**Flat Assembler – FASM**

**Инструменты**

* FASM 1.68 или выше (последняя версия 1.71.60) for Windows —ассемблер для x86.
* FASM 1.71.21 или выше (последняя версия 1.71.60) for Windows —ассемблер для x86.
* PROC16 — макросы для создания и вызова процедур (16-битный код).
* Turbo Debugger v5.0 —отладчик под DOS.
* FASM Editor v1.0 — среда разработки для FASM.
* FASM Editor v2.0 — среда разработки для FASM версия 2.0.

**Документация**

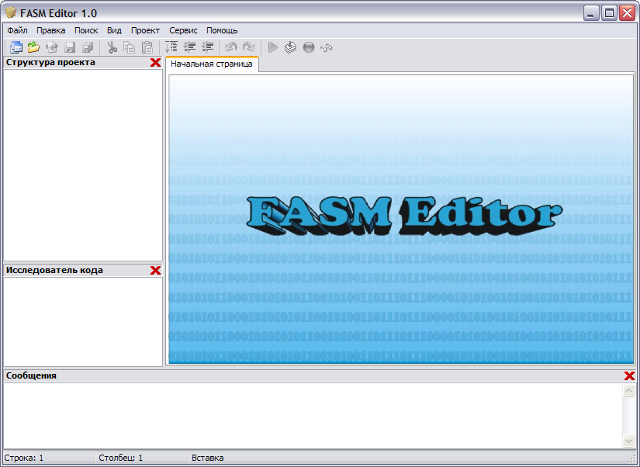
* Help! —справочник по функциям MS-DOS, BIOS и аппаратным ресурсам PC/XT/AT.
* FASM Manual — справочник по ассемблеру FASM v1.64 (на русском языке).

**Книги**

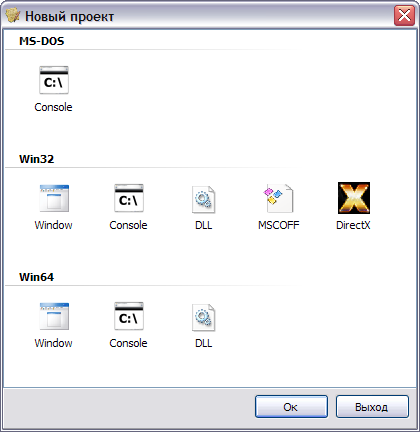
* «Intel Architecture Software Developer’s Manual. Volume 2: Instruction Set Reference»

FASM Editor —среда разработки для FASM, в которой есть все инструменты для удобного программирования на ассемблере.

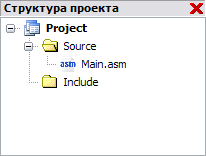
Fresh (IDE) — интегрированная среда разработки на визуальном языке ассемблера для Microsoft Windows со встроенным FASM. Поддерживает сборку программ для тех же платформ, что и FASM: DOS, Linux, FreeBSD, BeOS, MenuetOS.

[](http://asmworld.ru/content/tools/fasm_editor/shot00.png)

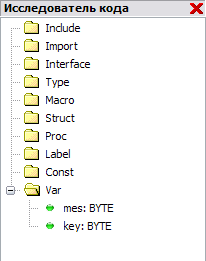
Установка программы не требуется, её можно запускать с флэш карты. FASM Editor позволяет создавать различные типы проектов для DOS, Win32 и Win64.



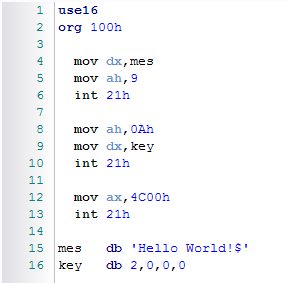
В верхнем левом окне отображается структура проекта:



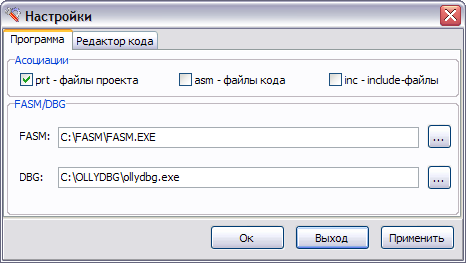
Под структурой проекта расположен «исследователь кода», который показывает переменные, метки, процедуры и ещё много всего полезного:



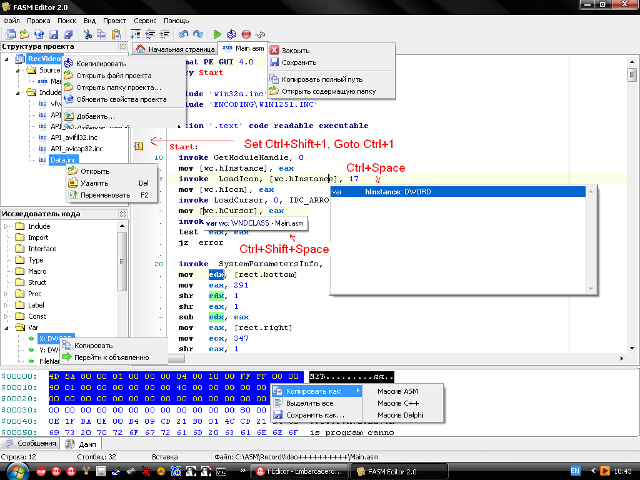
Очень аккуратный редактор кода с настраиваемой подсветкой синтаксиса.



Настроек у программы совсем немного — главное указать путь к FASM и к отладчику



Вторая версия среды разработки FASM Editor!

[](http://asmworld.ru/content/tools/fasm_editor_v2.0/shot00.png)

Учебник

Учебный курс программирования на ассемблере FASM

**Часть 0. Функции и возможности ассемблера**

В настоящее время существует множество языков программирования. Созданы самые разные языки, удобные для решения любых задач. Большинство этих языков является языками высокого уровня.

Ассемблер — это один из самых ранних языков программирования. До него было лишь программирование в машинных кодах

Знание ассемблера позволяет обеспечить:

1. Глубокое понимание работы компьютера и операционной системы.
2. Максимальную гибкость при работе с аппаратными ресурсами.
3. Оптимизацию программ по скорости выполнения.
4. Оптимизацию программ по размеру кода.
5. Дизассемблирование и отладку.

**Глубокое понимание работы компьютера и операционной системы.**

Если вы пишете программу на языке высокого уровня, то знание ассемблера поможет понять, как будет выполнятся программа, как хранятся переменные, как вызываются функции. А это позволит избежать многих очень неприятных ошибок. Специалист, владеющий ассемблером, будет лучше программировать и на других языках.

**Максимальная гибкость при работе с аппаратными ресурсами.**

Языки высокого уровня ограничены компилятором и используемыми библиотеками. А такие современные языки, как Java и C# вообще не позволяют работать с аппаратными ресурсами и операционной системой напрямую. Ассемблер позволяет работать с аппаратными ресурсами напрямую.

**Оптимизация программ по скорости выполнения.**

Современные компиляторы довольно неплохо оптимизируют код, поэтому писать на ассемблере все подряд, конечно, не имеет смысла. Однако, если вы пишите программы для шифрования или архивации больших файлов, то применение ассемблера позволит в несколько раз увеличить скорость выполнения программы. Причем достаточно реализовать на ассемблере небольшой критически важный участок программы, который производит вычисления или сложные преобразования, а интерфейс может быть написан на языке высокого уровня.

**Оптимизация программ по размеру кода.**

Программа на ассемблере, как правило, значительно меньше аналогичной программы на другом языке программирования. Для современных персональных компьютеров и серверов с терабайтными дисками и гигабайтами памяти это, конечно, вряд ли играет большую роль. Но для микроконтроллеров, где всего несколько килобайт памяти, маленький размер программы очень важен. Чем меньше программа, тем меньше памяти требуется и тем проще и дешевле будет используемая микросхема.

**Дизассемблирование и отладка.**

Знание ассемблера позволяет «раскрыть» и «понять» любую программу дизассемблером и изучить механизм её работы! Используя ассемблер можно «залезть» внутрь любой программы и посмотреть как она работает. Иногда бывает, что попадается интересная программа и не понятно, что там внутри, как она написана. Зная ассемблер, можно заглянуть внутрь любой программы и удовлетворить свое любопытство, даже не имея исходников.

Ассемблер очень может помочь при отладке. Иногда случаются ошибки и в компиляторах. Вроде бы корректно написанный код выполняется вовсе не так, как предполагалось. Чтобы обнаружить такую ошибку надо посмотреть, во что скомпилировался код, а разобраться в этом без ассемблера невозможно.

**Часть 1. Необходимые инструменты**

Учиться программировать начнем с процессора Intel 8086. Будем писать программы под DOS. Программирование под Windows и Linux сложнее. Поэтому начнем с простого и понятного 16-битного процессора 8086.

Практическая ценность от программирования под DOS в наше время не очень большая, если вы, конечно, не собираетесь тесно работать с этой операционной системой. Но она позволит нам быстро освоить основы ассемблера, а потом мы уже перейдем к программированию под 32-битные системы.

Все программы можно запустить под Windows. Конечно, реально они будут работать в эмуляторе DOS, в режиме виртуального процессора 8086.

Для программирования на ассемблере нам прежде всего необходим компилятор. Наиболее известные компиляторы это TASM, MASM и FASM. В нашем учебном курсе мы будем использовать FASM. Это довольно новый, удобный, быстро развивающийся компилятор ассемблера, написанный на себе самом. Его преимущества — это поддержка сложных макросов и мультиплатформенность. Есть версии под DOS, Windows и Linux.

С его помощью можно сгенерировать файл любого формата, не обязательно исполняемый файл, так что FASM — это превосходный инструмент для экспериментов и исследований.

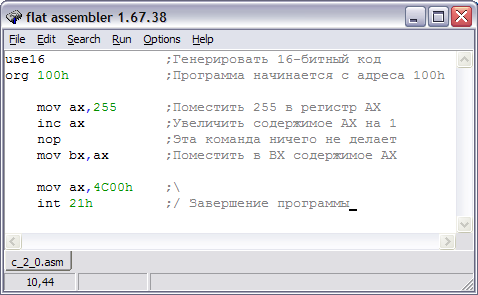
Последнюю версию FASM’a можно скачать с официального сайта <http://www.flatassembler.net/>.

На момент написания данной лекции последняя версия 1.71.60. Для установки содержимое архива надо распаковать в какую-нибудь папку. Например, пусть это будет *C:\FASM*.

Для отладки написанных программ будем использовать Turbo Debugger из пакета TASM. Пусть это будет *C:\TD*.

**Часть 2. Первая программа**

Напишем совсем простую программу. Мы будем писать только COM-программы под DOS, так они проще, чем EXE. Для того, чтобы написать программу, нам надо запустить fasmw.exe. Откроется окно, в которое можно вводить код:



В это окошко надо ввести следующее:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **ax**,255 *;Поместить 255 в регистр AX*  **inc** **ax** *;Увеличить содержимое AX на 1*  **nop** *;Эта команда ничего не делает*  **mov** **bx**,**ax** *;Поместить в BX содержимое AX*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы* |

**Первая строка** *«use16»* сообщает FASM’у, что нужно генерировать 16-битный код. Нам нужен именно такой для нашей первой программы. Точка с запятой — это символ комментария. Все что идет после *«;»* до конца строки игнорируется компилятором. Там можно писать все что угодно.

**Вторая строка** *«org 100h»* объясняет FASM’у, что следующие команды и данные будут располагаться в памяти, начиная с адреса 100h. Дело в том, что при загрузке нашей программы в память, DOS размещает в первых 256 байтах (с адресов 0000h — 00FFh) свои служебные данные. Нам эти данные изменять нежелательно.

Далее идут непосредственно команды! Программа на ассемблере состоит из команд процессора. Каждая команда обозначается мнемоникой (символическим именем). Например *«mov»*, *«inc»*, *«nop»* и т.д. После мнемоники могут идти операнды. Они отделяются одним или несколькими пробелами (или табуляцией).

Команды бывают без операндов, с одним или несколькими операндами. Если операндов больше одного, то они отделяются друг от друга запятыми.

Отступы не обязательны, но желательны — с ними код гораздо легче читать. Пустые строки игнорируются. Регистр символов значения не имеет. Можно писать и большими и маленькими буквами.

**Четвертая строка** определяет команду «поместить число 255 в регистр AX». *«mov»* — это мнемоника команды (от английского «MOVe»). AX — первый операнд — приёмник. 255 — второй операнд — источник. Первый операнд является регистром. Второй операнд — константа 255.

