные (при объединении слов, операнд1 и операнд2 – 2 байтные) 64разрядных (при объединении двойных слов, операнд1 и операнд2 – 4 байтные) значений.

**op1 dw ?** ; определение перем-ой op1

**…**

; сдвиг 32 битного числа влево на 2 бита, его старшая часть – в **op1**, мл - в **bx**:

**mov op1, 1110 000 b** ; старшая часть сдвигаемого числа

**mov bx, 110 0000 000 000 000b** ; младшая часть сдвигаемого числа

mov cl, 2 ; счетчик сдвигов = 2 бита

**shld op1, bx, cl** ; в рез в op1 =1110 000 б **11 1000 011 b**, содержимое **bx** не мен

**shl bx, cl** ; коррекция рез-та – действительный (а не псевдо) сдвиг

; значения в **bx** на 2 **00**, теперь в **bx** – все 00 (15 нулей).

4) Команда **shrd** - shift right double (двойной сдвиг вправо содержимое **операнд1** на кол-во бит счетчика) - сдвиг повышенной точности вправо:

**shrd операнд1, операнд2, счетчик**

**операнд2** **операнд1** CF

0

ст. мл. ст. мл.

**shrd** двигает вправо 32 разрядные (объединение слов, операнд1 и операнд2 2байтные) и 64 разрядные значения (объединение двойных слов, операнд1 и операнд2 4 байтные).

**9.2.4. Циклические сдвиги rol, ror, rcl, rcr.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Сдвиг | | Выдвигаемые биты | пихают | |
| **rol** | циклическ вращать влево | копируют в **CF** | | | Справа (младш бит) вдвигают старший (выдвигаемый) |
| **ror** | циклическ вращать вправо | --//--//---//--- | | | Слева (старш бит) вдвигают младший (выдвигаемый) |
| **rcl** | циклическ вращать через перенос **CF** влево | Справа (в младший бит) пихают бит из **CF**, затем значение выдвигаемого (старшего) бита приемника идет в **CF**. | | | |
| **rcr** | циклическ вращать через перенос **CF** вправо | Слева (в старший бит) вдвигают бит из CF, затем значение выдвигаемого (младшего) бита приемника заносят в CF. | | | |

У всех приемник **-** Рег/ЯчП, счетчик - значение **cl/** Непосред Оп-д = 1.

1) Команда **rol** -rotate left (вращать влево) - циклический сдвиг влево, сдвигает содержимое приемника влево на кол-во бит счетчика:

**rol приемник, счетчик**

CF **приемник**

ст

.

мл

.

**mov al, 100 000 01 b** ; занесение в al 10000001b

**rol al, 1** ; цикл сдв влево на **1** бит, после rol в al – **000 000 11** b.

2) Команда **ror** - rotate right (вращать вправо) - циклический сдвиг вправо, сдвигает содержимое приемника вправо на кол-во бит счетчика:

**ror приемник, счетчик**

3) **rcl** - rotate through carry left (вращать через перенос влево) - циклический сдвиг **через CF влево**, сдвигает содержимое приемника влево на кол-во бит счетчика:

**rcl приемник, счетчик**

CF приемник

ст

.

мл

.

**stc**  ; CF = **1**

**mov bl, 001 110 00 b** ; инициализация bl; **mov** не модифицирует Фл.,

|  |  |  |
| --- | --- | --- |
|  |  | ; поэтому CF не изменяют |
| **mov cl, 3** |  | ; счетчик сдвигов |
| **rcl bl, cl** |  | ; рез-т - в bl **110 001 00 b** и CF = 1 |

4) **rcr** - rotate through carry **right** (вращать через перенос вправо) - для циклического сдвига через CF вправо, двигает содержимое приемника вправо на кол-во бит счетчика:  **rcr приемник, счетчик**

**приемник** CF

0

ст

.

мл

.

Часть 14. **9.3. КОМАНДЫ МАНИПУЛЯЦИИ ОТДЕЛЬНЫМИ**

**БИТАМИ.**

**9.3.1. Команды установленного в 1 бита поиска bsf**, **bsr** – поиск I бита, **= 1.**

|  |  |  |
| --- | --- | --- |
|  | сканирование для поиска I бита операнда | Если его нет (источник = 0), то ZF=1 и приемник не меняют, иначе в приемник поза I бита, |
| **bsf** | вперед, = 1, начиная с младшег (0) | = 1 |
| **bsr** | назад, = 1, начин со старшего | = 1 с младш бит |

У всех приемник (результат - № индекса I бита, =1) – 2, 4- байтн Рег, источник – 2, 4- байтн Рег/ 2, 4- байтн ЯчП.

Команды манипулирования отдельными битами: **bsf** и **bsr** - для поиска первого бита, =1.

1) **bsf** - scan bit forward (сканирование бита вперед) - для поиска вперед I бита операнда, = 1, сканирует биты источника, начиная с младшего (нулевого):

**bsf приемник, источник.**

После **bsf** значение ZF определяется источником, значения CF, PF, AF, SF и OF не определены. Пр-р:

**mov bx, 1000 100 b**  ; инициализация bx

**bsf ax, bx** ; после **bsf** в **ax** – 2 (№ - с 0 с младш байт).

2) Команда **bsr** - scan bit reverse (сканирование бита назад) - для поиска назад первого бита операнда, = 1, сканирует со старшего:

**bsr приемник, источник**

**mov dx, 0100 0000 1000 1001 b** ; инициализация Рег dx

**bsr ax, dx** ; после выполнения bsr в Рег **ax** – 14.

**9.3.2. Команды проверки и установки отдельных бит bt, bts, btr, btc.**

– для проверки заданного бита и, м б, установки его значения.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | Задача 1 | Задача 2 |
| **bt** | test bit |  | По **bt…** значение бита приемника (его поза задана в источнике) 1) заносят во Фл. CF | и 2) затем этот бит приемника |
| **bts** | test bit & set | установить этот бит в **1** | устанавливают **1** |
| **btr** | test bit & reset | сбросить этот бит в **0** | сбрасывают в **0** |
| **btc** | test bit & compliment | инвертировать этот бит | и 2) затем их инвертируют |

У всех задача 1 – проверить значения бита в определенной позе; Приемник – 2, 4- байтн Рег/ 2, 4- байтн ЯчП. Источник – 2, 4- байтн Рег/ Непосред Оп-д. Эти команды модифицируют CF.

1) Команда **bt** - test bit (проверить бит)

**bt приемник, источник**

2) Команда **bts** - test bit and **set** (проверить бит и установить бит в 1):

**bts приемник, источник**

3) Команда **btr** - test bit and **reset** (проверить бит и сбросить бит в 0):

**btr приемник, источник**

4) Команда **btc** - test bit and **compliment** (проверить бит и дополнить бит):

**btc приемник, источник**

**mov dx, 01001b** ; в dx 1001b

**bt dx, 3**  ; № проверяемой позиции - проверка 3го бита, **CF = 1**

**mov bx, 2** ; № проверяемой позиции - проверка 2го бита

**bts dx, bx** ; после проверки **CF = 0** (т к 2й бит = **0**) и установка

; 2го бита в **dx**, т о в **dx** б **– 1101b**.

Часть 15. **10.1. КОМАНДЫ ПЕРЕХОДОВ far, near.**

# **10.1.1. Общие сведения Far**-, **Near**-, **eip/ip, cs.**

Команды переходов передают управление в некоторую **точку** проги без сохранения адреса возврата. Делая переход, эти команды изменяют значение **eip/ip** (использовать 2 байтн **ip** / 4 байтн **eip** – зависит от режима адресации – use16 / use32) и, возм, значения кодового Рег **cs**. Изменение содержимого **cs** определяет вид перехода. Есть переходы:

дальний – **Far**-переход - (м/у сегм-и) межсегм-ая передача управления –изменяет значен **cs** и **eip/ip**

ближний – **Near**-переход (внутри 1 сегм-а) влияет лишь на содержимое **eip/ip**.

Команды переходов никогда не изменяют Рег Фл.

# **10.1.2. Команда безусловного перехода jmp.**

Команда **jmp** - jump (прыгать) (порожденный транслятором) передает управление по некоторому адресу без сохранения адреса возврата (старые значения Рег-ов **eip/ip** и **cs** теряют):

**jmp [модификатор (near** **ptr)] операнд**

**jmp** определяет необязательные поля модификатора и операнда. Если собрать эти команды в группы по виду перехода, то команда **jmp** реализует:

**Внутри-сегментный** переход:

1) **короткий** относительный внутрисегм-тный переход (**относительный** – потому что в формате машинной команды хранят относительное смещение, / расстояние от команды **jmp** до метки);

2) относительный переход в пределах всего сегм-а;

3) **косвенный** внутрисегм-тный переход в пределах сегм-а;

**Меж-сегментный** переход:

4) **прямой** межсегм-тный переход (прямой – потому что в формате машинной команды - полный адрес перехода);

5) **косвенный** межсегм-тный переход.

Подробно:

1) Для **короткого относительного внутрисегм**-тного перехода - команда:

**jmp short метка**

**jmp метка**

Эта команда выполняет короткий (**short**) относительный переход типа **near**, его диапазон – -128..127 байт. Операнд – 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) отнимают значение смещения команды, следующей за **jmp**.

Относительное смещение трактуют как знаковое число – отсюда возм-сть перехода как вперед, так и назад. После выполнения команды вычисленное относительное смещение б прибавлено к содержимому **eip/ip**; и т о, б относительный переход. Пр-р:

|  |  |
| --- | --- |
| **L1:** | ; метка, на которую передается управление |
| **…** | ; расстояние от метки до следующей за jmp команды **<= 128** байт |
| jmp **short L1**  … | ; короткий переход на метку L1 |
| jmp L2 | ; короткий переход на метку L2, переход б коротким т к  ; расстояние от **jmp** до **L2** <= 127 байт; если бы расстояние  ; было вне -128..127, то транслятор сформировал бы команду  ; относительного перехода в пределах всего сегм-а |
| … | ; общий размер кода команд не превышает 127 байт |
| L2: | ; метка, на которую передают управление. |

2) Для **относительного перехода в пределах всего сегм-а** - команда:

**jmp near ptr метка**

**jmp метка**

Эта команда делает относительный переход в любую точку внутри сегм-а. Модификатор **near** **ptr** необязательно указывать, формат машинной команды содержит относительное смещение, которое прибавляют к указателю инструкций **eip/ip**. Пр-р:

**L1:**

**…**  ; расстояние от метки до следующей за jmp команды >128 байт.

**jmp L1** ; относительный переход на метку L1

…

**jmp near ptr L2**

… ; расст-ие от метки **L2** до следующей за jmp команды> 127 байт

**L2 label near**  ; задание ближней метки (типа **near**); такая запись

; аналогична L2:

…

**jmp near ptr L1** ; относительный переход на метку L1.

3) Для **внутрисегм-тного косвенного перехода** – команда:

**jmp источник**.

Источник – Рег / ЯчП, его размер – 2-/4- байта (16-/32- битн реж адресации). При выполнении этой команды б косвенный переход типа **near**. Косвенный – т к адрес перехода лежит не в самой команде, а в Рег / ЯчП.

Фактически, эта команда копирует данные из источника в Рег указателя инструкций **eip/ip**.

Пр-р для 16 битного реж 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

; здесь – идет переход на метку **L1**

Межсегментные:

4) Для **прямого межсегм-тного (/ дальнего) перехода** – команда:

**jmp метка\_типа\_far.**

**jmp far ptr метка.**

Формат Маш Ком содержит полный адрес перехода – адрес начала сегм-а – в нём описана принимающая управление метка, и смещение этой метки относительно начала сегм-а. При выполнении команды значение смещения загрузили в Рег **eip/ip**, значение начала сегм-а – в **cs**. Т о, управление идет на метку, заданную в команде. Если метка объявлена как **имяМетки label far**, то модификатор **far ptr** м не использовать. Пр-р:

|  |  |
| --- | --- |
| **code segment**  **…** |  |
| **jmp L1** | ; дальний переход на метку L1 |
| **EXIT:** | ; метка, на кот из другого сегм-а м передать управление, |
| **…**  **code ends**  **seg1 segment**  **…** | ; 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** |

5) Для **косвенного межсегм-тного перехода** – команда:

**jmp источник.**

- делает косвенный межсегм-тный переход. Источник – ЯчП, размер– 4-/6- байт (16-/32- битн реж адресации).

При 16 битн реж адресац (для нахождения адреса перехода) 2 младших байта источника (младшее слово) грузят в **ip** (смещение), 2 старших (старшее) – в **cs** (№ сегмента).

Пр-р: если в команде по указанному адресу - цепочка байт **12345678h** (старший байт – 12h, младший – 78h), то **cs** б иметь **1234h**, а **ip** – **5678h**.

**farАddr16 dw 0aabb h** ; смещение 0aabbh

|  |  |
| --- | --- |
| **dw 0ccdd h**  … | ; сегм-т (№) |
| jmp dword ptr **farАddr16** | ; межсегм-тный переход, в **ip** – **0aabbh** в **cs** – **0ccddh** |

При 32 битн реж адресации 4 младшие байта копируют в **eip,** 2 старших (старшее слово) источника - в **cs**.

|  |  |  |
| --- | --- | --- |
| **farАddr32 dd 123** |  | ; смещение **123** |
| **dw 0bh**  **…** |  | ; сегм-т (№) |
| **jmp fword farАddr32** |  | ; дальний переход по адресу, определяемому |
|  |  | ; значением **farАddr32** |

**10.1.3. Команды переходов по условию** (условных переходов) **jge, jae, ….**

- делают относительные переходы типа **near**. Они делают передачу управления в завис-ти от установки определенных Фл. в Рег flags.

Их обозначение – **jcc –** jump on cc (перейти, если условие **cc** истинно):

**jcc метка**

м подставить любую из команд, делающих переход в завис-ти от состояния Фл. Диапазон перехода – -128…127 байт. Передача управления – в пределах всего сегм-а. Если проверяемое условие – истина, идет относительный внутрисегм-тный переход по адресу в метке.

2 команды **условного перехода** (**jecxz** / **jcxz**) - выполняют передачу управления в завис-ти от значения в **ecx/cx**.

В завис-ти от сочетания проверяемых Фл Команда **jcc** делает переход, если сделано условие сравнения:

1) 2 знаковых чисел;

2) 2 без-знаковых чисел;

3) переход по состоянию отдельных Фл.

Для установки в Рег **flags** верных значений – команда сравнения **cmp**. Для установки **отдельных** Фл. – команда **test** (аналогична команде **and**, но не меняет значение операнда). **jl** (< =), jnl / jge (> =), jle / jng (< =), jnle / jg (> =).

В табл – команды переходов по рез-ту сравнения 2 знаковых чисел (2 числа в д/п коде):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Команда | Состояние флагов | | | Перейти, если |
| OF | ZF | SF |
| jl / jinge | 0 | - | 1 | < = |
| 1 | - | 0 |
| jnl / jge | 0 | - | 0 | > = |
| 1 | - | 1 |
| jle / jng | 0 | 1 | 1 | < = |
| 1 | 1 | 0 |
| jnle / jg | 0 | 0 | 0 | > = |
| 1 | 0 | 1 |

Буква ‘g’ – “**greater**” (>), ‘l’ – “**less**” (<); ‘e’ - “**equal**”, ‘n’ – “**not**”. Пр-р:

|  |  |
| --- | --- |
| **mov cl, 0feh** | ; в cl – **-2** |
| **cmp cl, 7dh** | ; значение в cl сравнивают с **+125** |
| **jge L2** | ; перейти, если > =; здесь – перехода нет (-2 < +125) |

…

**L2:**

Команды переходов, которые сравнивают 2 без-знаковых числа и анализируют Фл:

|  |  |  |  |
| --- | --- | --- | --- |
| **Команда** | **Сост Фл.** | | **Перейти, если** |
| **CF** | **ZF** |
| **jb / jnae / jc** | 1/1 | -/0 | Ниже (<)/не выше (не>=/ есть перенос |
| **jbe / jna** | 0/1 | 1/1 | Ниже (>)= /не выше (не >) |
| **jnb / jae / jnc** | 0 | - | Не ниже (не <) / выше (>)/ = / нет переноса |
| **jnbe / ja** | 0 | 0 | Не ниже (не <) / = / выше (>) |

‘b’ – “**below**” (ниже), ‘a’ – “**above**” (выше), ‘c’ – “**carry**” (перенос).

|  |  |
| --- | --- |
| **mov cl, 0feh** | ; в cl – **254** |
| **cmp cl, 7dh** | ; значение в cl сравнивают с **+125** (254 >125) |
| **jae L2** | ; беззнаков число в cl 254 > 125, поэтому переход на метку L2 |

**…**

**L2:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Команда | Состояние Фл. | | | | Перейти, если |
| OF | ZF | PF | SF |
| je / jz | – | 1 | – | – | = / 0 |
| j**ne** / j**nz** | – | 0 | – | – | Не = / не 0 == (jump not zero) |

Для проверки отдельных Фл. берут логические команды – они быстрее арифметических. Пр-р: вместо **add al, 0** **– test al, 0ffh.** Пр-р:

**L1:**

**…**

**mov dl, -98** ; занесение в dl -98 = **100 111 10b**

**and dl, 11111111b** ; поразрядное «и» (\*) **111 111 11b**

**js L1** ; перейти, если число в **dl** – отрицат-ое, идет переход на L1.

Формат машинной команды **jcc** имеет относительное смещение, =е разности м/у смещением метки и смещением следующей за **jcc** команды. Для Проц-ра относительное смещение 1 байтное, поэтому делают лишь короткие **near**-переходы. Пр-р:

**cmp cx, 3**

**jg L1** ; перейти, **если больше**

**…**

**L1:**

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

|  |  |
| --- | --- |
| **cmp cx, 3** |  |
| **jle Continue** | ; перейти, если < =. **Continue** – метка |
| **jmp L1 Continue:**  **…**  **L1:** | ; без-условный переход  ; - метка – сделан переход??  ; переход на метку безусловн переход |

Еще 1 вид команд **jcc**: **jc** – их машинный формат имеет 2-/ 4- байтн (для 16-/32- битного) относительные смещения, поэтому такие команды реализуют относительные **near**-переходы в любую точку внутри сегм-а. Их синтаксис == простых команд **jcc**. Транслятор сам порождает нужный код, анализируя смещение до метки.

**L1:**

**…**  ; расстояние от метки до следующей за **jc** командой **<= 128** байт

**jc L1** ; транслятор создает Маш Ком **jc (**jc – **< =**), ее машинный

|  |  |  |
| --- | --- | --- |
| **…**  **L2:** |  | формат имеет 2- /4- байтное относительное смещение. |
| **…** |  | ; расстояние от метки до следующей за **jc** командой > 128 байт |
| **jc L2** |  | ; создают команду условного перехода **jc** |

2 команды перехода по состоянию Рег счетчика **ecx/cx**:

**j**e**cxz метка / jcxz метка.**

**jcxz – jump cx (**e**cx) zero** - выполняет переход, если **cx = 0.**

Команды делают относительные переходы типа **near**, их диапазон -128..127 байт. В отличие от **jcc**, для **jcxz** / **jecxz** диапазон не зависит от реж работы Проц-ра и метка относительно этих команд не м б выше 127 и ниже 128 байт.

Маш Коды **jcxz** (для **cx**) и **jecxz** (для **еcx**) ==, просто в завис-ти от реж адресации в формат машинной команды может включаться байт переопределения размера операнда, определяя, б ли использоваться **cx / ecx**.

|  |  |
| --- | --- |
| **mov cx, 27** | ; счетчик цикла (что-то д б сделано 27 раз) |
| **startloop:** | ; начало цикла (метка) |
| **jcxz endloop** | ; выход из цикла, если в **cx** лежит 0 |
| **sub cx, 3** | ; уменьшение значения в **cx** на 3 |
| **…** | ; тело цикла б выполнено 9 раз |

**jmp short startloop** ; переход на начало цикла

**endloop:** ; конец цикла

Часть 16. **10.2. КОМАНДЫ УПРАВЛЕНИЯ ЦИКЛАМИ loop.**

# **10.2.1. Общие сведения.**

2 способа организации циклов, используя:

1) команды переходов;

2) спец команды управления циклами **loop** (/ команды организации циклов).