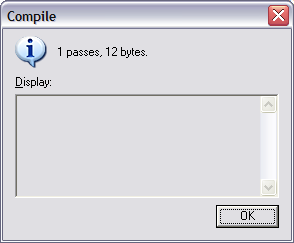
**Пятая строка**. Тут команда *«inc»* с одним операндом. Она заставит процессор выполнить инкремент, то есть увеличение на единицу. Единственный операнд — это регистр AX, содержимое которого и будет увеличено на 1.

**Шестая строка**. Команда *«nop»* — без операндов. Эта команда ничего не делает и ее функции мы рассмотрим ниже.

**Седьмая строка**. Снова команда *«mov»*, но на этот раз оба операнда являются регистрами. Команда скопирует в BX содержимое AX.

**Две последние строки** — это стандартное завершение процесса в DOS. Так мы будем завершать все наши программы. Команда *«mov»* должна быть понятна, а команду *«int»* мы рассмотрим ниже.

Чтобы откомпилировать программу надо выбрать меню *Run->Compile*. FASM предложит сохранить файл, если вы этого ещё не сделали, а затем скомпилирует. То есть переведет текст, набранный нами, в машинный код и сделает его программой. Файл с расширением .asm — это исходный код или исходник, обычный текстовый файл. При желании его можно открыть блокнотом )



Целых 12 байт получилось

В каталоге с asm-файлом появился файл .com — это и есть наша программа! Если в коде что-то неправильно, то в этом окне вы увидите сообщение об ошибке.

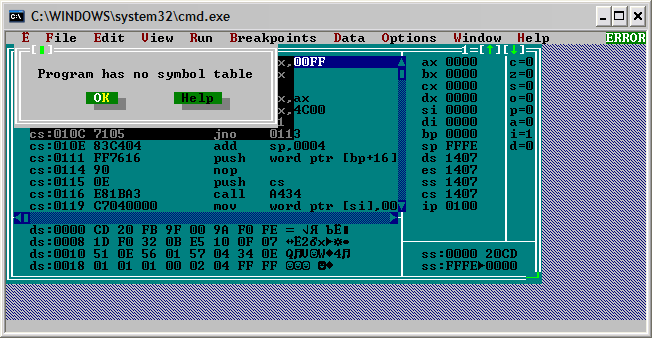
В следующем разделе мы рассмотрим как работать с отладчиком.

**Часть 3. Turbo Debugger**

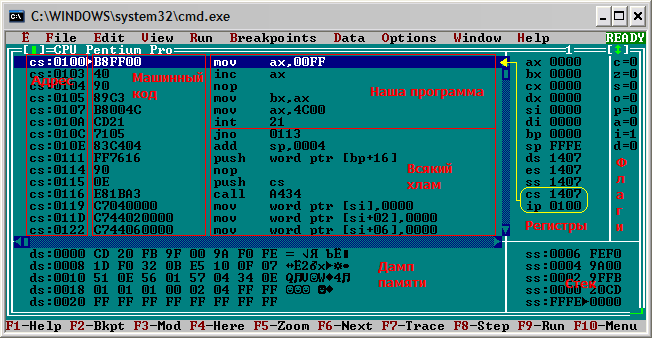
Прежде всего необходимо запустить отладчик. Для этого удобно использовать bat-файл. Создайте в каталоге программы текстовый файл, назовите его, например, «debug.bat». В него надо записать всего одну строку:

|  |  |
| --- | --- |
| 1 | C:\TD\td.exe <файл\_программы>.com |

После запуска этого bat-файла вы увидите примерно такое окно:



Сообщение означает, что в исполняемом файле нет специальных данных для отладки. Но нам эти данные и не нужны, потому что программа простая и понятная. Нажимаем ОК. Turbo Debugger отображает окно CPU, в котором можно увидеть, как выполняется программа.

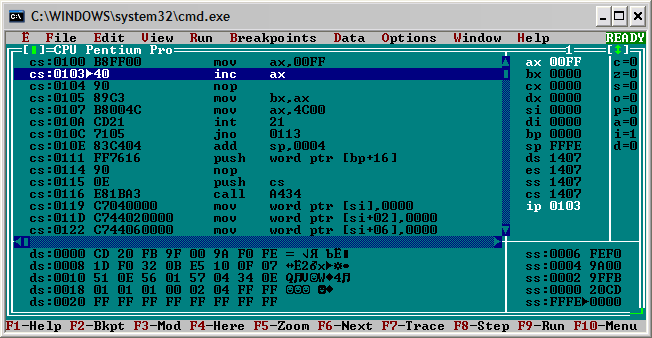


В большой области мы видим код нашей программы. Самый левый столбец — адреса, правее отображаются байты машинного кода, а ещё правее — символическое обозначение команд. Программа размещается в памяти, начиная с адреса 0100h в сегменте кода. В нашей программе всего 6 машинных команд, а за ними в памяти находится случайный «мусор» (точные значения неизвестны).

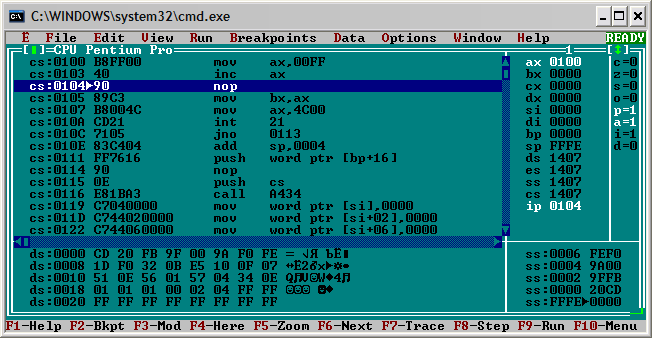
Обратите внимание, что отладчик показывает адреса и значения в шестнадцатеричном виде.

В правой части окна CPU отображаются регистры процессора и флаги. В нижней части можно увидеть дамп области памяти и стек. Стек — это специальная структура данных, с которой работают некоторые команды процессора.

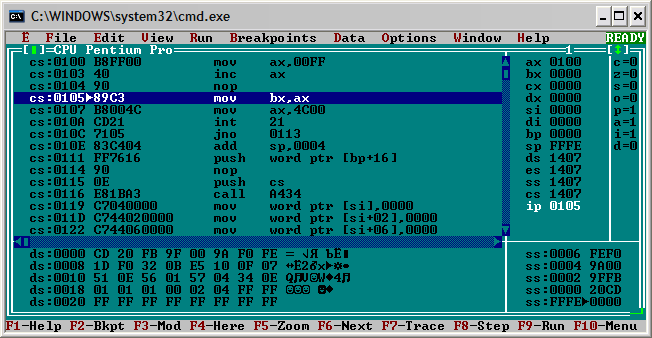
Адрес текущей машинной команды определяется регистрами CS и IP, эта команда показана выделенной строкой и стрелкой. Теперь нажмите F8, чтобы выполнить первую команду.



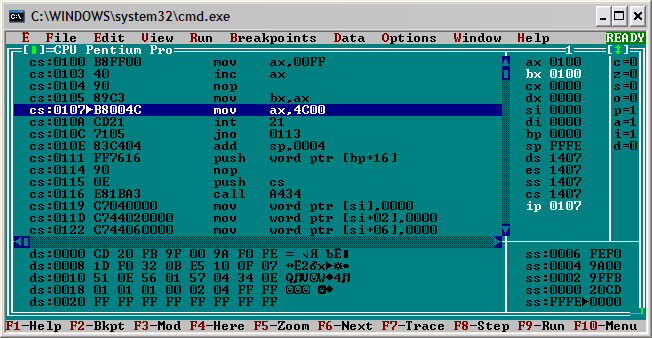
Теперь стрелка указывает на вторую команду. Изменившиеся регистры выделены белым цветом. Регистр AX теперь содержит значение 00FFh (то есть 255, т.е. команда «mov ax,255» была исполнена). Также изменилось значение регистра IP — оно увеличилось на размер выполненной машинной команды, а именно на 3. Теперь СS:IP указывает на следующую команду. Снова нажимаем F8.



Значение регистра AX увеличилось на 1 и стало равным 0100h (256). Значение IP тоже увеличилось на 1, потому что длина команды «inc ax» — 1 байт. Процессор выполняет программу последовательно, одну команду за другой. Ещё раз нажимаем F8.



Команда NOP ничего не делает. Меняется только значение IP — снова увеличивается на 1. Снова F8.



Значение BX становится равным AX. После ещё двух нажатий F8 программа завершается. Закрыть отладчик можно с помощью меню *File->Quit*.

**Часть 4. Регистры процессора 8086**

Для того, чтобы писать программы на ассемблере, необходимо знать, какие регистры процессора существуют и как их можно использовать. Все процессоры архитектуры x86 (даже многоядерные, большие и сложные) являются дальними потомками древнего Intel 8086 и совместимы с его архитектурой. Это значит, что программы на ассемблере 8086 будут работать и на всех современных процессорах x86.

Все внутренние регистры процессора Intel 8086 являются 16-битными:



Всего процессор содержит 12 программно-доступных регистров, а также регистр флагов (FLAGS) и указатель команд (IP).

**Регистры общего назначения (РОН)** AX, BX, CX и DX используются для хранения данных и выполнения различных арифметических и логических операций. Кроме того, каждый из этих регистров поделён на 2 части по 8-бит, с которыми можно работать как с 8-битными регистрами (AH, AL, BH, BL, CH, CL, DH, DL). Младшие части регистров имеют в названии букву L (от слова *Low*), а старшие H (от слова *High*). Некоторые команды неявно используют определённый регистр, например, CX может выполнять роль счетчика цикла.

**Индексные регистры** предназначены для хранения индексов при работе с массивами. SI (*Source Index*) содержит индекс источника, а DI (*Destination Index*) — индекс приёмника, хотя их можно использовать и как регистры общего назначения.

**Регистры-указатели** BP и SP используются для работы со стеком. BP (*Base Pointer*) позволяет работать с переменными в стеке. Его также можно использовать в других целях. SP (*Stack Pointer*) указывает на вершину стека. Он используется командами, которые работают со стеком.

**Сегментные регистры** CS (*Code Segment*), DS (*Data Segment*), SS (*Stack Segment*) и ES (*Enhanced Segment*) предназначены для обеспечения сегментной адресации. Код находится в сегменте кода, данные — в сегменте данных, стек — в сегменте стека и есть еще дополнительный сегмент данных. Реальный физический адрес получется путём сдвига содержимого сегментного регистра на 4 бита влево и прибавления к нему смещения (относительного адреса внутри сегмента). Подробнее о сегментной адресации рассказывается в части 31.

COM-программа всегда находится в одном сегменте, который является одновременно сегментом кода, данных и стека. При запуске COM-программы сегментные регистры будут содержать одинаковые значения.

**Указатель команд** IP (*Instruction Pointer*) содержит адрес команды (в сегменте кода). Напрямую изменять его содержимое нельзя, но процессор делает это сам. При выполнении обычных команд значение IP увеличивается на размер выполненной команды. Существуют также команды передачи управления, которые изменяют значение IP для осуществления переходов внутри программы.

**Регистр флагов** FLAGS содержит отдельные биты: флаги управления и признаки результата. Флаги управления меняют режим работы процессора:

* D (*Direction*) — флаг направления. Управляет направлением обработки строк данных: DF=0 — от младших адресов к старшим, DF=1 — от старших адресов к младшим (для специальных строковых команд).
* I (*Interrupt*) — флаг прерывания. Если значение этого бита равно 1, то прерывания разрешены, иначе — запрещены.
* T (*Trap*) — флаг трассировки. Используется отладчиком для выполнения программы по шагам.

Признаки результата устанавливаются после выполнения арифметических и логических команд:

* S (*Sign*) — знак результата, равен знаковому биту результата операции. Если равен 1, то результат — отрицательный.
* Z (*Zero*) — флаг нулевого результата. ZF=1, если результат равен нулю.
* P (*Parity*) — признак чётности результата.
* C (*Carry*) — флаг переноса. CF=1, если при сложении/вычитании возникает перенос/заём из старшего разряда. При сдвигах хранит значение выдвигаемого бита.
* A (*Auxiliary*) — флаг дополнительного переноса. Используется в операциях с упакованными двоично-десятичными числами.
* O (*Overflow*) — флаг переполнения. CF=1, если получен результат за пределами допустимого диапазона значений.

**Часть 5. Директивы объявления данных**

Практически любая программа кроме машинных команд содержит данные. Например, числа, текстовые строчки, идентификаторы, различные ресурсы и т.д. Данные могут быть как константами, не меняющими своё значение во время выполнения программы, так и переменными, в которых хранятся всякие промежуточные результаты.