Они делают лишь короткие относительные переходы: расстояние от метки до следующей команды за **loop** <= 127 байт: переход вперед (метка ниже команды управления циклами), 128 байт – переход назад (метка описана выше). Рег Фл. не меняет.

# **10.2.2. Команды управления циклами loop, loopw, loopd, loope/loopz, loopew**/**loopzw, looped**/**loopzd, loopne/loopnz.**

Общее для всех **loop: Сначала** идет декремент на 1 содержимого **ecx/cx**, затем анализ **ecx/cx**: если не = 0, то – переход на метку. Иначе (**ecx/cx** **= 0**) - выход из цикла и делают следующую за **loop** команду.

1) Команда **loop** (петля)**: loop метка, loopw метка (**–с **cx**)**, loopd метка** (–**ecx**)

2, 3) **loopw** - loop word и **loopd** - loop dword указывают явно, какой Рег использовать в этой команде. Пр-р: если при 16 битн реж адресации надо использовать **ecx**, то команда **loopd** при трансляции б преобразована в команду **loop** с префиксом размера операнда:

**mov cx, 10**  ; в счетчике цикла **cx** - число повторов = 10

**lp1:**

**…**  ; если в теле цикла значение **cx** не меняют, то цикл б

**loop lp1** ; выполнен 10 раз

**loop** делает те же действия, что и последов-сть **dec ecx/cx, jnz** **метка**, но **loop** на 1 байт короче и быстрее.

4, 5) Команда **loope/loopz** - loop while **equal** / **zero** (цикл, пока =) – расширение цикла **loop**, условие выхода – **ecx/cx = 0 / ZF = 0**:

**loope (== loopz) метка**

**loop –ew/-zw, -ed/-zd метка**

**loopew** / **loopzw** и **loopnew** /**loopnzw** работают с **cx**, **looped**/**loopzd** и **loopned**/ **loopnzd** – с **ecx**; они – модификации команды **loope/loopz**. Пр-р:

; в проге идет многократный вызов процедуры, которая при возникновении

; ошибки ложит код ошибки в Рег **ax**. Если процедура отработала

; нормально, то **ax** = 0:

**…**

|  |  |
| --- | --- |
| **mov cx, 20** | ; счетчик числа повторов |
| **lp1:**  **…** | ; начало цикла |
| **call Мyproc**  **test ax, 0ffff h** | ; вызов процедуры, команда **call** передает управление  ; процедуре Мyproc. При выходе из процедуры  ; управление передают на следующую за **call** команду. |
| **loopz lp1** | ; цикл завершится, когда в **ax** б значение не = 0, /  ; когда счетчик **cx** б = 0. |

6, 7) Команда **loopne/loopnz** - loop while not **zero**/**equal** (цикл, пока не = 0) - модификации **loope/loopz**. Условие выхода – **ecx/cx** = 0 / **ZF** **= 1**.

**loopne (== loopnz) метка**

**loopn-ew/-zw, - ed/-zd метка**

**mov cx, 10** ; счетчик цикла

**lp1:**

**…**

**cmp al, 27**  ; цикл выполнится 10 раз или б прекращен, когда

**loopne lp1** ; значение в **al** станет = 27.

# **10.2.3. Работа с командами управления циклами.**

- организуют циклы и с пост-, и с пред-условием.

1a) Цикл с постусловием (== repeat Pascal) - **вначале метка потом loop**:

**mov cx, 100** ; счетчик числа повторений

**L1:**  ; **вначале метка потом loop**

**…**

**loop L1**

1b) Цикл с предусловием (== if…then /while...do) - вначале **loop**, потом метка:

**mov cx, 10** ; счетчик числа повторений

|  |  |  |
| --- | --- | --- |
| **L1:**  **loop L2** |  | ; метка (оператор цикла)  ; вначале **loop**, потом метка |
| **jmp Exit**  **L2:** |  | ; если cx = 0, то выход из цикла |
| **…** |  | ; тело цикла |
| **jmp L1**  **Exit:** |  | ; возврат на оператор цикла (на команду **loop**) |

2) При пост-условии??: если при первом выполнении **loop** счетчик **cx** = 0, то тело б выполнено 65536 раз, если **ecx** = 0 – 4294967296 раз (=65536\*65536).

**Сounter dw ?**

**…**

**mov cx, Сounter** ; если Сounter = 0, то тело б выполнено 65536 раз

**Lp1:**  ; это – пост-условие

**…**

**loop Lp1**

Почему 65536 раз:вначале декремент **ecx/cx**, и потом проверка **ecx/cx** на = 0. А если (0 – 1) и рез-т поместить в 2 байтную ячейку, то в ней б число **65535**. Поэтому, если число повторов заранее не известно, и берется как значение какой-то перем-ой, то используют **jcxz/jecxz** перед меткой начала тела цикла:

**Сounter dw ?**

**…**

**mov cx, Сounter** ; **cx** - счетчик

**jcxz endLoop ; переход на конец цикла при значении счетчика cx = 0**

**StartLoop:**

**…**

**loop StartLoop**

**endLoop:**

Рег Фл. не меняют. Поэтому если в цикле последняя команда изменяет Фл, то после **loop** применяют условные переходы:

**mov cx, 15**

**StartLoop:**

**…**

**add ax, 5**

**loop StartLoop**

**jp L1**  ; переход на L1, ели Фл. четности PF = 1

**…**

**L1:**

С командами **loop, loope/loopz, loopne/loopnz** м делать **вложенные** циклы. Но т к все эти команды используют **ecx/cx**, его содержимое надо временно сохранять и затем восстанавливать. Лучше – в **стеке** – тогда м б любой уровень вложенности.

|  |  |
| --- | --- |
| **mov cx, 20**  **lp1:** | ; счетчик внешнего цикла |
| **…** | ; \*\*\* внешний цикл |
| **push cx** | ; занесение в стек внешнего цикла (кол-ва повторений) |
| **mov cx, 10**  **lp2:** | ; счетчик внутреннего (**вложенного**) цикла |
| **…**  **loop lp2** | ; \*\* внутренний (**вложенный)** цикл |
| **pop cx** | ; извлечение из стека кол-ва повторений внешнего цикла |

**…**  ; \*\*\* внешний цикл

**loop lp1**

**===================== ЧАСТЬ 3 ===============**

**11.1. КОМАНДЫ ОБРАБОТКИ СТРОК. Цепочки (текст)**

# **11.1.1. Общие сведения movs, cmps, scas, lods, stos, ins, outs.**

Команды обработки строк / цепочечные команды – класс команд Проц-ра, обрабатывают последовательности 1типных элементов, такие последоват-ти называют **строками / цепочками**. Обрабатываемый элемент строки = байт, слово, двойное слово. Эти команды пересылают (**movs**), сравнивают (**cmps**), сканируют (**scas**), загружают (**lods**) и сохраняют (**stos**) строки, вв-выв строки из порта (**ins**) и в порт (**outs**). После них идет авто-инкремент (**DF = 0**) либо авто-декремент (**DF = 1**) содержимого **esi/si** и **edi/di**.

Строка м б источником / приемником.

Строка-источник адресуют парой **ds:esi/si**, в **esi/si** – смещение относительно **ds** текущего элемента строки.

Приемник адресуют парой **es:edi/di**, в **di** - смещение относительно **es** текущего элемента. При обработке строк всегда идет авто-инкремент / авто-декремент индексных Рег-ов, значения в них изменяют на величину, = р-ру эл-та строки в байтах. Т е, если идет обработка строк слов, то **esi/si** и **edi/di** будут изменять на 2. То, в какую сторону будут изменять значения **si**, **di** (индексных Рег-ов), определяет **DF**: если **DF = 0**, то **esi/si** и **edi/di** **ув**еличивают, иначе (**DF = 1**) - **ум**еньшают. Для управления **DF** – 2 команды – **std** (DF = 1 **ум**) - set direction flag и **cld** - clear direction flag (DF = 0 **ув**).

При работе со строками надо вызывать 1 и ту же команду неск-ко раз. Для оптимизации этого процесса – **спец префиксы - повторения:**

# **11.1.2. Префиксы повторения rep, repe/repz, repne/repnz.**

Синтаксис команд с префиксами:

**rep** строковаяКоманда

**repe/repz** строковаяКоманда

**repne/repnz** строковаяКоманда

У всех команд - После каждого повтора декремент **ecx/cx**.

1) Префикс **rep** - repeat (повторять) - повторяет выполнение команды, пока значения **ecx/cx** **не = 0**. значения Фл. не меняют.

2) Префикс **repe/repz** - repeat while **equal** / **zero** (повторять, пока =) – повторение строковой команды, пока **ecx/cx** не = 0 **и** **ZF = 1**; этот Фл. меняют строковыми командами (сравнения и сканирования).

3) Префикс **repne/repnz** - repeat while **not** **equal** / **zero** (повторять, пока не =) повторяет строковую команду, пока **ecx/cx** не = 0 и **ZF = 0**.

Префиксы повторения аналогичны по ф-циям командам организации циклов. Пр-р: рез-т выполнения **rep movsb** аналогичен рез-ту фрагмента:

**jcxz l2**

**l1:**

**movsb ; movs** - move string + **b =** переслать строку байтов

**loop l1**

**l2:**

Но строковая команда с префиксом **rep** быстрее, чем команды внутри цикла **loop** т к выполняется только сама команда, уменьшается счетчик и анализируется условие; не изменяет **eip/ip**, не делает переходов.

При повторении команд **movs, stos, lods** **ZF** не учитывают.

И для команд **movs, stos, lods**, и для **cmps** и **scas** с префиксами схема работы одинаковая кроме пункта **4** – этот только для **cmps** и **scas**: 1) проверка **ecx/cx** на = 0, если да – выход из цикла; 2) выполнение команды (строковой); 3) декремент **ecx/cx**;4) проверка ZF: если **ZF = 0** (для **repe**) / **ZF = 1** (**repne**) - выход из цикла.

**11.1.3. Строковые команды movs. – 3.05.2020, 17.05.2020, 22.05, 29.05**

С командами **cmps** и **scas** используют префиксы **repe/repz** и **repne/repnz** + команды условных переходов.

Операнды **stos, cmps, scas** – для определения размера данных, а операнды **movs и lods** – для определения типа данных, но не для адресации.

1) Для **пересылки** (элемента) строки - команда **movs** - move string:

**movs приемник, источник**

**movsb, movsw, movsd**

**movs** копирует данные, адресуемые парой **ds:esi/si** в область памяти, на младший байт которой указывает пара **es:edi/di**. Транслятор всегда преобразует команду **movs приемник, источник** в эквивалентную без-операндную команду.

Пр-р: если приемник и источник определены как байты (**db**), то транслятор породит команду **movsb**. Её используют вместе с префиксом **rep**. После **movs** Рег Фл. не меняют. Пр-р: копирование 1 строки в другую:

**str1 db** **‘abcdе$’** ; всего 5 элементов

**str2 db 5 dup (?)** ; то, куда скопируют **str1** (в str2)

**…**

**mov si, offset str1** ; смещение строки-источника

**mov di, offset str2** ; смещение строки-приемника

**mov cx, 5** ; пересылают 5 элементов

**cld**  ; сброс в 0 DF – знач индексных Рег **si, di** б ув-ся

**rep movsb** ; копирование строки **str1** в строку **str2**.

2) Для **сравнения** (элементов) строк – команда **cmps** - compare string:

**cmps источник, приемник**

**cmpsb, cmpsw, cmpsd** (сравнение строк по байтам, словам)

**cmps** сравнивает данные, адресуемые парой **ds:esi/si** с данными, на которые указывает **es:edi/di**. Т е, из элемента-источника (адресуемого **ds:esi/si**) вычитаю элемент-приемник (адресуемый **es:edi/di**), но рез-т вычитания никуда не ложат, а по нему устанавливают Фл. в Рег **flags**. Пр-р:

; поэлементное сравнение 2 строк

**str1 dw 5 dup (?) ;** строка-источник

**str2 dw 5 dup (?) ;** строка-приемник

**…**

**mov si, offset str1** ; смещение строки-источника

**mov di, offset str2** ; смещение строки-приемника

|  |  |
| --- | --- |
| **mov cx, 5** | ; сравнивают 5 элементов |
| **cld** | ; сброс в 0 DF – значения индексн Рег-ов б ув-ся |
| **repe cmpsw** | ; поэлементное сравнение слов |
| **jz EqualEl**  **…**  **EqualEl:** | ; переход, если все элементы строк совпадают |

3) Для **сравнения значений аккумулятора с элементом строки (**Сканирование строки**)** - команда **scas** - scan string (сканировать строку):

**scas приемник**

**scasb, scasw, scasd**

Команда сравнивает значение в аккумуляторе **al, ax, eax** (в завис-ти от формата команды) с данными, на которые указывает пара **es:edi/di**. После сравнения ставят все Фл. в соответст с вычитанием из аккумулятора данных по адресу **es:edi/di**, но сами аккумулятор и строку не изменяют. Пр-р: определение позиции **I** эл-та, совпадающего с символом в **al**

**str1 db ‘abc1de2 $’**

**…**

**mov di, offset str1**  ; смещение строки-приемника

**mov cx, 8** ; кол-во элементов строки (пробел тоже считается??)

**cld**  ; сброс в 0 DF = 0

**repne scasb** ; сканирование

**jcxz noFound** ; если в **cx** – 0, то не найдено ни 1 совпадающего эл-та

**sub di, offset str1** ; в **di** – позиция совпавшего элемента

**…**

**noFound:**

4) Для **загрузки элемента из строки в аккумулятор**

– команда **lods** - load string (загрузить строку):

**lods источник**

**lodsb, lodsw, lodsd**

**lods** загружает байт (на который указывает пара **ds:esi/si**) – в **al** (**lodsb**), слово – в **ax** (**lodsw**), двойное слово – в **eax** (**lodsd**).

**lods** не влияет на Рег Фл. **lods** - без префиксов. Пр-р:

; фрагмент, после выполнения которого в **al** положат I несовпадающий

; символ строки str1, сравниваемой с 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** | **; сравнивают 5 элементов** |
| **cld** | ; сброс Фл. направления |
| **repe cmpsb** | ; сравнение строк идет пока элементы =ы и cx не = 0 |
| **jcxz Equal** | ; если нет несовпадающих элементов – переход на  ; метку **Equal** (т е все эл-ты совпадают) |
| **dec si** | ; уменьшение значения в **si** – теперь **ds:si** указывает  ; на I не совпавший элемент |
| **lodsb**  **…**  **Equal:** | ; загрузка этого элемента в **al** |

Команда **lods** == **mov al (ah, ax),** **byte ptr [si]**. Но лучше **lodsb**.

5) Для **записи содержимого аккумулятора (al, ah, ax) в элемент строки** (Сохранение эл-та в строке) – команда **stos** - store string (сохранить строку):

**stos приемник (di)**

**stosb, stosw, stosd**

**stos** записывает содержимое al, ax, eax в байт, слово, двойное слово, который адресуется парой **es:edi/di**.

**stos** используют с префиксом **rep** для заполнения строки определенным значением. Пр-р: инициализация эл-ов массива значением 1 (единицами)

**Mas dd 10 dup (?)**

**…**

**mov di, offset Mas** ; смещение первого элемента массива

|  |  |
| --- | --- |
| **mov cx, 10** | ; кол-во элементов = 10 |
| **mov eax, 1** | ; значения, которыми инициализируют массив |
| **cld** | ; сброс DF = 0 |
| **rep stosd** | ; сохранение элементов строки. |

# **11.1.4. Работа с командами обработки строк es, ds.**

Для адресации строки-приЕмника – **es (extra),** строки-источника – **ds (data)**. Но если обе строки лежат в сегм-те данных **ds**, то **es** и **ds** м инициализировать 1аковыми значениями:

**data segment**

**…**

**data ends**

**code segment**

**…**

**mov ax, data mov ds, ax**

**mov es, ax**

**…**

**code ends**

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 и если **ecx/cx = 0** перед циклом, то тело не выполнят ни разу; а с **loop** – 65536 раз. М не использовать проверку счетчика на = 0 перед выполнением **rep строкКом**:

**len db 10** ; длина строки

**…**

**mov cx, len**

**rep movsb** ; если cx = 0, то ни 1 пересылки не б

Часть 2. **11.2. ТАБЛИЧНОЕ ПРЕОБРАЗОВАНИЕ.**

**11.2.1. Команда табличного преобразования xlat.**

Для выбора эл-та из строки по индексу – команда **xlat** - translate string:

**xlat источник**

**xlatb**

**xlat** заменяет содержимое **al** значением с индексом из табл, на которое указывает пара **es:ebx/bx**. Содержимое **al** считают **индексом** выбираемого элемента. Операнд команды **xlat** не используют, транслятор всегда порождает команду **xlatb**. Рег Фл. не изменяют. Пр-р: РАБОТАЕТ.

; прога перекодирует строку позиций в строку осмысленных символов; Табл

; преобразования – английский алфавит; рез - на экране – строка “assembler”

**data segment** ; сегм-т данных

**Table db** 'abcdefghijklmnopqrstuvwxyz' ; Табл преобразов, из кот выбир знач

**PosMas db 0,18,18,4,12,1,11,4,17** ; массив позиций – строка, кот б преобраз

|  |  |  |
| --- | --- | --- |
| **Str1 db 10 dup (?)**  **data ends** |  | ; результир-ая строка, в кот преобр-ют position\_mas |
| **stack1 segment stack** |  | ; сегм-т стека **stack1** |
| **db 100h dup(?)**  **stack1 ends** |  | ; под стек отводят 256 байт |
| **code segment** |  | ; сегм-т кода |

**assume cs:code, ds:data, ss:stack1**

**begin:**

**mov ax, data** ; занесение в ax начала сегм-а данных

**mov ds, ax** ; инициализация Рег **ds**

**mov es, ax** ; в **es** заносят адрес начала сегм-а данных

**mov bx, offset Table** ; в bx заносят смещение перекодирующей табл

**mov si, offset PosMas** ; 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 DF, индексные Рег будут увеличены | | |
| **lodsb** | ; в **al** заносят значение из строки **PosMas** | | |
| **xlatb** | ; значение в **al** воспринимают как индекс в таблице  ; **Table**, по этому индексу выбирают значение, котор  ; грузят в **al** **(опять в ал??)**. | | |
| **stosb** |  | ; занесение значения **al** в строку **str1** |
| **loop L1** |  | ; цикл повтор 9 раз = длине перекодируемой строки |
| mov byte ptr es:[di],'$' |  | ; символ конца строки - для корректного вывода на  ; экран функцией 9 прерывания 21h (прер-ие DOS) |
| **mov ah, 09h** |  | ; для прерывания 21h № ф-ции заносят в Рег **ah** |
| **sub di, 9** |  | ; значен **di** уменьшают на величину размера массива |
| **mov dx, di** |  | ; в dx заносят смещен **str1** –д/9й ф-ции прер 21 h |
| **int 21h** |  | ; вызов прерывания |
| **mov ah, 4ch** |  | ; ф-ция 4ch прерывания 21h – завершение раб 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.**

**Прерывание** – процесс (инициируемый определенным образом) временно переключает Проц-р выполнять другую прогу с последующим возобновлением выполнения прерванной проги.

**Обработчик прерывания** – прога, которая прерывает выполнение текущей проги и принимает управление.

К генерации (/ вызову) прерывания приводят 2 ситуации:

1) **внешнее** событие по отношению к выполняемой проге, инициируют внешние устройства - вырабатывает определенный сигнал, по которому Проц-р д обслужить устройство (нажатие клавиши, деление на 0 в Проц-ре);

2) **внутреннее** событие выполняемой проги – спец командами.

Всего - 256 прерываний. Первые 32 зарезервированы Проц-ром, остальные **224** – пользовательские для прогера.

У каждого прерывания – свой №, кот соответствует индексу в **таблице** **адресов** (- **Вектора** прерываний) обработчиков прерываний – Табл векторов прерываний - начинается с адреса **00 00 h:00 00 h**. Эта Табл лежит в Опер Пам-и и Проц-р знает адрес ее начала. Каждый её элемент – 4 байта. В старшем слове - адрес начала сегм-а, в младшем – смещение относительно этого сегм-а. Когда вызывают прерывание из этой табл по заданному индексу, то выбирают адрес обработчика и ему передают управление.