Прежде всего нужно научиться объявлять данные в программе. Для этого в ассемблере существуют директивы объявления данных.

|  |  |  |
| --- | --- | --- |
| **Размер (в байтах)** | **Объявление** | **Резервирование** |
| 1 | db | rb |
| 2 | dw du | rw |
| 4 | dd | rd |
| 6 | dp df | rp rf |
| 8 | dq | rq |
| 10 | dt | rt |
| N | file |  |

Наиболее полезными для наших задач будут директивы *db*, *dw* и *dd*.

**Синтаксис объявления данных**

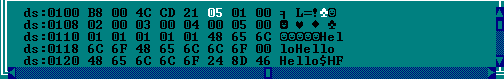
Объявлять данные очень просто — например, чтобы объявить байт cо значением 5 достаточно написать:

|  |
| --- |
| x **db** 5 |

где *x* — название нашей переменной или константы, *db* — директива объявления байта, а *5* — значение. С помощью названия в программе можно будет обращаться к ячейке памяти, содержащей наш байт. Вообще, название не обязательно и можно его не писать, если оно не требуется:

|  |
| --- |
| **db** 5 |

Если запустить программу в отладчике Turbo Debugger, то в окне дампа можно увидеть результат работы директивы *db*:



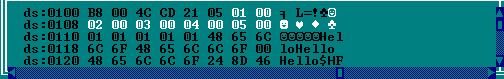
**Объявление последовательностей (массивов)**

Иногда в программе требуется объявить массив, то есть несколько переменных одинакового размера, расположенных в памяти друг за другом. Например, чтобы объявить массив из 5 двухбайтных чисел можно написать:

|  |
| --- |
| array1 **dw** 1,2,3,4,5 |

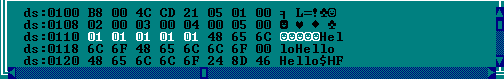
где *array1* — название массива, *1,2,3,4,5* — значения элементов. Вместо *array1* компилятор FASM будет подставлять в программу адрес начала массива, то есть адрес первого элемента.

Дамп памяти будет выглядеть следующим образом (обратите внимание, младший байт каждого слова расположен перед старшим):



Для объявления повторяющихся элементов можно использовать такую запись (объявляем массив из 5 байтов, равных 1):

|  |
| --- |
| array2 **db** 5 dup(1) |



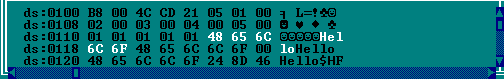
А ещё можно вот так объявить массив (догадайтесь сами, что тут получается):

|  |
| --- |
| array3 **dd** 4 dup(3,7,0) |

**Объявление строк**

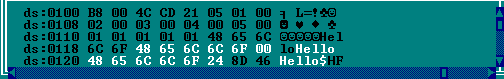
Строка представляет собой массив байтов-символов и записывается в одинарных кавычках:

|  |
| --- |
| str1 **db** 'Hello' |



Для обозначения конца строки используется специальный символ. Обычно это нулевой байт, но для функций DOS используется символ ’$’.

|  |
| --- |
| str2 **db** 'Hello',0 *;Обычно так*  str3 **db** 'Hello$' *;Для DOS* |



**Резервирование данных (точнее памяти для них)**

Можно объявлять переменные, не имеющие определённого начального значения. Такие переменные называются неинициализированными. Например, их можно использовать в программе для хранения временного или промежуточного значения. Фактически под переменную просто резервируется место в памяти. Объявлять такие переменные можно с помощью директив *db*, *dw*, *dd*, … и знака вопроса вместо значения.

|  |
| --- |
| x1 **db** ?  x2 **dw** ?,?,?  x3 **dd** 10 dup(?) |

Кроме того, FASM поддерживает специальные директивы резервирования данных. Число после директивы обозначает количество резервируемых элементов. То же самое можно объявить по другому:

|  |
| --- |
| x1 rb 1  x2 rw 3  x3 rd 10 |

С неинициализированными переменными следует быть внимательным. Не надо рассчитывать, что по умолчанию значение будет нулевым или ещё каким-то, иначе это может привести к ошибке.

**Директива *file***

*file* — это особая директива объявления данных, которая позволяет добавить в исполняемый файл последовательность байтов из внешнего файла. Иногда это может быть очень удобно. Например, если вы хотите добавить изображение в исполняемый файл (в виде данных), или большой кусок текста, или даже код из другого файла. Директива используется следующим образом:

|  |
| --- |
| data1 file 'data.bin' *;Добавить файл data.bin целиком.*  data2 file 'data.bin':20 *;Добавить байты из файла data.bin, начиная со смещения 20.*  data3 file 'data.bin':20,5 *;Добавить 5 байтов из файла data.bin, начиная со смещения 20.* |

**Часть 6. Hello, world!**

В этой части лекций напишем программу вывода сообщения «Hello, world!». Для начала необходимо с помощью директивы *db* объявить строку, содержащую сообщение «Hello, word!». Лучше сделать это в конце программы, за последней командой, иначе процессор может принять строку за код и попытаться её выполнить.

Для вывода строки используется системная функция DOS. Чтобы напечатать строку, нужно поместить 9 в регистр AH, а в регистр DX поместить адрес строки, которая должна заканчиваться символом ‘$’. Обращение к функциям DOS осуществляется с помощью команды *int 21h*. Вот код программы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **dx**,hello *;В DX адрес строки.*  **mov** **ah**,9 *;Номер функции DOS.*  **int** 21h *;Обращение к функции DOS.*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  hello **db** 'Hello, world!$' |

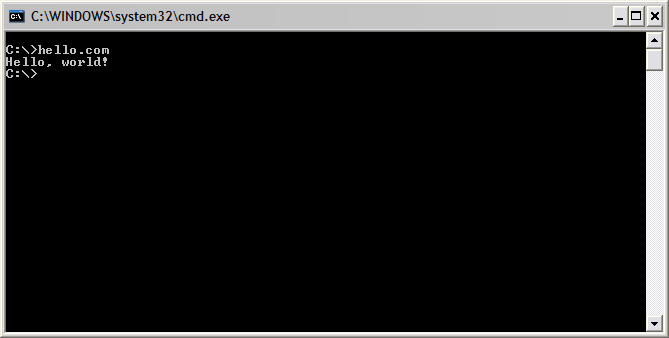
В четвёртой строке FASM подставит адрес строки вместо *hello*. Не трудно догадаться, что завершение программы — это тоже функция DOS с номером 4Ch. Перед её вызовом в регистр AL помещается код завершения программы (ноль соответствует успешному завершению). Можно объединить эти две операции и сразу поместить в AX значение 4C00h.

В учебном курсе мы не будем подробно описывать функции DOS, лишь кратко рассмотрим те функции, которые мы будем использовать.

Чтобы увидеть работу программы, надо запустить её из командной строки, иначе она печатает строку и сразу закрывается. Или можно написать простенький bat-файл для запуска:

|  |  |
| --- | --- |
| 1  2 | hello.com  **pause** |

Результат работы программы:



Если вы запустите программу в отладчике, то просмотреть выводимую строку можно, нажав Alt+F5 или выбрав в меню Turbo Debuger пункт *Window->User Screen*.

**Часть 7. Системы счисления**

На самом деле процессор работает только с двоичными числами, состоящими из единиц и нулей. В виде двоичных чисел хранятся и обрабатываются все данные и команды любой программы. Однако, двоичная запись чисел слишком громоздка и неудобна для человека, поэтому в программах на ассемблере используются и другие системы счисления: десятичная, шестнадцатеричная и восьмеричная.

Прежде всего разберёмся, в чём различие между системами счисления. Любое десятичное число можно представить в таком виде:

12310 = 1·102 + 2·101 + 3·100

Индекс внизу обозначает, что это десятичное число. Цифра каждого разряда умножается на 10 в степени, равной номеру разряда, если считать с нуля справа налево. В более общем виде:

abc*r* = a·*r*2 + b·*r*1 + c·*r*0,

где a, b и с — какие-то цифры, а r — основание системы счисления. Для десятичной системы *r* = 10, для двоичной — *r* = 2, для троичной *r* = 3 и т.д. Например, то же самое число в других системах:

4435 = 4·52 + 4·51 + 3·50 = 4·25 + 4·5 + 3·1 = 12310 (пятеричная система)

1738 = 1·82 + 7·81 + 3·80 = 1·64 + 7·8 + 3·1 = 12310 (восьмеричная система)

11110112 = 1·26 + 1·25 + 1·24 + 1·23 + 0·22 + 1·21 + 1·20 = 1·64 + 1·32 + 1·16 + 1·8 + 0·4 + 1·2 + 1·1 = 12310 (двоичная система)

**Шестнадцатеричная система**

В шестнадцатеричной системе для обозначения цифр больше 9 используются буквы A = 10, B = 11, C = 12, D = 13, E = 14, F = 15. Например:

C716 = 12·161 + 7·160 = 12·16 + 7·1 = 19910

Удобство шестнадцатеричной системы в том, что в неё очень легко можно переводить двоичные числа (и в обратную сторону тоже). Четыре разряда двоичного числа (тетрада) представляются одним разрядом шестнадцатеричного. Для перевода достаточно разбить число на группы по 4 бита и заменить каждую тетраду соответствующей шестнадцатеричной цифрой.

|  |  |
| --- | --- |
| **Двоичная тетрада** | **Шестнадцатеричная цифра** |
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

Для записи одного байта требуется всего 2 шестнадцатеричные цифры:

0101 10112 = 5B16

0110 00002 = 6016

1111 11112 = FF16

**Восьмеричная система**

Восьмеричная система также удобна для представления двоичных чисел, хотя она используется намного реже, чем шестнадцатеричная. Для быстрого перевода надо разбить двоичное число на группы по 3 разряда (триплеты или триады).

|  |  |
| --- | --- |
| **Двоичная триада** | **Восьмеричная цифра** |
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

Например: 001 110 1012 = 1658

**Синтаксис ассемблера FASM**

По умолчанию, число в программе воспринимается ассемблером как десятичное. Чтобы обозначить двоичное число, необходимо к нему в конце добавить символ *’b’*. Восьмеричное число обозначается аналогично с помощью символа *’o’*. Для записи шестнадцатеричного числа FASM поддерживает 3 формы записи:

* перед числом записываются символы *’0x’* (как в C/C++);
* перед числом записывается символ *’$’* (как в Pascal);
* после числа записывается символ *’h’*. Если шестнадцатеричное число начинается с буквы, необходимо добавить в начале ноль (иначе непонятно, число это или имя метки).

Этот синтаксис используется как при объявлении данных, так и в командах. Приведем примеры записи чисел во всех четырёх системах:

|  |
| --- |
| **mov** **ax**,537 *;Десятичная система*    **mov** **bl**,11010001b *;Двоичная система*    **mov** **ch**,57o *;Восьмеричная система*    **mov** **dl**,**$**C2 *;\*  **mov** **si**,0x013A *; \*  **mov** **ah**,18h *; / Шестнадцатеричная система*  **mov** **al**,0FFh *;/*    **mov** **al**,FFh *;Ошибка!* |

**Часть 8. Числа со знаком и без знака**

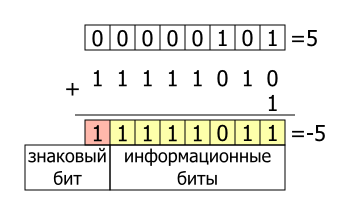
**Числа со знаком и дополнительный код**

Помимо того, что процессор работает с двоичными числами, эти числа могут быть со знаком или без знака. Если число без знака, то оно просто представляет собой результат перевода десятичного числа в двоичный вид. Все биты в таком числе являются *информационными* и оно может принимать только неотрицательные значения.

Для представления чисел со знаком используется специальное кодирование. Старший бит в этом случае обозначает знак числа. Если знаковый бит равен нулю, то число положительное, иначе — отрицательное. Понятно, что положительное число со знаком будет выглядеть точно так же, как и число без знака.

С отрицательными числами чуть сложнее. Исторически для представления отрицательных чисел в компьютерах использовались разные виды кодирования: прямой, обратный и дополнительный код. В настоящее время наиболее часто используется дополнительный код, в том числе и в процессорах x86.