Часть 4. **12.2. КОМАНДЫ РАБОТЫ С ПРЕРЫВАНИЯМИ.**

**12.2.1. Вызовы прерывания int, int 3, bound.**

1)Для вызова прерывания с заданным № - команда **int** - interrupt (прервать):

**int число**

**int** сначала заносит в стек значения **eflags/flags, cs, eip/ip**, затем управление передают обработчику прерывания, его № = числу в этой команде 0…255. Занесение в стек Рег Фл. делают, чтобы при возврате из обработчика этот Рег был восстановлен. Значения **cs:eip/ip** - адреса возврата для правильного возврата из проги обработчика. при занесении в стек значения **eip/ip,** этот Рег указывает на следующую за int команду.

Обработчики прерываний имеют много ф-ций, кот зав-ят от передаваемых им параметров.

Пр-р: для установки реж дисплея - прерывание BIOS 10h (обслуживание видеос-мы), причем в **ah** указывают № ф-ции 0 (установка реж дисплея), а в **al** задают сам реж:

**xor ah, ah** ; ф-ция 0 прерывания 10h – установка реж дисплея

**mov al, 3** ; в al задают реж дисплея, 3й реж – текстов: 80 столб, 25 стр

**int 10h** ; вызов прерыван, после вып-ия команды устанавлив реж диспл

2) Вызов 3го прерыван в Проц-ре - спец 1байтн команда (любая др – 2байтн):

**int 3**

Для прерывания **int 3 (**по умолчанию) устанавливают пустой обработчик, вызов **int 3** не приводит ни к каким последствиям, м написать свой обработчик 3го прерывания. Проги отладчики используют прерывание 3 для установки точки **прекращения** проги.

4) Команда вызова прерывания **5** – при выходе значения за рамки диапазона - **bound** (предел) - для работы с массивами:

**bound индекс** (Рег)**, источник** (ЯчП)

**bound** проверяет, находится ли индекс в заданном диапазоне. Значения индекса и источника – знаковые числа. **Источник** задает нижнюю и верхнюю границы диапазона, и если оно вне границ, то генерируют прерывание с № 5. В стек заносят пару **cs:eip/ip**, указывающую на саму команду **bound**, а не на следующую за ней команду (как при выполнении **int)**. Пр-р:

|  |  |
| --- | --- |
| **min\_max dw -6, 9**  **…** от -6 до 9 | ; определение значен источника, младш слово – нижн  ; граница диапазона (-6), старш слово – верхн граница (9) |
| **mov bx, 10** | ; занесение в bx 10 – проверяемый индекс |
| **bound bx, min\_max** | ; т к значение в bx лежит **вне** диапазона, то идет  ; вызов прерывания **5** |

**12.2.2. Корректный выход из обработчика прерывания** (из запоя ☺ - рвота) **iret**.

- команда **iret** - interrupt return:

**iret iretd**

**iret** берет из стека значения **ip**, **cs,** **flags** (1й - **ip**), а **iretd** – **eip**, **ecs**, **eflags**. Занесения в **cs:eip/ip** данных из стека реализует дальний (**far**) переход. Т о, после выполнения **iret/iretd** управление передают команде, на которую указывает пара **cs:eip/ip.**

Маш Код **iret**, **iretd** ==. Пр-р: если при 32 битн реж адресац 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 НИМИ.**

**13.1.1. Описание процедур в Ассемблере proc**, **endp.**

**Процедура** (/ подпрога) – группа логически выделенных команд, которые м многократно вызывать прогой. Вызов процедуры – (на, держи) передача ей управления; вызов требует д/п время на переход и возврат из процедуры. Для описания процедур – 2 директивы – **proc** и **endp**:

**имяПроцедуры proc [тип** – near, far**] [язык] [USES Рег-ы]** ; заголовок проц

… ; команды, составляющие тело проц-р

**имя\_процедуры endp**

Идентификатор **имяПроцедуры** – метка, указывает на I команду процедуры. Модификатор **тип** м б значением **near** и **far**, а создаваемые процедуры б определять как ближние и дальние. Если есть модификатор **USES**, то транслятор создаст последов-сть команд **push** в начале процедуры, т о, в стек будут заносить значения Рег-ов, указанных в **USES**. Перед командой **ret** б создана последов-сть команд **pop**. Если **тип** процедуры не задан, то неявно он выбирается по используемой модели памяти (**tiny**, **small** и **compact** – тип **near**).

Процедура м б в любом месте проги, но после компиляции ее код не отделим от остального кода проги. Когда прога дойдет до первой команды процедуры без ее явного вызова, то приведет к неожиданным последствиям.

Чтобы избежать этого, надо размещать процедуры либо в начале / конце проги (до первой исполняемой команды / после команды, возвращающей управление ОС-ме), либо в теле проги делать обход процедуры с помощью команд перехода. Пр-р: правльн описания процедуры:

**.model small**

**.data**

**message db "Hello!!!$"**

**.code**

**showMessage proc**

**mov dx, offset message**

**mov ah, 9**

**int 21h**  ; вывод строки

**ret**  ; возврат управления вызвавшей проге

**showMessage endp**

**Start:**  ; метка, начало проги

**mov ax, @data**

**mov ds, ax**

**call showMessage** ; вызов процедуры

**mov ax, 4C00h**

**int 21h**  ; выход из проги

**end Start**

**13.1.2. Команды работы с процедурами call, ret, enter, leave.**

Для передачи управления процедуре - любая команда перехода, в том числе и условного. Лучше - команду вызова **call** (звонить):

**call [модификатор] операнд**.

Команда **call** сохраняет в стеке адрес возврата и передает управление по адресу - операнд команды **call**. Операнд:

1) имя процедуры/метки: для ближнего (вну3сегм-тного) перехода в формат Маш Ком записывают относительное смещение, а для дальнего (межсегм-тного) перехода – в Маш Ком полный адрес ячейки, на которую передают управление – сегм-т + смещение;

2) значения ЯчП / Рег - используют как адрес перехода (для ближнего перехода извлеченное значение модифицирует **eip/ip**, для дальнего – **cs** и **eip/ip**).

Адрес **возврата** (заносит в стек команда **call)** – адрес младшего байта команды, следующей за **call** (напр-р, mov). Для ближнего перехода в стек заносят только значение **eip/ip**, для дальнего – сначала **cs**, затем **eip/ip**. Модификатор – **near ptr (**ближний**)** и **far ptr (**дальний**)** вызовы. Если **тип** процедуры указан явно в директ-е **proc**, то модификатор м не использовать.

; ближние переходы:

**call Proc1**

**call near ptr Proc1**

**mov dx, offset Proc1**

**call dx**

**…**

**Proc1 proc near**

**…**  ; тело процедуры

**Proc1 endp**

; дальние переходы

**addr16 dw ?** ; смещение

**addr16 dw ?** ; сегм-т

**…**

**call Proc2**

**call far ptr Proc2**

**mov addr16, offset Proc2**

**mov addr16+2, seg Proc2**

**call dword ptr addr16**

**…**

**Proc2 proc far**

**…** ; тело процедуры

**Proc2 endp**

Для возврата из процедуры - команда **ret** - return:

**ret [число]**

**retn [число] (**возврат из ближней процедуры**),**

**retf [число] (**из дальней**)**

**ret** передает управление по адресу, который лежит на вершине стека. **ret** всегда преобразуется транслятором в **retn /** **retf**. **retn** для 16 битн реж адресац берет из стека 2 байта и загружает их в **ip** (для 32 битн – 4 байта в **eip**). А для

**retf** для 16 битн реж тоже берет из стека 2 байта и загружает их в **ip** (32 битн – 4 байта в **eip**) и + ещё берет из стека 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 параметры, передаваемые процедуре через стек.

При 16 битн реж адресац сегм-а число в команде (**ret??**) определяет байты, которые надо удалить из стека (от **esp/sp** отнимают число), при 32 битн реж – число определяет слова (от **esp/sp** отнимают число**×2**). Пр-р:

|  |  |
| --- | --- |
| **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

**Proc1 endp** ; **call**, здесь – команде **inc bx. retn - near**

**Proc2 proc**

**…**

**retf**  ; дальн возврат следующей за call управлен команде – **dec bx**

**Proc2 endp**

**Proc3 proc**

**…**

**retn 4**  ; удаление из стека переданных проц-ре пар-ров: **sp** минус 4

**Proc3 endp.**

2 спец команды для **создания enter** и **удаления** **leave** локальных (внутренних) переменных процедуры: команда **enter** - в стеке выделяют участок памяти заданного размера для хранения данных, которые не доступны из внешнего блока, и они живут сколько живет процедура:

**enter размер, вложенность** (числа 0…31)

**enter** сначала ложит в стек текущее значение **ebp/bp**, чтобы по команде **leave** (освобождение области) **ebp/bp** содержал значение, которое было в нем до вызова **enter**. Затем содержимое **ebp/bp** уменьшают на 2 / на 4 (- зав-т от реж адресац) и в стек ложат данные, на которые указывает **ebp/bp**. Т о, уменьшение значения в **ebp/bp** и занесение в стек косвенно адресуемых через этот Рег данных повторяется вложенность -1 раз.

После этого в **ebp/bp** заносят значение, которое имел указатель вершины стека **esp/sp** после самого первого занесения в стек данных – после занесения в стек значения **ebp/bp**. Это надо для корректного срабатывания команды **leave**. Затем содержимое **esp/sp** уменьшают на значение, определяемое операндом **размер**. Т о, после выполнения **enter,** **esp/sp** указывает на младший байт только что выделенного участка памяти. Пр-р:

; пусть в этой точке содержимое **sp = 00fa h**,

значение по адресу **ss:[1232h] = 0fc76 h**, значение по адресу

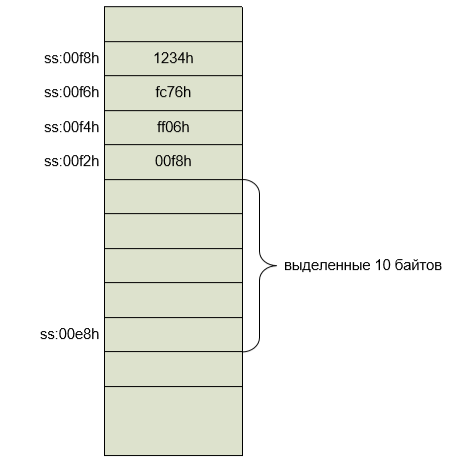
**ss:[1230h] = 0ff06h** – эти предположения - для определенности - какой??

**mov bp, 1234h**

**enter 10, 3**  ;выделяют 10 байт, вложенность = 3

; вид стека после выполнения команды enter - на рис,

; значение в **bp = 00 f8 h**, в **sp = 00 e8 h**.



Если значение вложенности = 0, то в стек кладут только текущее значение **ebp/bp**, и сразу после этого **esp/sp** уменьшают. Перед выполнением **leave** Рег **ebp/bp** д содержать то же самое значение, что и сразу после выполнения **enter**. Если размер в **enter** не кратен 2/ 4 (для 16 / 32 битн реж адресации), то выделяемую область выравнивают на границу слова / двойного слова соотв-но.

Пр-р: для команды **enter 9, 0** под локальные переменные при 16 бит реж адресац стекового сегм-а выделяют 10 байт = 5-и 2 байтн эл-ов стека; при 32 бит – 12 байт = 3м 4байтн элементам стека.

Освобождение участка памяти, отведенного командой **enter** – команда **leave** (покидать):

**leave**

Сначала **leave** копирует содержимое **ebp/bp** в **esp/sp**. Затем **leave** берет с вершины стека значение (на неё указывает обновленный **esp/sp)**, загружаемое в **ebp/bp**. Т о, после выполнения **leave** выделенный участок памяти в стеке свободен, а **ebp/bp** содержит значение до вызова **enter**. Пр-р:

**.186**  ; пусть в **sp** – **00fa h** – для определенности

**mov bp, 1234h**

**enter 10, 3**  ; содержимое **bp** перед **leave** д соответствовать

**…** ; содержимому **bp** сразу после выполнения **enter**

**leave** ; после выполнения **leave** в **sp - 00fa h**, в **bp – 1234h**.

Пр-р проги как идет доступ к локальным переменным:

; в проге вызов процедуры, кот создает 3 переменные (локальные 1 байтные).

; процедура вычисляет сумму этих элементов и заносит ее в **al**

**stack1 segment stack use16** ; сегм-т стека

|  |  |
| --- | --- |
| **db 100h dup(?)**  **stack1 ends** | ; под стек отводят 256 байт (дебил дупло) |
| **code segment use16**  **assume cs:code, ss:stack1**  **Begin:** | ; сегм-т кода |
| **call near ptr calSum** | ; вызов процедуры |

**cmp al, 60** ; если после вызова процедуры значение в **al**

; = 60, то на экране б символ ‘!’

**jnz Еxit**

**mov ah, 02h**

**mov dl, '!'**

**int 21h**  ; вывод символа ‘!’:

**Еxit: mov ah, 4ch**

**int 21h**  ; выход из проги

**calcSum proc near**

**enter 3, 0** ; enter - для создания локальн переменных проц-ры

**mov byte ptr [bp-2], 10** ; инициализация переменных значениями 10,

**mov byte ptr [bp-3], 20 ;** 20 и 30

**mov byte ptr [bp-4], 30**

**xor al, al**

**add al, [bp-2]** ; суммирование значений локальных переменных

**add al, [bp-3]**

**add al, [bp-4]**

**leave**

**retn**

**calcSum endp**

**code ends**

**end Begin**

Часть 7. **13.2. ПЕРЕДАЧА ПАРАМЕТРОВ ПРОЦЕДУРАМ.**

# **13.2.1. Общие сведения.**

3 способа передачи параметров процедурам через: 1) Рег-ы, 2) общую область памяти: глобальные переменные и 3) стек (тоже общ память).

# **13.2.2. Передача параметров через Рег-ы (1й способ).**

Использование Рег-ов Проц-ра – самый быстрый и простой. Суть – сохранить нужные параметры в Рег-х и использовать их в процедуре. Это дает, что команда **call** не меняет никаких Рег-ов (кроме **eip/ip** и **cs**). Пр-р:

**.model small**

**.data**

**msg1 db "First", 10, 13, '$'**

**msg2 db "Second", 10, 13, '$'**

**msg3 db "Third", 10, 13, '$'**

**.stack 100h** ; под стек отвели 256 байт

**.code**

**Begin:**

**mov ax, @data**

**mov ds, ax**

|  |  |
| --- | --- |
| **mov dx, offset msg1** | ; процедуре передают 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 **dx** проц-ре передают эфф-ый адрес строки для выв

**mov ah, 9**

**int 21h** ; вывод строки

**ret**

**Show endp**

**end Begin**

«+» метода передачи параметров через Рег-ы: наиболее быстрый и простой.

«-»: 1) ограниченное кол-во Рег-ов, 2) ограничение размера передаваемых параметров, 3) Рег надо сохранять, т к значение Рег не д меняться до места в процедуре.

# **13.2.3. Передача параметров через глобальные переменные (2й способ).**

Вызывающая и вызываемая проги используют общую область памяти. Пр-р:

**.model small**

**.stack 100h**

**.data**

**switcher dw ?**  ;общая переменная, б им эффект-ый адрес строки (вместо **dx**)

**msg1 db "First", 10, 13, '$'**

**msg2 db "Second", 10, 13, '$'**

**msg3 db "Third", 10, 13, '$'**

**.code**

**Begin:**

**mov ax, @data**

**mov ds, ax**

|  |  |
| --- | --- |
| **mov switcher, offset msg1** | ; передаваемый процедуре параметр (вместо dx) |
| **call Show** | ; вызов процедуры |
| **mov switcher, offset msg2** | ; передаваемый процедуре параметр |
| **call Show** | ; вызов процедуры |
| **mov switcher, offset msg3** | ; передаваемый процедуре параметр |
| **call Show** | ; вызов процедуры |

**mov ah, 4ch**

**int 21h**

**Show proc near**

**mov ah, 9**

**mov dx, switcher** ; смещение строки, всё равно в **dx** отдали

**int 21h**

**ret** ; вывод строки **ret**

**Show endp**

**end Begin**

«+» передачи парам-ов через общую обл памяти: гибкость и нет «-» (1) м-да.

«-»: 1) проц-ру невозм исп-ть вне проги где не описана глобальн переменная.

# **13.2.4. Передача параметров через стек (3й способ).**

Суть: перед вызовом процедуры в стек ложат параметры, которые ей надо передать. Затем в вызванной процедуре идет обращение к этим параметрам, но после выполнения **call,** на вершине стека всегда лежит **адрес возврата**. Пр-р:

|  |  |
| --- | --- |
| **op1 dw ?**  **op2 dw ?**  **…** |  |
| **mov op1, 300** | ; инициализация перем-ой op1 |
| **mov op2, 200** | ; инициализация перем-ой op2 |
| **push op1** | ; занесли в стек значения op1 (лежит на дне) |
| **push op2** | ; занесли в стек значения op2 (лежит на вершине стека) |
| **call near ptr calcSum**  **…** | ; вызов ближней процедуры |
| **calcSum proc near** | ; проц-ра, вычисляет сумму параметров и возвр в **ах** |
| **mov bp, sp** | ; т к **sp** нельзя использовать для косвенной адресации, |
|  | ; то вместо **sp** используют **bp** |
| **mov ax, [bp+2]** | ; в ax б число **200** | |
| **add ax, [bp+4]**  **retn 4** ; **calcSum endp** | ; в ax б сумма 200 + 300 = 500  возврат из процедуры, переданные параметры удаляют |

Для **дальнего** вызова процедур на вершине стека лежит пара **cs:eip/ip**, поэтому на последний переданной через стек параметр (лежит на вершине) указывает значение **sp+2** (16 битн реж адресац) / **sp+6** (32 битн).

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 м выполнять команды **pop** после возврата из процедуры.

2 вида передачи параметров, используя стек (по значению, по адресу):

1) передача по значению (в стеке – сами значения переменных, и процедура работает с ними; т о, в вызывающей проге сами переменные не изменяют);

2) передача по адресу (в стеке – адрес перем-ой, и процедура м изменять значение этой перем-ой).

«+» способа передачи параметров через стек: 1) самый гибкий, 2) любое кол-во параметров, 3) процедура независима от внешней проги и м использовать в других прогах.

«-»: 1) больше размер кода проги, 2) меньше быстродействие.

Часть 8. **14.1. ВЫРАЖЕНИЯ И ОПЕРАТОРЫ.**

**14.1.1. Общие сведения: арифм, логич, сравнен.**

Выражения – чтобы легче писать проги.

**Выражение** – совокупность переменных + констант, связанных операторами. Операторы реализуют арифметические, логические, и другие операции.

Арифметические **операторы**: сложен (+), вычит (-), умнож (\*), целочисленное деление (/), остаток от деления (mod), унарные + и -.

Логические операторы: логическое: умножение (and), сложение (or), исключающее / (xor), + сдвиги влево (shl) и вправо (shr).

Операторы сравнения = 1, если условие истинно **true**, и 0 – ложь **false**. **Операторы** сравнения: = (eq), > = (ge), > (gt), < = (le), < (lt) и не = (ne).

Значения выражений вычисляют только на этапе транслирования проги, но не на этапе ее выполнения. Пр-р: б неверным выражение **al\*7**, т к транслятору не известно содержимое Рег **al**. Пр-р:

|  |  |
| --- | --- |
| **add al, 5\*6** | ; к значению в al прибавить число 30, число 5 было  ; умножено на 6 во время транслирования |
| **mov bx, 1 shl 3** | ; занесли в **bx** = **8**, здесь **shl** – оператор, число 1 двиг влев на |

;этапе трансляц на 3 =0001b: 3 =1000b, а в Маш Ком ложат рез вычис выр-ия

**14.1.2. Операторы, присваивающие атрибуты ptr, short, this.** (опер-т Лена)

1) Оператор **ptr** - pointer (указатель) - для явного указания типа (/ размера) выражения:

**тип ptr выражение.**

Атрибуты - это значение, характеризующее объект в его классе (имя, значен).