Чтобы сделать из положительного числа отрицательное, необходимо проинвертировать все его биты (0 заменяем на 1, а 1 заменяем на 0) и затем к младшему разряду прибавить единицу. Например, представим -5в дополнительном коде:



В обратную сторону переводится точно также

**Синтаксис FASM**

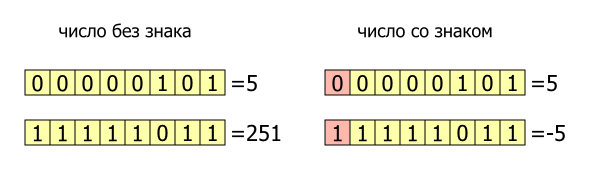
Для записи отрицательного числа в программе на ассемблере используется символ *‘-‘*, например:

|  |
| --- |
| x **db** -5 |

Это работает и с числами в других системах счисления, и даже с символами

|  |
| --- |
| y **db** -25h  z **db** -77o  k **db** -101b  s **db** -'a' |

Со знаковыми и беззнаковыми числами нужно быть внимательным, потому что только программист знает, какие числа используются в программе! И невнимательность может привести к ошибке. Один и тот же байт может интерпретироваться по-разному, в зависимости от того со знаком число или без. Например, числу со знаком -5 соответствует число без знака 251:



**Диапазоны значений чисел со знаком и без**

При программировании на ассемблере (как, впрочем, и на многих других языках) необходимо учитывать ещё один важный момент. А именно — ограничение диапазона представления чисел. Например, если размер беззнаковой переменной равен 1 байт, то она может принимать всего 256 различных значений. Это означает, что мы не сможем представить с её помощью число, больше 255 (111111112). Для такой же переменной со знаком максимальным значением будет 127 (011111112), а минимальным -128 (100000002). Аналогично определяется диапазон для 2- и 4-байтных переменных.

Кстати, так как процессор Intel 8086 был 16-битным и обрабатывал за одну команду 16-бит, то 16-битная переменная называется *слово (word)*, а 32-битная — *двойное слово (double word, dword)*. Эти названия сохранились в ассемблере даже для 32-битных процессоров (и в WIN32 API, например). И от них же происходят названия директив *dw (Define Word)* и *dd (Define Dword)*. Ну а *db* — это *Define Byte*.

Для наглядности рассмотрим таблицу диапазонов чисел:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размер переменной** | **Число без знака** | | **Число со знаком** | |
| **min** | **max** | **min** | **max** |
| байт | 00000000 | 11111111 | 10000000 | 01111111 |
| 0 | 255 | -128 | 127 |
| слово | 00000000 00000000 | 11111111 11111111 | 10000000 00000000 | 01111111 11111111 |
| 0 | 65 535 | -32 768 | 32 767 |
| двойное слово | 0000…0000 | 1111…1111 | 1000…0000 | 0111…1111 |
| 0 | 4 294 967 295 | -2 147 483 648 | 2 147 483 647 |
| и т.д. | … | … | … | … |

Если результат какой-то операции выйдет за пределы диапазона представления чисел, то случится *переполнение* и результат будет некорректным. (Например, при сложении двух положительных чисел, можно получить отрицательное число!) Поэтому нужно быть внимательным при программировании и предусмотреть обработку таких ситуаций, если они могут возникнуть.

**Часть 9. Сложение и вычитание**

Теперь перейдем к изучению команд процессора. Начнём с самых простых арифметических операций: сложения и вычитания.

**Сложение**

Для сложения двух чисел предназначена команда ADD. Она работает как с числами со знаком, так и с числами без знака (это особенность дополнительного кода).

Операнды должны иметь одинаковый размер (нельзя складывать 16- и 8-битное значение). Результат помещается на место первого операнда. В общем, эти правила справедливы для большинства команд.

После выполнения команды изменяются флаги, по которым можно определить характеристики результата:

* Флаг *CF* устанавливается, если при сложении произошёл перенос из старшего разряда. Для беззнаковых чисел это будет означать, что произошло переполнение и результат получился некорректным.
* Флаг *OF* обозначает переполнение для чисел со знаком.
* Флаг *SF* равен знаковому биту результата (естественно, для чисел со знаком, а для беззнаковых он равен старшему биту и особо смысла не имеет).
* Флаг *ZF* устанавливается, если результат равен 0.
* Флаг *PF* — признак чётности, равен 1, если результат содержит нечётное число единиц.

Примеры:

|  |
| --- |
| **add** **ax**,5 *;AX = AX + 5*  **add** **dx**,**cx** *;DX = DX + CX*  **add** **dx**,**cl** *;Ошибка: разный размер операндов.* |

**Вычитание**

Вычитание выполняется с помощью команды [SUB](http://asmworld.ru/spravochnik-komand/sub/). Результат также помещается на место первого операнда и опять же выставляются флаги. Разница в том, что происходит вычитание, а не сложение.

На самом деле вычитание в процессоре реализовано с помощью сложения. Процессор меняет знак второго операнда на противоположный, а затем складывает два числа. Если вам необходимо в программе поменять знак числа на противоположный, можно использовать команду [NEG](http://asmworld.ru/spravochnik-komand/neg/). У этой команды всего один операнд.

Примеры:

|  |
| --- |
| **sub** **si**,**dx** *;SI = SI - DX*  **neg** **ax** *;AX = -AX* |

**Инкремент и декремент**

Очень часто в программах используется операция прибавления или вычитания единицы. Прибавление единицы называется инкрементом, а вычитание — декрементом. Для этих операций существуют специальные команды процессора: [INC](http://asmworld.ru/spravochnik-komand/inc/) и [DEC](http://asmworld.ru/spravochnik-komand/dec/). Причем эти команды не изменяют значение флага *CF*.

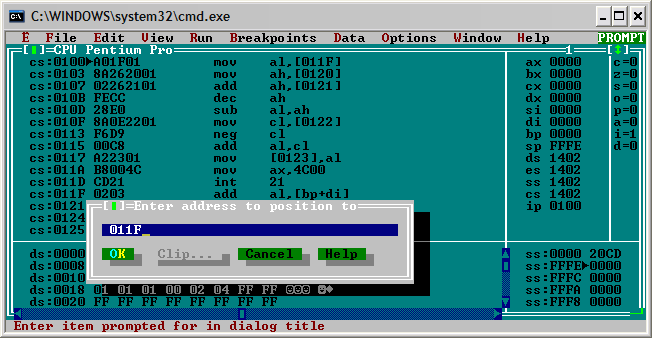
**Пример программы**

Чтобы всё стало совсем понятно, напишем небольшую программу. Требуется вычислить значение формулы: *e=a-(b+c-1)+(-d)*. Все числа являются 8-битными целыми со знаком. Объявим их после кода и введем какие-нибудь значения.

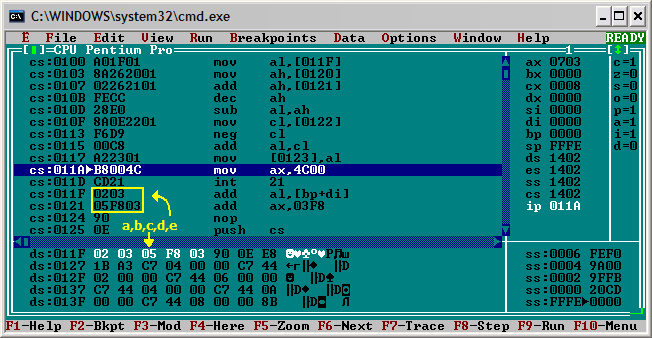
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **al**,[a] *;Загружаем значение a в AL*  **mov** **ah**,[b] *;Загружаем значение b в AH*  **add** **ah**,[c] *;AH = AH + c = b+c*  **dec** **ah** *;AH = AH - 1 = b+c-1*  **sub** **al**,**ah** *;AL = AL - AH = a-(b+c-1)*  **mov** **cl**,[d] *;CL = d*  **neg** **cl** *;CL = -CL = -d*  **add** **al**,**cl** *;AL = AL + CL = a-(b+c-1)+(-d)*  **mov** [e],**al** *;Сохраняем результат в e*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  a **db** 2  b **db** 3  c **db** 5  d **db** -8  e **db** ? |

Квадратные скобки означают, что операнд находится по адресу, указанному внутри этих скобок. Так как вместо имени переменной FASM подставляет её адрес, то такая запись позволяет прочитать или записать значение переменной.

Запустив программу в Turbo Debugger, можно посмотреть её выполнение по шагам. Значения переменных можно увидеть в окне дампа памяти. Для этого нужно кликнуть правой кнопкой в этом окне и выбрать в меню пункт *Goto…*. Переменные начинаются в памяти с адреса 011Fh (этот адрес в первой команде).



В этих байтах легко угадываются наши переменные:



**Упражнение**

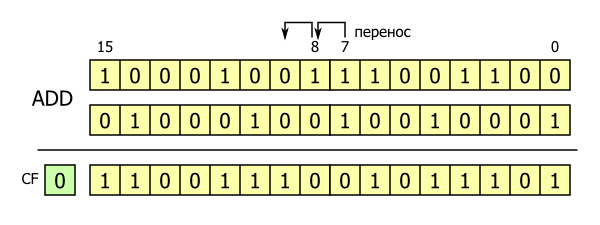
Напишите программу для вычисления формулы *k=m+1-(n-1-r)*. Все числа 16-битные целые со знаком. Запустите в отладчике и проверьте правильность вычисления.

**Часть 10. Сложение и вычитание с переносом**

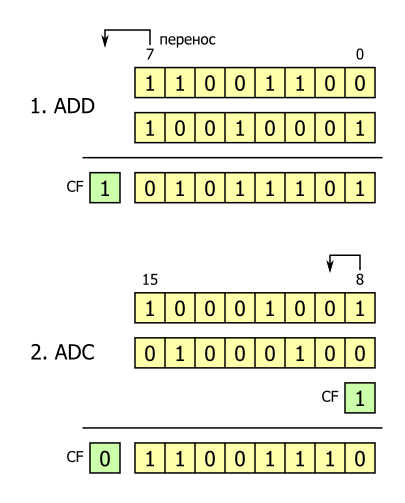
В системе команд процессоров x86 имеются специальные команды сложения и вычитания с учётом флага переноса (*CF*). Для сложения с учётом переноса предназначена команда ADC, а для вычитания — SBB. В общем, эти команды работают почти также, как ADD и SUB, единственное отличие в том, что к младшему разряду первого операнда прибавляется или вычитается дополнительно значение флага *CF*.

Зачем нужны такие команды? Они позволяют выполнять сложение и вычитание многобайтных целых чисел, длина которых больше, чем разрядность регистров процессора (в нашем случае 16 бит). Принцип программирования таких операций очень прост — длинные числа складываются (вычитаются) по частям. Младшие разряды складываются(вычитаются) с помощью обычных команд ADD и SUB, а затем последовательно складываются(вычитаются) более старшие части с помощью команд ADC и SBB. Так как эти команды учитывают перенос из старшего разряда, то мы можем быть уверены, что ни один бит не потеряется. Этот способ похож на сложение(вычитание) десятичных чисел в столбик.

На следующем рисунке показано сложение двух двоичных чисел командой ADD:



При сложении происходит перенос из 7-го разряда в 8-й, как раз на границе между байтами. Если мы будем складывать эти числа по частям командой ADD, то перенесённый бит потеряется и в результате мы получим ошибку. К счастью, перенос из старшего разряда всегда сохраняется в флаге CF. Чтобы прибавить этот перенесённый бит, достаточно применить команду ADC:



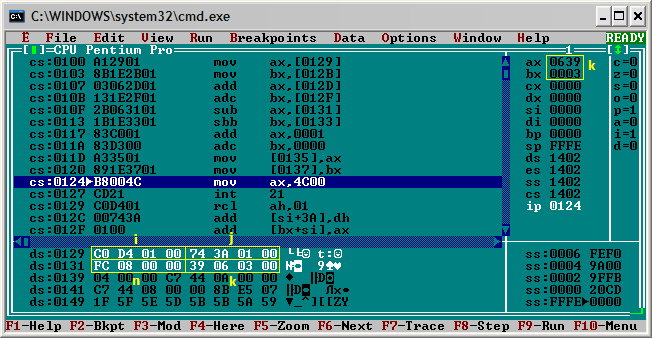
Аналогичная ситуация возникает с вычитанием чисел по частям. Чтобы было совсем понятно, приведу пример программы. Допустим, требуется вычислить значение формулы *k=i+j-n+1*, где переменные *k*, *i*, *j* и *n* являются 32-битными целыми числами без знака. Складывать и вычитать такие числа придётся в два этапа: сначала вычисления будут производиться с младшими словами операндов, а затем со старшими с учётом переноса.

Для прибавления единицы в данном примере нельзя использовать команду INC, так как она не влияет на флаг *CF* и мы можем получить ошибку в результате!

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **ax**,**word**[i] *;Загружаем младшую часть i в AX*  **mov** **bx**,**word**[i+2] *;Загружаем старшую часть i в BX*    **add** **ax**,**word**[j] *;Складываем младшие части i и j*  **adc** **bx**,**word**[j+2] *;Складываем старшие части i и j*    **sub** **ax**,**word**[n]  **sbb** **bx**,**word**[n+2] *;BX:AX = i+j-n*    **add** **ax**,1 *;Команда INC здесь не подходит!*  **adc** **bx**,0 *;BX:AX = i+j-n+1*    **mov** **word**[k],**ax** *;\*  **mov** **word**[k+2],**bx** *;/ Сохраняем результат в k*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  i **dd** 120000  j **dd** 80500  n **dd** 2300  k **dd** ? |

Запись *word[i]* означает, что мы переопределяем размер переменной (она объявлена как DWORD) и обращаемся к младшему слову. Старшее слово расположено в памяти после младшего, поэтому к адресу переменной надо прибавить 2, и соответствующая запись будет иметь вид *word[i+2]*.

Посмотреть работу программы можно в отладчике:



Обратите внимание, как хранятся переменные в памяти. В процессорах Intel младший байт всегда хранится по младшему адресу, поэтому получается, что в окне дампа значения надо читать справа налево. В регистрах же числа записываются в нормальном виде. Сравните, как выглядит одно и то же значение *k* в памяти и в регистрах (старшая часть находится в BX, а младшая — в AX).

Одно из преимуществ ассемблера в том, что на нём можно реализовать работу с собственными форматами чисел, например с очень длинными целыми. А в языках высокого уровня выбор всегда ограничен компилятором. Следующая программа складывает два 7-байтных значения.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **ax**,**word**[x]  **add** **ax**,**word**[y]  **mov** **word**[z],**ax**    **mov** **ax**,**word**[x+2]  **adc** **ax**,**word**[y+2]  **mov** **word**[z+2],**ax**    **mov** **ax**,**word**[x+4]  **adc** **ax**,**word**[y+4]  **mov** **word**[z+4],**ax**    **mov** **al**,**byte**[x+6]  **adc** **al**,**byte**[y+6]  **mov** **byte**[z+6],**al**    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  x **dd** 0xF1111111  **dw** 0xF111  **db** 0x11  y **dd** 0x22222222  **dw** 0x2222  **db** 0x22  z rb 7 |

Обращение к старшему байту записывается как *byte[x+6]*. Команда MOV не меняет состояние флагов, поэтому её можно ставить между командами сложения.

**Упражнение**

Напишите программу для вычисления формулы *d=b-1+a-c*. Все числа — 3-х байтные целые без знака. Проверьте работу программы в отладчике.

**Часть 11. Умножение и деление**

Умножение и деление выполняются по-разному для чисел со знаком и без, поэтому в системе команд процессора x86 есть отдельные команды умножения и деления для чисел со знаком и для чисел без знака.

**Умножение чисел без знака**

Для умножения чисел без знака предназначена команда MUL. У этой команды только один операнд — второй множитель, который должен находиться в регистре или в памяти. Местоположение первого множителя и результата задаётся неявно и зависит от размера операнда:

|  |  |  |
| --- | --- | --- |
| **Размер операнда** | **Множитель** | **Результат** |
| Байт | AL | AX |
| Слово | AX | DX:AX |

Отличие умножения от сложения и вычитания в том, что разрядность результата получается в 2 раза больше, чем разрядность сомножителей. Также и в десятичной системе — например, умножая двухзначное число на двухзначное, мы можем получить в результате максимум четырёхзначное. Запись «DX:AX» означает, что старшее слово результата будет находиться в DX, а младшее — в AX. Примеры:

|  |
| --- |
| **mul** **bl** *;AX = AL \* BL*  **mul** **ax** *;DX:AX = AX \* AX* |

Если старшая часть результата равна нулю, то флаги *CF* и *ОF* будут иметь нулевое значение. В этом случае старшую часть результата можно отбросить. Это свойство можно использовать в программе, если результат должен быть такого же размера, как множители.

**Умножение чисел со знаком**

Для умножения чисел со знаком предназначена команда IMUL. Эта команда имеет три формы, различающиеся количеством операндов:

* *С одним операндом* — форма, аналогичная команде MUL. В качестве операнда указывается множитель. Местоположение другого множителя и результата определяется по таблице.
* *С двумя операндами* — указываются два множителя. Результат записывается на место первого множителя. Старшая часть результата в этом случае игнорируется. Кстати, эта форма команды не работает с операндами размером 1 байт.
* *С тремя операндами* — указывается положение результата, первого и второго множителя. Второй множитель должен быть непосредственным значением. Результат имеет такой же размер, как первый множитель, старшая часть результата игнорируется. Это форма тоже не работает с однобайтными множителями.

Примеры:

|  |
| --- |
| **imul** **cl** *;AX = AL \* CL*  **imul** **si** *;DX:AX = AX \* SI*  **imul** **bx**,**ax** *;BX = BX \* AX*  **imul** **cx**,-5 *;CX = CX \* (-5)*  **imul** **dx**,**bx**,134h *;DX = BX \* 134h* |

*CF* = *OF* = 0, если произведение помещается в младшей половине результата, иначе *CF* = *OF* = 1. Для второй и третьей формы команды *CF* = *OF* = 1 означает, что произошло переполнение.

**Деление чисел без знака**

Деление целых двоичных чисел — это всегда деление с остатком! По аналогии с умножением, размер делителя, частного и остатка должен быть в 2 раза меньше размера делимого. Деление чисел без знака осуществляется с помощью команды DIV. У этой команды один операнд — делитель, который должен находиться в регистре или в памяти. Местоположение делимого, частного и остатка задаётся неявно и зависит от размера операнда:

|  |  |  |  |
| --- | --- | --- | --- |
| **Размер операнда (делителя)** | **Делимое** | **Частное** | **Остаток** |
| Байт | AX | AL | AH |
| Слово | DX:AX | AX | DX |

При выполнении команды DIV может возникнуть *прерывание* (старайтесь избегать таких случаев):

* если делитель равен нулю;
* если частное не помещается в отведённую под него разрядную сетку (например, если при делении слова на байт частное больше 255).

Примеры:

|  |
| --- |
| **div** **cl** *;AL = AX / CL, остаток в AH*  **div** **di** *;AX = DX:AX / DI, остаток в DX* |

**Деление чисел со знаком**

Для деления чисел со знаком предназначена команда IDIV. Единственным операндом является делитель. Местоположение делимого и частного определяется также, как для команды DIV. Эта команда тоже генерирует прерывание при делении на ноль или слишком большом частном.

**Пример программы**

Допустим, в программе требуется вычислять координату какого-то движущегося объекта по формуле:

*x = x0 + v0t + at2/2*

Все числа в правой части — 8-битные целые без знака, а *x* — 16-битное целое и тоже без знака. Здесь нужно внимательно следить за размерами операндов.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **al**,[v0] *;AL = v0*  **mov** **cl**,[t] *;CL = t*  **mul** **cl** *;AX = AL\*CL = v0\*t*  **mov** **bx**,**ax** *;BX = AX = v0\*t*    **mov** **al**,[a] *;AL = a*  **mul** **cl** *;AX = AL\*CL = a\*t*  **mov** **ch**,0 *;Преобразуем t в слово в регистре CX*  **mul** **cx** *;DX:AX = AX\*CX = a\*(t^2)*  **mov** **cl**,2 *;CL = 2 = CX, так как CH = 0*  **div** **cx** *;AX = DX:AX/2 = a\*(t^2)/2*    **add** **ax**,**bx** *;AX = AX+BX = v0\*t + a\*(t^2)/2*  **add** **al**,[x0] *;\*  **adc** **ah**,**ch** *;/ AX = AX+x0 = x0 + v0\*t + a\*(t^2)/2*    **mov** [x],**ax** *;Сохраняем результат в x*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  x0 **db** 188  v0 **db** 7  a **db** 3  t **db** 25  x **dw** ? |

В 7-й строке промежуточный результат сохраняется в bx. В 11-й строке происходит преобразование байта в слово, путём добавления нулевой старшей части. Такой метод подходит для чисел без знака, но приведёт к ошибке для чисел со знаком (в случае отрицательного числа). Прибавление *x0* происходит в два этапа (строки 17 и 18) с учётом переноса, так как мы складываем слово и байт.

**Упражнение**

Напишите программу для вычисления формулы *z = (x·y) / (x + y)*. Все числа 16-битные целые со знаком.

**Сложное упражнение**

Напишите программу для умножения двух 32-битных целых без знака. В результате должно получиться 64-битное целое без знака. Подсказка: используйте комбинацию умножения по частям и сложения, как в способе умножения столбиком.

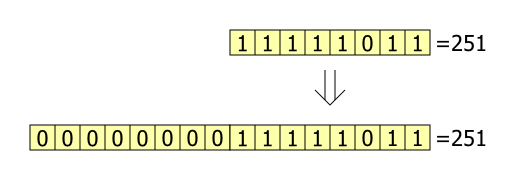
**Часть 12. Преобразование типов**

Очень часто в программах приходится выполнять действия с числами разного размера, например складывать или умножать байт и слово. Напрямую процессор не умеет выполнять такие операции, поэтому в этом случае необходимо выполнять преобразование типов. Сложность представляет преобразование меньших типов в большие (байта в слово, слова в двойное слово и т.д.)

Преобразовать больший тип в меньший гораздо проще — достаточно отбросить старшую часть. Но такое преобразование небезопасно и может привести к ошибке. Например, если значение слова без знака больше 255, то сделав из него байт, вы получите некорректное значение. Будьте внимательны, если вы используете данные преобразования в программе.

**Преобразование типов без знака**

Преобразование типов выполняется по-разному для чисел со знаком и без. Для преобразования чисел без знака необходимо просто заполнить все старшие биты нулями. Например, так будет выглядеть преобразование байта в слово:



Такое преобразование можно выполнить с помощью обычной команды MOV (*x* объявлен как байт):

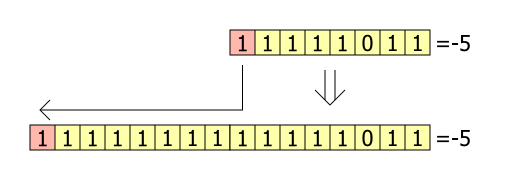
|  |
| --- |
| **mov** **bl**,[x]  **mov** **bh**,0 *;BX = x*  *;Или вот так:*  **mov** **bl**,[x]  **sub** **bh**,**bh** *;BX = x* |

Но кроме того в системе команд процессора существует специальная команда — MOVZX (копирование с нулевым расширением). Первый операнд команды имеет размер 16 бит (слово), а второй — 8 бит (байт). Тот же результат можно получить так:

|  |
| --- |
| **movzx** **bx**,[x] *;BX = x* |

**Преобразование типов со знаком**

Для чисел со знаком всё немного сложнее. Если мы просто заполним старшую часть нулями, то результат будет положительным, а это не всегда верно. Поэтому преобразование выполняется путём копирования знакового бита на всю старшую часть. То есть для положительного числа со знаком старшая часть будет заполняться нулями, а для отрицательного — единицами:



Для такого преобразования предназначена команда MOVSX (копирование со знаковым расширением). Первый операнд — слово, второй операнд — байт. Например (*y* объявлен как байт):

|  |
| --- |
| **movsx** **cx**,[y] *;CX = y* |

Существуют ещё две команды для преобразования типов со знаком: CBW (*Convert Byte to Word* — преобразовать байт в слово) и CWD (*Convert Word to Double word* — преобразовать слово в двойное слово). У этих команд нет явных операндов. Команда CBW преобразует байт, находящийся в регистре AL, в слово в регистре AX. Команда CWD преобразует слово, находящееся в регистре AX, в двойное слово в регистрах DX:AX. Эти команды удобно использовать вместе с командами умножения и деления. Например:

|  |
| --- |
| **mov** **al**,[y]  **cbw** *;AX = y*  **cwd** *;DX:AX = y* |

**Пример программы**

Допустим, требуется вычислить значение формулы *x = (a + b) / c*. Все числа со знаком. Размер *x* — двойное слово, размер *a* — байт, размер *b* и *c* — слово.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **movsx** **ax**,[a] *;AX = a*  **add** **ax**,[b] *;AX = a+b*  **cwd** *;DX:AX = a+b*  **idiv** [c] *;AX = (a+b)/c, в DX остаток*  **cwd** *;DX:AX = (a+b)/c*  **mov** **word**[x],**ax** *;\*  **mov** **word**[x+2],**dx** *;/ x = DX:AX*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  a **db** -55  b **dw** -3145  c **dw** 100  x **dd** ? |

**Упражнение**

Напишите программу для вычисления формулы z = (x·y) / (x + y). Все числа со знаком. Размер *x* — байт, размер *y* — слово, размер *z* — двойное слово. Проверьте работу программы в отладчике.