Параметр **тип** м иметь значения / имена (атрибуты):

|  |  |  |  |
| --- | --- | --- | --- |
| Имя Значение | Имя Значение | dword 4 | **far 0FFFEh** |
| byte 1 | tbyte 10 | qword 8 |  |
| word 2 | **near 0FFFFh** |

Типы **near** и **far** используют с метками, остальные типы – с операндами из памяти. Пр-р:

**op1 dw ?**

**…**

**mov al, byte ptr op1** ; в **al** загружают младший байт перем-ой **op1**

**call far ptr Proc1** ; вызов дальней процедуры

**Proc1 proc**

**…**

**Proc1 endp**

2) Оператор **short** устанавливает тип метки **short –** когдарасстояние м/у меткой и командой перехода – **-128..127** байт:

**short метка.**

Длина Маш Ком с меткой **short** короче любой команды перехода.

3) Оператор **this** (этот) создает операнд, его адрес – текущее значение смещения относительно начала сегм-а:

**this тип**

Тип - определяет **как** трактовать извлекаемый из памяти операнд. Параметр тип м б любой: byte, …, far. Пр-р:

**mov bx, this word-2** ; занесли в **bx** последнее слово предыдущей команды

**14.1.3. Операторы, возвращающие значение high, low, seg, offset, type, length, size, $.** (оператор Лена)

1) Оператор **high** (высокий) возвращает старшие 8 бит выражения:

**high выражение.**

Пр-р: **mov bl, high 1234h** ; занесли в **bl** число **12**h

2) Оператор **low** - low (низкий) возвращает младший байт выражения:

**low выражение**.

Пр-р: **mov bl, low 1234h** ; занесли в **bl** число 34h

3) Оператор **seg** - segment (сегм-т) возвращает значение сегм-а (сегм-тная компонента), в котором лежит выражение, выражение – перем-ая, метка, имя сегм-а:

**seg выражение**

**op1 dw ?**

**…**

**mov ax, seg op1** ; занесли в **ax** сегм-тную компоненту адреса перем-ой **op1**

**mov es, ax** ; инициализация **es**

4) Оператор **offset** (**смещение**) – возвращает число байт **м/у выражением и началом сегм-а** (где оно определено):

**offset выражение**

где **выражение** – метка, имя сегм-а / перем-ой. Если имя сегм-а, то возвращают смещение **от начала** сегм-а **до последнего байта** (сгенерированного) в этом сегм-те.

**data segment**

**str1 db 5 dup(?)**

**data ends**

**…**

**mov dx, offset str1** ; занесли в **dx** эффективный адрес перем-ой **str1**

; (смещение относительно сегм-а **data**)

5) Оператор **type** (тип) возвращает число байт того типа (для хранения перем-ой), каким является выражение; для метки **near** возвращает 0FFFFh (65535), для **far** - 0FFFE h (65534):

**type выражение**.

|  |  |  |
| --- | --- | --- |
| **op1 dd ?**  **op2 dw 5 dup(?)**  **…** |  |  |
| **mov al, type op1** |  | ; занесли в al 4 (тип **dd** 4 байта) |
| **mov bl, type op2** |  | ; занесли в bl 2. (тип **dw** 2 байта) |

6) Оператор **length** возвращает число единиц типа byte, word, dword, qword, занимаемых перем-ой:

**length переменная**

Тип перем-ой определяют в единицах измерения ее длины. Т е, возвращаемое значение оператором **length** – число перед оператором **dup** (в описании перем-ых) только для переменных и описанных оператором **dup**, отличное от 1. + вернут 1 и строковые константы. Пр-р:

**op1 db 5 dup(?)**

**str1 db 'abc'**

**…**

**mov al, length op1** ; в Рег **al** заносят **5**

**mov cl, length str1** ; в Рег **cl** заносят **1**

7) Оператор **size** возвращает число байт, занимаемых перем-ой:

**size переменная**.

Значение, возвращаемое **size** = длине **length** перем-ой, повторенной **type** раз,

**size переменная = (length** переменной**) × (type** переменной**)**

**op1 dw 5 dup (?)**

**…**

**mov si, size op1** ; в **si** - число 10 = 5\*dw = 5 \* 2 = 10

8) Оператор **$** возвращает текущее значение **счетчика** адреса - т е смещение адреса текущего оператора:

**$ выражение**.

**jmp $+3** ; переход на метку **L1**

**db ? ; резервирование байта памяти**

**L1:**

Часть 9. **14.2. МАССИВЫ.**

**14.2.1. Объява массива. Работа с массивами.**

**Массив** – упорядоченный набор 1типных элементов.

Для задания массивов целых чисел - оператор дублирования **dup**.

Пр-р: задание массива из **10** 2байтных (dw) элементов:



В Ассемблере нельзя объявлять 2- и более мерных массивов: надо самой их обрабатывать - хранить по строкам (см рис.).

Массив **A m×n (m строк × n столбов)**:

**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]**  - в памяти идут в 1 строку: **a[1,1]** … a[1,n]… **a[m,n]**

; Пр-р поиска максимального элемента в 3й строке массива 2 х10 2байтных ; элементов: ПРОГА НИЧЕГО НЕ ВЫДАЕТ, но нет ошибок.

**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(?)** ; под стек отвели 256 байт

**stack1 ends ; -------------------------------------**

**code segment**

**assume cs:code, ds:data, ss:stack1**

**begin:**

**mov ax, data**

**mov ds, ax**  ; занесение в **ds** адреса начала сегм-а данных data

; позиционирование в массиве на 1-ый элемент 3-ей строки = 10

**mov ax, 2**  ; нумерация - с 0

**mov cl, 10\*type arr** ; 10 **2**байтных элементов

**mul cl** ; в **ax** – позиция первого эл-та 3й строки: ax\* cl???

**mov bx, ax** ; занесение в bx значения из ax, т к **ax** нельзя

; использовать для косвенной адресации

**mov ax, arr[bx]** ; занесение в **ax** первого элемента 3й строки

; теперь найдем максимум в этом массиве:

|  |  |
| --- | --- |
| **mov cx, 10-1**  **L1:** | ; будем сравнивать 9 элементов |
| **add bx, type arr** | ; увеличение на 2 индекса массива |
| **cmp ax, arr[bx]** | ; в **ax** – текущий максимум |
| **jge noNewMax** | ; если в **ax** число, меньшее элемента массива, то |
| **mov ax, arr[bx] noNewMax:** | ; занесение нового максимума в **ax** |
| **loop L1**  **mov ah, 4ch** | ; цикл, после его выполнения в ax б число 251 |
| **int 21h**  **code ends** | ; выход из проги |

**end begin** - НИЧЕГО НЕ ПОНЯЛА

Часть 10. **14.3. СТРУКТУРЫ.**

# **14.3.1. Объява структур. Работа со структурами struc,** оператор **.** (точка)**.**

**Структура** – тип данных; определенное к-во эл-ов (полей) разного типа.

**Поля – э**лементы в составе структуры. Т е, **структура** – объединение разных полей под 1 именем (record in Pascal).

Для задания структур - директива **struc** - structure (структура):

**имя\_структуры struc**

**… ; определение полей структуры имя\_структуры ends**

Поля (эл-ты) стр-ры определяют **директивой определения данных** (db / dw).

**date struc**

**year dw ?**

**month db 10**

**day db 28**

**other db 10 dup (?)**

**date ends**

**struc** не выделяет памяти под структуру, а служит лишь для описания шаблона. Для создания физической перем-ой (для выделения памяти под переменную типа структуры):

**[имя\_перем-ой] имя\_структуры <[значения]>**

В <> - начальные значения полей по порядку. Если значения параметров не указывают (пустой список параметров, <>), то все поля инициализируют значениями, которые они имели в шаблоне структуры. Если надо инициализировать какое-то определенное поле значением, которое оно имело в структуре, то его просто отделяют запятой, не указав значения. В памяти поля перем-ой (типа структуры) лежат линейно друг за другом, как и в шаблоне. Пр-р:

**date struc**

**year dw ?**

**month db 10**

**day db 28**

**other db 10 dup (?)**

**date ends**

**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**

**mov event1.month, 12** ; занесение в поле **month** перем-ой **event1** значения **12**

Определение массивов структур. Пр-р:

**events date 10 dup(<>)**

**…**

**mov bx, 4**

**mov events[bx].day, 5** ; занесли в 4й эл-т массива **events** в поле date 5

Часть 11. **14.4. ЗАПИСИ.**

**14.4.1. Объява записей. Работа с записями record, width, mask.**

**Запись (record)** – тип данных, состоит из фиксированного числа битовых полей. Для задания записей - директива **record** (== Паскале):

**имя\_записи record имя\_поля : длина\_поля [= значение] [,]**

**[имя\_поля : длина\_поля [= значение] ] [, …].**

**record** - для определения 2ичного (**бит**ового) набора в байте, слове, двойном слове. Общий размер записи = сумме длин ее полей <= 8, 16, 32 бит. Если сумма всех полей < 8, 16, 32 бит, то элемент записи выравнивают на границу байта, слова, двойного слова.