**Часть 13. Циклы и команда LOOP**

До этой части все наши программы выполнялись последовательно — в них не было ветвлений и переходов. Далее мы научимся делать простейшие циклы. Циклом называется повторяющееся выполнение последовательности команд. Но для начала нужно научиться объявлять метки.

**Синтаксис объявления меток**

Метка представляет собой символическое имя, вместо которого компилятор подставляет адрес. В программе на ассемблере можно присвоить имя любому адресу в коде или данных. Обычно метки используются для организации переходов, циклов или каких-то манипуляций с данными. По сути имена переменных, объявленных с помощью директив объявления данных, тоже являются метками. Но с ними компилятор дополнительно связывает размер переменной. Метка объявляется очень просто: достаточно в начале строки написать имя и поставить двоеточие. Например:

|  |
| --- |
| m1: **mov** **ax**,4C00h  **int** 21h |

Теперь вместо имени *m1* компилятор везде будет подставлять адрес комады *mov ax,4C00h*. Можно объявлять метку на пустой строке перед командой:

|  |
| --- |
| exit\_app:  **mov** **ax**,4C00h  **int** 21h |

Имя метки может состоять из латинских букв, цифр и символов подчёркивания, но должно начинаться с буквы. Имя метки должно быть уникальным. В качестве имени метки нельзя использовать директивы и ключевые слова компилятора, названия команд и регистров (в этом случае FASM покажет сообщение об ошибке). FASM различает регистр символов в именах меток. Можно также объявлять несколько меток на один адрес. Например:

|  |
| --- |
| no\_error:  exit\_app:  m1: **mov** **ax**,4C00h |

Подробнее о синтаксисе объявления меток рассказывается в части 27.

**Команда LOOP**

Для организации цикла предназначена команда LOOP. У этой команды один операнд — имя метки, на которую осуществляется переход. В качестве счётчика цикла используется регистр CX. Команда LOOP выполняет декремент CX, а затем проверяет его значение. Если содержимое CX не равно нулю, то осуществляется переход на метку, иначе управление переходит к следующей после LOOP команде.

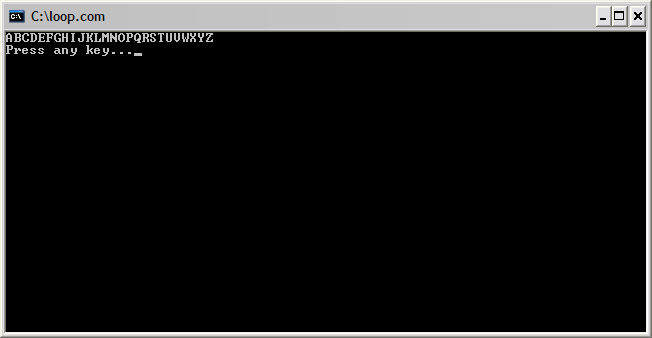
Содержимое CX интерпретируется командой как число без знака. В CX нужно помещать число, равное требуемому количеству повторений цикла. Понятно, что максимально может быть 65535 повторений. Ещё одно ограничение связано с дальность перехода. Метка должна находиться в диапазоне -127…+128 байт от команды LOOP (если это не так, FASM сообщит об ошибке).

**Пример цикла**

В качестве примера рассмотрим простую программу, которая будет печатать все буквы английского алфавита. ASCII-коды этих символов расположены последовательно, поэтому можно выводить их в цикле. Для вывода символа на экран используется функция DOS *02h* (выводимый байт должен находиться в регистре DL).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **ah**,02h *;Для вызова функции DOS 02h - вывод символа*  **mov** **dl**,'A' *;Первый выводимый символ*  **mov** **cx**,26 *;Счётчик повторений цикла*  metka:  **int** 21h *;Обращение к функции DOS*  **inc** **dl** *;Следующий символ*  **loop** metka *;Команда цикла*    **mov** **ah**,09h *;Функция DOS 09h - вывод строки*  **mov** **dx**,press *;В DX адрес строки*  **int** 21h *;Обращение к функции DOS*    **mov** **ah**,08h *;Функция DOS 08h - ввод символа без эха*  **int** 21h *;Обращение к функции DOS*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  press:  **db** 13,10,'Press any key...$' |

Команды *«int 21h»* и *«inc dl»* (строки 8 и 9) будут выполняться в цикле 26 раз. Для того, чтобы программа не закрылась сразу, используется функция DOS *08h* — ввод символа с клавиатуры без эха, то есть вводимый символ не отображается. Перед этим выводится предложение нажать любую кнопку (но *Reset* лучше не нажимать). Для примера адрес строки объявлен с помощью метки. Символы с кодами 13 и 10 обозначают переход на следующую строку (символ 13(0Dh) называется CR — *Carriage Return* — возврат каретки, а символ 10(0Ah) LF — *Line Feed* — перевод строки . Эти символы унаследованы со времён телетайпов, когда текст печатался, как на печатной машинке). Так выглядит результат работы программы:



**Вложенные циклы**

Иногда требуется организовать вложенный цикл, то есть цикл внутри другого цикла. В этом случае необходимо сохранить значение CX перед началом вложенного цикла и восстановить после его завершения (перед командой LOOP внешнего цикла). Сохранить значение можно в другой регистр, во временную переменную или в стек. Следующая программа выводит все доступные ASCII-символы в виде таблицы 16×16. Значение счётчика внешнего цикла сохраняется в регистре BX.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **ah**,02h *;Для вызова функции DOS 02h - вывод символа*  **sub** **dl**,**dl** *;Первый выводимый символ*  **mov** **cx**,16 *;Счётчик внешнего цикла (по строкам)*  lp1:  **mov** **bx**,**cx** *;Сохраняем счётчик в BX*  **mov** **cx**,16 *;Счётчик внутреннего цикла (по столбцам)*  lp2:  **int** 21h *;Обращение к функции DOS*  **inc** **dl** *;Следующий символ*  **loop** lp2 *;Команда внутреннего цикла*    **mov** **dh**,**dl** *;Сохраняем значение DL в DH*  **mov** **dl**,13 *;\*  **int** 21h *; \*  **mov** **dl**,10 *; / Переход на следующую строку*  **int** 21h *;/*  **mov** **dl**,**dh** *;Восстанавливаем значение DL*    **mov** **cx**,**bx** *;Восстанавливаем значение счётчика*  **loop** lp1 *;Команда внешнего цикла*    **mov** **ah**,09h *;Функция DOS 09h - вывод строки*  **mov** **dx**,press *;В DX адрес строки*  **int** 21h *;Обращение к функции DOS*    **mov** **ah**,08h *;Функция DOS 08h - ввод символа без эха*  **int** 21h *;Обращение к функции DOS*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  press **db** 13,10,'Press any key...$' |

Результат работы программы:



**Упражнение**

Напишите программу для вычисления степени числа 3 по формуле *a = 3n*. Число *a* — 16-битное целое без знака, число *n* — 8-битное целое без знака (используйте *n*<11, чтобы избежать переполнения). Проверьте работу программы в отладчике (нажимайте F7 на команде LOOP, чтобы осуществить переход).

**Часть 14. Режимы адресации**

Режимы адресации — это различные способы указания местоположения операндов. До этой части в учебном курсе использовались только простые режимы адресации: операнды чаще всего находились в регистрах или в переменных в памяти. Но в процессоре Intel 8086 существуют также более сложные режимы, которые позволяют организовать работу с массивами, структурами, локальными переменными и указателями. В этой части мы рассмотрим возможные режимы адресации, приведем примеры их использования.

**1. Неявная адресация**

Местоположение операнда фиксировано и определяется кодом операции. Примеры:

|  |
| --- |
| **cbw**  **mul** **al** |

Команда CBW всегда работает с регистрами AX и AL, а у команды MUL фиксировано положение первого множителя и результата. Такой режим адресации делает машинную команду короткой, так как в ней отсутствует указание одного или нескольких операндов.

**2. Непосредственная адресация**

При непосредственной адресации значение операнда является частью машинной команды. Понятно, что в этом случае операнд представляет собой константу. Примеры:

|  |
| --- |
| **mov** **al**,5  **add** **bx**,1234h  **mov** **dx**,a |

Обратите внимание, что в третьей строке в DX помещается *адрес* метки или переменной *a*, а вовсе не значение по этому адресу. Это особенность синтаксиса FASM. По сути адрес метки тоже является числовой константой.

**3. Абсолютная прямая адресация**

В машинной команде содержится адрес операнда, находящегося в памяти. Пример:

|  |
| --- |
| **mov** **dx**,[a] |

Вот тут уже в DX помещается значение из памяти по адресу *a*. Сравните с предыдущим пунктом. Квадратные скобки обозначают обращение по адресу, указанному внутри этих скобок.

**4. Относительная прямая адресация**

Этот режим используется в командах передачи управления. В машинной команде содержится смещение, которое прибавляется к значению указателя команд IP. То есть указывается не сам адрес перехода, а на сколько байтов вперёд или назад надо перейти. Пример:

|  |
| --- |
| metka:  ...  **loop** metka |

У такого режима адресации два преимущества. Во-первых, машинная команда становится короче, так она содержит не полный адрес, а только смещение. Во-вторых, такой код не зависит от адреса, по которому он размещается в памяти.

**5. Регистровая адресация**

Операнд находится в регистре. Пример:

|  |
| --- |
| **add** **ax**,**bx** |

**6. Косвенная регистровая (базовая) адресация**

Адрес операнда находится в одном из регистров BX, SI или DI. Примеры:

|  |
| --- |
| **add** **ax**,[**bx**]  **mov** **dl**,[**si**] |

Размер операнда в памяти здесь определяется размером первого операнда. Так как AX — 16-разрядный регистр, то из памяти берётся слово по адресу в BX. Так как DL — 8-разрядный регистр, то из памяти берётся байт по адресу в SI. Это правило верно и для других режимов адресации.

**7. Косвенная регистровая (базовая) адресация со смещением**

Адрес операнда вычисляется как сумма содержимого регистра BX, BP, SI или DI и 8- или 16-разрядного смещения. Примеры:

|  |
| --- |
| **add** **ax**,[**bx**+2]  **mov** **dx**,[array1+**si**] |

В качестве смещения можно указать число или адрес метки. О размере смещения не беспокойтесь — компилятор сам его определяет и использует нужный формат машинной команды.

**8. Косвенная базовая индексная адресация**

Адрес операнда вычисляется как сумма содержимого одного из базовых регистров BX или BP и одного из индексных регистров SI или DI. Примеры:

|  |
| --- |
| **mov** **ax**,[**bp**+**si**]  **add** **ax**,[**bx**+**di**] |

Например, в одном из регистров может находиться адрес начала массива в памяти, а в другом — смещение какого-то элемента относительно начала.

**9. Косвенная базовая индексная адресация со смещением**

Адрес операнда вычисляется как сумма содержимого одного из базовых регистров BX или BP, одного из индексных регистров SI или DI и 8- или 16-разрядного смещения. Примеры:

|  |
| --- |
| **mov** **al**,[**bp**+**di**+5]  **mov** **bl**,[array2+**bx**+**si**] |

**Пример программы**

Допустим, имеется массив 32-битных целых чисел со знаком. Количество элементов массива хранится в 16-битной переменной без знака. Требуется вычислить среднее арифметическое элементов массива и сохранить его в 32-битной переменной со знаком. В примере намеренно использованы разные режимы адресации, хотя тоже самое можно написать проще.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **sub** **ax**,**ax** *;AX = 0*  **cwd** *;DX = 0*  **mov** **si**,**ax** *;SI = 0 - смещение элемента от начала массива*  **mov** **bx**,array *;Помещаем в BX адрес начала массива*  **mov** **di**,n *;Помещаем в DI адрес n*  **mov** **cx**,[**di**] *;CX = n*  lp1:  **add** **ax**,[**bx**+**si**] *;Прибавление младшего слова*  **adc** **dx**,[**bx**+**si**+2] *;Прибавление старшего слова*  **add** **si**,4 *;Увеличиваем смещение в SI на 4*  **loop** lp1 *;Команда цикла*    **idiv** **word**[**di**] *;Делим сумму на количество элемнтов*  **cwd** *;DX:AX = AX*  **mov** **word**[m],**ax** *;\ Сохраняем*  **mov** **word**[m+2],**dx** *;/ результат*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;-------------------------------------------------------*  n **dw** 10  array **dd** 10500,-7500,-15000,10000,-8000  **dd** 6500,11500,-5000,10500,-20000  m **dd** ? |

**Упражнение**

Объявите в программе два массива 16-битных целых со знаком. Количество элементов массивов должно быть одинаковым и храниться в 8-битной переменной без знака. Требуется из последнего элемента второго массива вычесть первый элемент первого, из предпоследнего — вычесть второй элемент и т.д.

**Часть 15. Логические операции**

Логические операции выполняются поразрядно, то есть отдельно для каждого бита операндов. В результате выполнения изменяются флаги. В программах эти операции часто используются для сброса, установки или инверсии отдельных битов двоичных чисел.

**Логическое И**

Если оба бита равны 1, то результат равен 1, иначе результат равен 0.

|  |  |  |
| --- | --- | --- |
| **AND** | **0** | **1** |
| **0** | 0 | 0 |
| **1** | 0 | 1 |

Для выполнения операции логического И предназначена команда AND. У этой команды 2 операнда, результат помещается на место первого операнда. Часто эта команда используется для обнуления определённых битов числа. При этом второй операнд называют *маской*. Обнуляются те биты операнда, которые в маске равны 0, значения остальных битов сохраняются. Примеры:

|  |
| --- |
| **and** **ax**,**bx** *;AX = AX & BX*  **and** **cl**,11111110b *;Обнуление младшего бита CL*  **and** **dl**,00001111b *;Обнуление старшей тетрады DL* |

Ещё одно использование этой команды — быстрое вычисление остатка от деления на степень 2. Например, так можно вычислить остаток от деления на 8:

|  |
| --- |
| **and** **ax**,111b *;AX = остаток от деления AX на 8* |

**Логическое ИЛИ**

Если хотя бы один из битов равен 1, то результат равен 1, иначе результат равен 0.

|  |  |  |
| --- | --- | --- |
| **OR** | **0** | **1** |
| **0** | 0 | 1 |
| **1** | 1 | 1 |

Логическое ИЛИ вычисляется с помощью команды OR. У этой команды тоже 2 операнда, и результат помещается на место первого. Часто это команда используется для установки в 1 определённых битов числа. Если бит маски равен 1, то бит результата будет равен 1, остальные биты сохранят свои значения. Примеры:

|  |
| --- |
| **or** **al**,**dl** *;AL = AL | DL*  **or** **bl**,10000000b *;Установить знаковый бит BL*  **or** **cl**,00100101b *;Включить биты 0,2,5 CL* |

**Логическое НЕ (инверсия)**

Каждый бит операнда меняет своё значение на противоположное (0 → 1, 1 → 0). Операция выполняется с помощью команды NOT. У этой команды только один операнд. Результат помещается на место операнда. Эта команда не изменяет значения флагов. Пример:

|  |
| --- |
| **not** **byte**[**bx**] *;Инверсия байта по адресу в BX* |

**Логическое исключающее ИЛИ (сумма по модулю два)**

Если биты имеют одинаковое значение, то результат равен 0, иначе результат равен 1.

|  |  |  |
| --- | --- | --- |
| **XOR** | **0** | **1** |
| **0** | 0 | 1 |
| **1** | 1 | 0 |

Исключающим ИЛИ эта операция называется потому, что результат равен 1, если один бит равен 1 или другой равен 1, а случай, когда оба равны 1, исключается. Ещё эта операция напоминает сложение, но в пределах одного бита, без переноса. 1+1=10, но перенос в другой разряд игнорируется и получается 0, отсюда название «сумма по модулю 2». Для выполнения этой операции предназначена команда XOR. У команды два операнда, результат помещается на место первого. Команду можно использовать для инверсии определённых битов операнда. Инвертируются те биты, которые в маске равны 1, остальные сохраняют своё значение. Примеры:

|  |
| --- |
| **xor** **si**,**di** *;SI = SI ^ DI*  **xor** **al**,11110000b *;Инверсия старшей тетрады AL*  **xor** **bp**,8000h *;Инверсия знакового бита BP* |

Обозначение операции в комментарии к первой строке используется во многих языках высокого уровня (например C, C++, Java и т.д.). Часто XOR используют для обнуления регистров. Если операнды равны, то результат операции всегда равен 0. Такой способ обнуления работает быстрее и, в отличие от команды MOV, не содержит непосредственного операнда, поэтому команда получается короче (и не содержит нулевых байтов, что особенно нравится хакерам):

|  |
| --- |
| **mov** **bx**,0 *;Эта команда занимает 3 байта*  **xor** **bx**,**bx** *;А эта - всего 2* |

**Пример программы**

Допустим, у нас есть массив байтов. Размер массива хранится в байте без знака. Требуется в каждом байте сбросить 1-й и 5-й биты, установить 0-й и 3-й биты, инвертировать 7-й бит. А затем ещё инвертировать целиком последний байт массива.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **bx**,array *;BX = адрес массива*  **movzx** **cx**,[length] *;CX = длина массива*    **mov** **di**,**cx**  **dec** **di**  **add** **di**,**bx** *;DI = адрес последнего элемента*  m1:  **mov** **al**,[**bx**] *;AL = очередной элемент массива*  **and** **al**,11011101b *;Сбрасываем 1-й и 5-й биты*  **or** **al**,00001001b *;Устанавливаем 0-й и 3-й биты*  **xor** **al**,10000000b *;Инвертируем 7-й бит*  **mov** [**bx**],**al** *;Сохраняем обработанный элемент*  **inc** **bx** *;В BX - адрес следующего элемента*  **loop** m1 *;Команда цикла*    **not** **byte**[**di**] *;Инвертируем последний байт массива*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;----------------------------------------------------------*  length **db** 10  array **db** 1,5,3,88,128,97,253,192,138,0 |

**Упражнение**

Объявите переменную *x* как двойное слово с каким-то значением. Инвертируйте 7-й, 15-й и 31-й бит. Обнулите младший байт переменной. Присвойте единичное значение битам 11-14 и 28-30. Результат сохраните в переменной *y* (естественно, она тоже должна быть объявлена как двойное слово). Инвертируйте значение *x*.

**Часть 16. Условные и безусловные переходы**

В этой части научимся программировать условные и безусловные переходы. Вообще, трудно представить себе программу без проверки условий и переходов. С их помощью в программе реализуются различные управляющие конструкции, ветвления и даже циклы.

**Безусловные переходы**

Безусловный переход — это переход, который выполняется всегда. Безусловный переход осуществляется с помощью команды JMP. У этой команды один операнд, который может быть непосредственным адресом (меткой), регистром или ячейкой памяти, содержащей адрес. Существуют также «дальние» переходы — между сегментами, однако здесь мы их рассматривать не будем. Примеры безусловных переходов:

|  |
| --- |
| **jmp** metka *;Переход на метку*  **jmp** **bx** *;Переход по адресу в BX*  **jmp** **word**[**bx**] *;Переход по адресу, содержащемуся в памяти по адресу в BX* |

**Условные переходы**

Условный переход осуществляется, если выполняется определённое условие, заданное флагами процессора (кроме одной команды, которая проверяет CX на равенство нулю). Состояние флагов изменяется после выполнения арифметических, логических и некоторых других команд. Если условие не выполняется, то управление переходит к следующей команде.

Существует много команд для различных условных переходов. Также для некоторых команд есть синонимы (например, JZ и JE — это одно и то же). Для наглядности все команды условных переходов приведены в таблице:

|  |  |  |
| --- | --- | --- |
| **Команда** | **Переход, если** | **Условие перехода** |
| JZ/JE | нуль или равно | ZF=1 |
| JNZ/JNE | не нуль или не равно | ZF=0 |
| JC/JNAE/JB | есть переполнение/не выше и не равно/ниже | CF=1 |
| JNC/JAE/JNB | нет переполнения/выше или равно/не ниже | CF=0 |
| JP | число единичных бит чётное | PF=1 |
| JNP | число единичных бит нечётное | PF=0 |
| JS | знак равен 1 | SF=1 |
| JNS | знак равен 0 | SF=0 |
| JO | есть переполнение | OF=1 |
| JNO | нет переполнения | OF=0 |
| JA/JNBE | выше/не ниже и не равно | CF=0 и ZF=0 |
| JNA/JBE | не выше/ниже или равно | CF=1 или ZF=1 |
| JG/JNLE | больше/не меньше и не равно | ZF=0 и SF=OF |
| JGE/JNL | больше или равно/не меньше | SF=OF |
| JL/JNGE | меньше/не больше и не равно | SF≠OF |
| JLE/JNG | меньше или равно/не больше | ZF=1 или SF≠OF |
| JCXZ | содержимое CX равно нулю | CX=0 |

У всех этих команд один операнд — имя метки для перехода. Обратите внимание, что некоторые команды применяются для беззнаковых чисел, а другие — для чисел со знаком. Сравнения «выше» и «ниже» относятся к беззнаковым числам, а «больше» и «меньше» — к числам со знаком. Для беззнаковых чисел признаком переполнения будет флаг CF, а соответствующими командами перехода JC и JNC. Для чисел со знаком о переполнении можно судить по состоянию флага OF, поэтому им соответствуют команды перехода JO и JNO. Команды переходов не изменяют значения флагов.

В качестве примера приведем небольшую программу для сложения двух чисел со знаком с проверкой переполнения. В случае переполнения будет выводиться сообщение об ошибке. Вы можете поменять значения объявленных переменных, чтобы переполнение возникало или не возникало при их сложении, и посмотреть, что будет выводить программа.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **al**,[x] *;AL = x*  **add** **al**,[y] *;AL = x + y*  **jo** error *;Переход, если переполнение*  **mov** **ah**,09h *;\*  **mov** **dx**,ok\_msg *; > Вывод строки 'OK'*  **int** 21h *;/*  exit:  **mov** **ah**,09h *;\*  **mov** **dx**,pak *; > Вывод строки 'Press any key...'*  **int** 21h *;/*    **mov** **ah**,08h *;\*  **int** 21h *;/ Ввод символа*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  error:  **mov** **ah**,09h *;\*  **mov** **dx**,err\_msg *; > Вывод сообщения об ошибке*  **int** 21h *;/*  **jmp** exit *;Переход на метку exit*  *;----------------------------------------------------------*  x **db** -89  y **db** -55  err\_msg **db** 'Error: overflow detected.',13,10,'$'  ok\_msg **db** 'OK',13,10,'$'  pak **db** 'Press any key...$' |

**Команды CMP и TEST**

Часто для формирования условий переходов используются команды CMP и TEST. Команда CMP предназначена для сравнения чисел. Она выполняется аналогично команде SUB: из первого операнда вычитается второй, но результат не записывается на место первого операнда, изменяются только значения флагов. Например:

|  |
| --- |
| **cmp** **al**,5 *;Сравнение AL и 5*  **jl** c1 *;Переход, если AL < 5 (числа со знаком)* |
| **cmp** **al**,5 *;Сравнение AL и 5*  **jb** c1 *;Переход, если AL < 5 (числа без знака)* |

Команда TEST работает аналогично команде AND, но также результат не сохраняется, изменяются только флаги. С помощью этой команды можно проверить состояние различных битов операнда. Например:

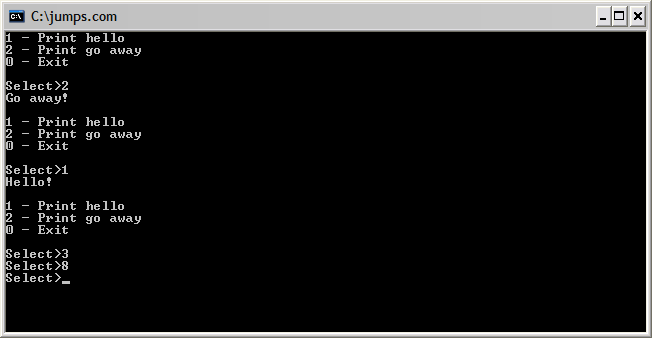
|  |
| --- |
| **test** **bl**,00000100b *;Проверить состояние 2-го бита BL*  **jz** c2 *;Переход, если 2-й бит равен 0* |

**Пример программы**

Простая программка, которая выводит меню и предлагает пользователю сделать выбор. Для ввода символа используется функция DOS 01h (при вводе символ отображается на экране). В зависимости от введённого символа осуществляется переход на нужный кусок кода. Для разнообразия, поместим данные в начале программы, а не в конце. Чтобы данные не выполнились как код, перед ними стоит команда безусловного перехода.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*  **jmp** **start** *;Безусловный переход на метку start*  *;----------------------------------------------------------*  menu **db** '1 - Print hello',13,10  **db** '2 - Print go away',13,10  **db** '0 - Exit',13,10,'$'  select **db** 13,10,'Select>$'  hello **db** 13,10,'Hello!',13,10,13,10,'$'  go\_away **db** 13,10,'Go away!',13,10,13,10,'$'  *;----------------------------------------------------------*  **start**:  **mov** **ah**,09h *;\*  **mov** **dx**,menu *; > Вывод меню*  **int** 21h *;/*    select\_loop:  **mov** **ah**,09h *;\*  **mov** **dx**,select *; > Вывод строки 'Select>'*  **int** 21h *;/*    **mov** **ah**,01h *;Функция DOS 01h - ввод символа*  **int** 21h *;Введённый символ помещается в AL*    **cmp** **al**,'1' *;Сравнение введённого символа с '1'*  **je** c1 *;Переход, если равно*  **cmp** **al**,'2' *;Сравнение введённого символа с '2'*  **je** c2 *;Переход, если равно*  **cmp** **al**,'0' *;Сравнение введённого символа с '0'*  **je** exit *;Переход, если равно*  **jmp** select\_loop *;Безусловный переход*  c1:  **mov** **ah**,09h *;\*  **mov** **dx**,hello *; > Вывод строки 'Hello'*  **int** 21h *;/*  **jmp** **start** *;Безусловный переход*  c2:  **mov** **ah**,09h *;\*  **mov** **dx**,go\_away *; > Вывод строки 'Go away'*  **int** 21h *;/*  **jmp** **start** *;Безусловный переход*  exit:  **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы* |

Скриншот работы программы:



**Упражнение**

Упражнение простое. Напишите программу для сравнения двух переменных со знаком *a* и *b*. В зависимости от результатов сравнения выведите «a < b», «a > b» или «a = b». Проверьте работу программы в отладчике.