Пр-р: элемент длиной 13 бит **выравнивают** до 2байтного значения (16). В **record** длины полей задают в битах (1,4,3). Необязательное указание **значения** определяет **значение**, которым инициализируют данное поле:

**bitSet record b1 : 1, b2 : 4, b3 : 3 = 101b.**

Директива **record** описывает шаблон записи. Поэтому для использования записей надо объявлять переменные:

**[имя\_перем-ой] имя\_записи <[значения]>**

Здесь переменные имеют тот же смысл, что и числовые константы. В <> - значения, которыми инициализируют поля этой перем-ой. Если надо проинициализировать все поля значениями, совпадающими теми, что описаны при задании записи в директиве **record**, то просто опусти все значения в угловых скобках (просто <>). Значения полей перечисляют через запятую, а если надо задать значение, совпадающее со значением в описании **record**, то это значение просто выделяю запятыми:

**bitSet record b1:1, b2:4, b3:3 = 101b**

**…**

**set1 bitSet <0, 0000b, >** ; поле b1=0, b2=0000b, b3 = 5 = 101b

**set2 bitSet < , 1001b>** ; поле b1=0, b2=1001b, b3 = 5 = 101b.

В отличие от структур, к полям записей нельзя обращаться в других случаях (кроме инициализации перем-ой). В тексте проги переменные типа запись используют везде, где м указать Непосред Оп-д длины байт, слово, двойное слово:

**bitSet record b1:1, b2:4 = 0, b3:3 = 101b**

**set1 bitSet <1, , 011>**

**…**

**mov bl, set1** ; занесли в **bl** число **1 0000 011b** (4 нуля)

2 спец оператора для работы с записями: **width** и **mask.**

1) Оператор **width** (ширина) возвращает **длину** в битах поля записи / всей записи:

**width имя\_поля**

**width имя\_записи.**

**bitSet record b1:1, b2:4, b3:3 = 101b**

**…**

**mov bl, width bitSet** ; в **bl** заносят значение 8 = 1+4+3

**mov dl, width b2**  ; в **dl** заносят значение 4

2) Оператор **mask** (маска из 00 и 11 в битах) возвращает число, где единицы поставлены в указанных местах (позициях), все остальные биты – 00:

**mask имя\_поля**.

**bit\_set record b1:1, b2:4, b3:3 = 101b ;** всего 1+4+3 = 8 бит

**…**

**mov bl, mask b2**  ; в **bl** заносят **01111000**b (**bl** – 8 бит)

**mov ax, mask b3** ; в **ax** заносят **0000 0000 0 0000 111**b (**ах** – 16 бит)

При использовании имен полей в командах, в формат Маш Ком-ы транслятор заносит Непосред Оп-д, его значение - число, на которое надо сдвинуть это поле, чтобы выровнять его по правому краю:

**bitSet record b1:1, b2:4, b3:3 = 101b**

**…**

**mov ax, b3** ; в **ax** заносят 0, т к маска поля **b3** = 0000**0111b**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **add dl, b2** | |  | ; к содержимому **dl** прибавляют число 3, т к маска  ; поля b2 = 01111000b (№ с 0) | |
| **mov cl, b1** | |  | ; в cl заносят 7, т к маска поля **b1** **= 10 00 00 00b** | |
|  |  | | |

Часть 12. **15.1. ОБЩИЕ СВЕДЕНИЯ о макросредствах, макросах.**

# **15.1.1. Понятие о макросредствах Ассемблера.**

- для облегчения работы при программировании на Ассемблере.

**2 части транслятора: 1)** непосредственно **транслятор**, формирующий объектный модуль, и 2) **макро-ассемблер**, обрабатывающий макросредства. Макросредства: 1) макро-определения, 2) макро-имена и 3) макро-директивы.

При компилировании проги сначала работает макроассемблер, а затем – сам транслятор.

Суть макросредств – использовать **подстановки**, которые замещают одну символьную последов-сть другой (симв-ой последов-тью). Эта последов-сть м описывать данные и программные коды.

Главное - на входе макроассемблера м б текст проги, далекий по виду от проги на Ассемблере, а на выходе – текст на чистом Ассемблере.

1) Макроопределение (/ **макрос**) – блок исходных команд / директив, определяемый особым образом с помощью макро-директив и имеющий имя. 2) Макро-команда – имя макроопределения, вместо которого в текст проги подставляют группу команд / директив Ассемблера. 3) Макро-директивы – спец директивы для обработки макро-определений.

# **15.1.2. Преимущества и недостатки использования макросов.**

«+»: 1) < объема текста проги и < времени разработки проги. 2) Макросы заменяют зашифрованные мнемоники Ассемблера на макро-имена удобного вида. 3) М подключать неск-ко файлов с исходным текстом проги к 1у файлу директивой **include**, чтобы использовать библиотеки макросов.

«-»: 1) макросы хранят в виде текста и, следовательно, д транслировать отдельно для каждой отдельной проги. Поэтому библиотека макросов увеличивает время транслирования проги.

Часть 13. **15.2. СОЗДАНИЕ И ВЫЗОВ МАКРООПРЕДЕЛЕНИЙ.**

**15.2.1. Создание макроопределений macro**, **endm.**

**Макроопределение** (/ **макрос**) - блок исходных команд и директив Ассемблера, начинается и заканчивается директивами **macro** и **endm**:

**имя\_макрокоманды macro [формальн\_пар-тр1, формальн\_пар-тр2, …]**

**…** ; команды и директивы, образующие тело макроОпределения

**endm**

МакроОпределения располагают:

1) в исх1 тексте проги до места, в котором их применяют;

2) в отдельном файле: чтобы эти макроопределения б доступны в конкретной проге, надо в начале исходного текста проги записать директиву **include имя\_файла.** Его «-»: в исходный текст проги включают все макроОпред-ия.

**message macro msg1**

|  |  |
| --- | --- |
| **lea dx, msg1** | ; в **dx** загрузили эффективный адрес **ячейки** msg1,  ; передаваемой как параметр макроОпределению |
| **mov ah, 9** | ; ф-ция вывода строки на экран |
| **int 21h**  **endm** | ; прерывание DOS |

**15.2.2. Вызов макроОпределений.**

Чтобы в текст проги вставить блок из макроОпределения - макрокоманды:

**имя\_макрокоманды [фактический** параметр1**,** фактическийпараметр2**,** …**]**

Рез-т применения такой конструкции в проге - ее замещение строками из тела макроопределения: у Макрокоманды есть список фактических параметров, которые корректируют макроопределение. Места в теле макроОпределения, которые будут заменять фактическими параметрами из макрокоманды, обозначают с помощью **формальных параметров.**

Т о, в рез-те применения макрокоманды в проге формальные аргументы в макроОпределении замещают соотв-ими фактическими аргументами. Процесс такого замещения называется **макрогенерацией**, а рез-т этого процесса - **макрорасширение**. Пр-р:

**sum macro op1, op2**

**add op1, op2**

**xor op2, op2**

**endm**

**…**

**mov ax, 5**

**mov cx, 10**

**sum ax, cx**  ; в **ax** = **cx+** **ax = 5+10=15**, обнуление **cx** (после

; этого данного фрагмента в **ax** б **15**, в **cx – 0**)

Каждый фактический параметр - строка символов, для их формирования применяют правила: 1) в строке не д б символов пробел, точка, запятая, точка с запятой, 2) чтобы указать, что некоторый символ внутри строки, представляющей фактический параметр - символ (а не разделитель / ограничивающие скобки), применяют спец оператор «**!**» / «<>». Его ставят перед описанным выше символом.

Часть 14. **15.3. МАКРООПЕРАТОРЫ И МАКРОДИРЕКТИВЫ.**

# **15.3.1. Макрооператоры &, <>, !, %, ;;.**

1) Макрооператор **&** (амперсант) – чтобы параметр, переданный как операнд макро-определению / макро-директиве, заменяли значением **до** обработки строки макроассемблером. Пр-р:

; макрос выполнит команду **push** **eax**, если его вызвать как **pushReg** **pushReg** **macro letter**

**push e&letter&x ; e&a&x???**

**endm**

2) Макрооператор **<>** (угловые скобки): текст в этих скобках рассматривают как текстовую строку, даже если там пробелы и другие разделители. Используют при передаче текстовых строк как параметры для макросов.

3) Макрооператор **!** (восклицат знак) - == <>, но действует только на 1 следующий символ; если этот символ – запятая / угловая скобка – б передан макросу как часть параметра.

4) Макрооператор **%** указывает, что текст за ним – выражение и д быть вычислен, чтобы передавать как параметр в макрос не само выражение, а его рез-т.

5) Макрооператор **;;** (2 точки с запятой) – начало макро-комментария, его текст не идет в листинг проги.

# **15.3.2. Макро-директивы** повторения, генерации макро-расширения, **local.**

2 группы макро-директив д/модификац набора строк и порядка их следов-ия: 1) макро-директивы повторения (**while, rept, irp, irpc**) – для создания макросов, содержащих неск-ко идущих подряд одинаковых последоват-тей строк; м частично модифицировать эти строки;

2) макро-директивы управления процессом генерации макро-расширения (**exitm, goto**) – м как исключать отдельные строки из макрорасширения, так и прекращать процесс генерации.

3) Макро-директива **local** (локальный) – для локализации набора меток внутри макроопределения:

**local формальное\_имя1[, формальное\_имя2,…]**

**local** – для многократного вызова макроОпределений, в которых есть метки. Для уникальности имен меток, для любого формального имени списка, порождается имя **??xxxx**. 2 символа ‘?’ и xxxx – 16 битн число 0...0ffffh.

**local** – только в макроопределениях и идет сразу после директивы **macro**:

; в макросе вычисляют сумму 2 операндов и рез-т сравнивают с числом **100**;

; если > 100, то идет переход на метку **L1**, иначе – на **L2**

**sum macro op1, op2**

**local L1, L2** ; локализация меток

**add op1, op2** ; op1 = op1 + op2

**cmp op1, 100**

**jg L1** ; if op1 > 100 then 🡪 L1

**mov op2, 5** ; else op2:= 5

**jmp L2** ; L2 = end

**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 – **IL, MSIL**), где есть вся инфа о проге, но не привязана к определенному языку программирования. Для запуска проги надо спец 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 .NET использует стандартные классы.

# **16.2. ДОСТОИНСТВА ПЛАТФОРМЫ .NET**

1) основана на единой объектно-ориентированной модели (ООП); все сервисы оформлены как единая иерархия классов.

2) приложения в архитектуре.NET – много-платформенные (т к промежуточное представление .NET не привязано к какой-то платформе).

3) автоматическое управление ресурсами решает проблемы: утечки памяти, повторное освобождение ресурса и т.п. В .NET нет способа явно освободить ресурс!

4) не использует реестр Windows за счет использования механизма метаданных.

5) приложения для **.NET** м сертифицировать на безопасность: Код для .NET м проверить на безопасность для гарантии, что приложение не навредит пользователю.

6) Обработка ошибок в .NET – через механизм исключительных ситуаций –некоторые ошибки обозначают кодом ошибки платформы Win32, некоторые возвращают HRESULTS и т.п.

# **16.3. НЕДОСТАТКИ ПЛАТФОРМЫ .NET**

1) сильное замедление выполнения проги, т к м/у исходным языком и машинным кодом есть д/п-ный уровень, **MSIL**. 2) при ее создании основной упор был на С++/Java-подобные языки, что ограничивает возм-ти интеграции некоторых языков с более богатыми возм-тями. 3) при развертывании приложения – использование разделяемых библиотек. Из-за этого установка нового приложения (-простое копирование всех файлов в определенный каталог) м привести к прекращению работы раннего приложения.