**Часть 17. Команды LOOPZ и LOOPNZ**

Кроме команды LOOP и команд условных переходов существуют ещё две команды, позволяющие организовывать циклы. Это команды LOOPZ (или её синоним LOOPE) и LOOPNZ (синоним — LOOPNE). Действие этих команд очень напоминает LOOP, за исключением того, что дополнительно анализируется флаг нуля ZF.

Переход к метке цикла осуществляется в том случае, если после декремента содержимое CX не равно 0 и выполняется условие: ZF=1 (для команды LOOPZ/LOOPE) или ZF=0 (LOOPNZ/LOOPNE).

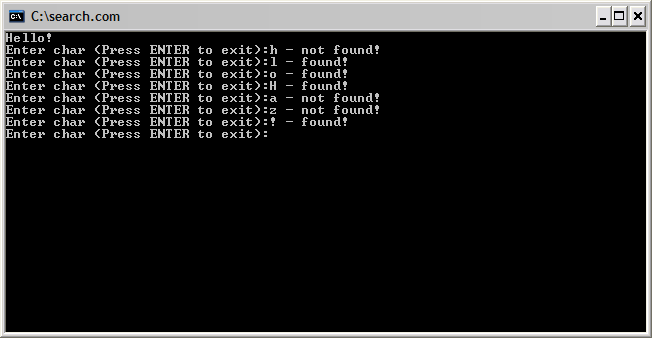
Эти команды удобно использовать в алгоритмах, где цикл должен завершаться в двух случаях:

* выполнено требуемое количество итераций;
* выполнено некоторое условие досрочного завершения цикла.

Простейший пример такого алгоритма — поиск числа или символа в массиве. Поиск завершается, если один из элементов массива совпал с искомым или если достигнут конец массива. В качестве примера рассмотрим программу для поиска символов в строке. Пользователь вводит символ, а программа определяет, содержится такой символ в строке или нет. Для выхода из программы нужно нажать ENTER.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*  **jmp** **start** *;Безусловный переход на метку start*  *;-- Данные ------------------------------------------------------------*  string **db** 'Hello!',13,10,'$'  length **db** 6  s\_entchar **db** 'Enter char (Press ENTER to exit):$'  s\_found **db** ' - found!',13,10,'$'  s\_nfound **db** ' - not found!',13,10,'$'  *;----------------------------------------------------------------------*  **start**:  **mov** **ah**,09h *;\*  **mov** **dx**,string *; > Вывод строки*  **int** 21h *;/*  main:  **mov** **dx**,s\_entchar *;\*  **int** 21h *;/ Вывод приглашения для ввода символа*    **mov** **ah**,01h *;\*  **int** 21h *;/ Ввод символа*  **cmp** **al**,0Dh *;Нажата клавиша ENTER?*  **je** exit *;Если да, то переход на метку exit*    **mov** **bx**,string-1 *;BX = (адрес строки - 1)*  **movzx** **cx**,[length] *;CX = длина строки*  search:  **inc** **bx** *;Инкремент BX*  **cmp** **al**,[**bx**] *;Сравнение введённого символа с символом строки*  **loopne** search *;Цикл, если не равно.*  **je** found *;Если равно, то символ найден.*    **mov** **dx**,s\_nfound *;DX = адрес строки ' - not found!'*  print\_result:  **mov** **ah**,09h *;\*  **int** 21h *;/ Вывод результата поиска*  **jmp** main *;Безусловный переход на метку main*  found:  **mov** **dx**,s\_found *;DX = адрес строки ' - found!'*  **jmp** print\_result *;Безусловный переход на метку print\_result*  exit:  **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы* |

Обратите внимание, изначально в BX загружается значение (адрес строки — 1), так как цикл начинается с команды инкремента. Результат работы программы:



**Упражнение**

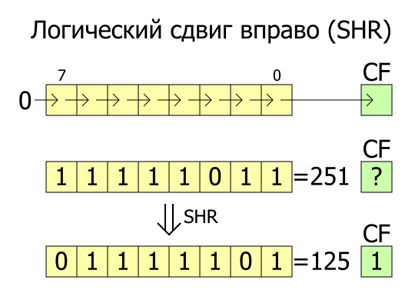
Объявите в программе два массива слов. Размер массивов должен быть одинаков и храниться в 8-битной переменной без знака. Напишите программу сравнения двух массивов, используя команду LOOPZ. (Массивы равны, если все их элементы соответственно равны. Цикл можно завершить, если найдена хотя бы одна пара не совпадающих элементов). Выведите на экран строку, сообщающую о результате сравнения. Сами массивы печатать не нужно.

**Часть 18. Линейный сдвиг**

Сдвиги — это особые операции процессора, которые позволяют реализовать различные преобразования данных, работать с отдельными битами, а также быстро выполнять умножение и деление чисел на степень 2. В этой части мы рассмотрим операции линейного сдвига, а в следующей части рассмотрим циклические операции сдвига.

**Логический сдвиг вправо**

Логический сдвиг всегда выполняется без учёта знакового бита. Для логического сдвига вправо предназначена команда SHR. У этой команды два операнда. Первый операнд представляет собой сдвигаемое значение и на его место записывается результат операции. Второй операнд указывает, на сколько бит нужно осуществить сдвиг. Этим операндом может быть либо непосредственное значение, либо регистр CL. Схема выполнения операции показана на рисунке:

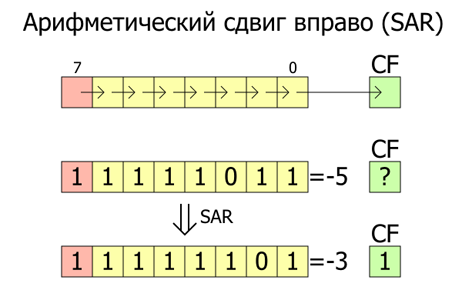


Все биты операнда сдвигаются вправо (от старших битов к младшим). Выдвинутый бит становится значением флага CF. Старший бит получает нулевое значение. Эта операция повторяется несколько раз, если второй операнд больше единицы. Логический сдвиг вправо можно использовать для деления целых чисел без знака на степень 2, причём сдвиг работает быстрее, чем команда деления DIV. Примеры:

|  |
| --- |
| **shr** **ax**,1 *;Логический сдвиг AX на 1 бит вправо*  **shr** **byte**[**bx**],**cl** *;Лог. сдвиг байта по адресу BX на СL бит вправо*  **shr** **cl**,4 *;CL = CL / 16 (для числа без знака)* |

**Арифметический сдвиг вправо**

Арифметический сдвиг отличается от логического тем, что он не изменяет значение старшего бита, и предназначен для чисел со знаком. Арифметический сдвиг вправо выполняется командой SAR. У этой команды тоже 2 операнда, аналогично команде SHR. Схема выполнения операции показана на рисунке:

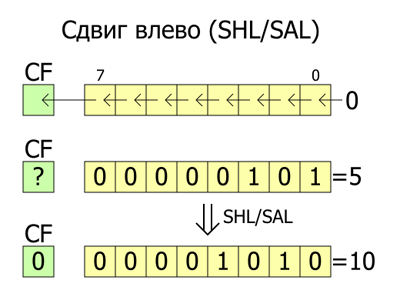


Выдвинутый бит становится значением флага CF. Знаковый бит не изменяется. При сдвиге на 1 бит сбрасывается флаг OF. Эту команду можно использовать для деления целых чисел со знаком на степень 2 (обратите внимание, что «округление» всегда в сторону меньшего числа, поэтому для отрицательных чисел результат будет отличаться от результата деления с помощью команды IDIV). Примеры:

|  |
| --- |
| **sar** **bx**,1 *;Арифметический сдвиг BX на 1 бит вправо*  **sar** **di**,**cl** *;Арифметический сдвиг DI на CL бит вправо*  **sar** [x],3 *;x = x / 8 (x - 8-битное значение со знаком)* |

**Логический и арифметический сдвиг влево**

Логический сдвиг влево выполняется командой SHL, а арифметический — командой SAL. Однако, на самом деле это просто синонимы для одной и той же машинной команды. Сдвиг влево одинаков для чисел со знаком и чисел без знака. У команды 2 операнда, аналогично командам SHR и SAR.Схема этой операции показана на рисунке:

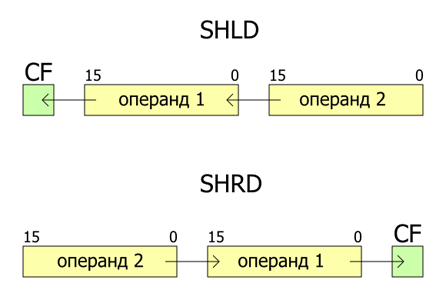


Старший бит становится значением флага CF, а младший получает нулевое значение. С помощью сдвига влево можно быстро умножать числа на степень 2. Но будьте внимательны, чтобы не получить в результате переполнение. Если при сдвиге на 1 бит меняется значение старшего бита, то устанавливается флаг OF. Примеры использования команды:

|  |
| --- |
| **shl** **dx**,1 *;Сдвиг DX на 1 бит влево*  **sal** **dx**,1 *;То же самое*  **shl** **ax**,**cl** *;Сдвиг AX на CL бит влево*  **sal** [x],2 *;x = x \* 4* |

**Сдвиги двойной точности**

Существуют ещё две команды, осуществляющие более сложные сдвиги. SHRD — сдвиг двойной точности вправо, SHLD — сдвиг двойной точности влево. У этих команд 3 операнда. Первый операнд — сдвигаемое значение и место для записи результата, должен иметь размер 16 бит. Второй операнд — источник вдвигаемых битов, тоже должен иметь размер 16 бит и находится в одном из регистров. Значение второго операнда не меняется. Третий операнд — счётчик сдвигов, может быть непосредственным значением или находиться в регистре CL. Схемы работы этих команд показаны на рисунке:



Небольшой пример использования команды SHLD:

|  |
| --- |
| **shld** **ax**,**bx**,3 *;Сдвинуть ax на 3 бита влево,*  *;3 старших бита BX становятся младшими битами AX* |

**Пример программы**

Программа печатает переменную размером 16-бит в двоичном виде. Используется команда сдвига влево на 1 бит, после чего анализируется значение флага CF. Если CF=0, выводим символ ‘0’, если СF=1 выводим символ ‘1’. Проверка битов осуществляется в цикле.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*  **jmp** **start** *;Безусловный переход на метку start*  *;-- Данные ------------------------------------------------------------*  v **dw** 12345  pak **db** 13,10,'Press any key...$'  *;----------------------------------------------------------------------*  **start**:  **mov** **bx**,[v] *;BX = v*  **mov** **ah**,2 *;Функция DOS 02h - вывод символа*  **mov** **cx**,16 *;Инициализация счётчика цикла*  lp:  **shl** **bx**,1 *;Сдвиг BX на 1 бит влево*  **mov** **dl**,'0' *;dl = '0'*  **jnc** print *;Переход, если выдвинутый бит равен 0*  **inc** **dl** *;dl = dl + 1 = '1'*  print:  **int** 21h *;Обращение к функции DOS 02h*  **loop** lp *;Команда цикла*    **mov** **ah**,9 *;\*  **mov** **dx**,pak *; > Вывод строки 'Press any key...'*  **int** 21h *;/*    **mov** **ah**,8 *;\*  **int** 21h *;/ Ввод символа без эха*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы* |

Результат работы программы:



**Упражнение**

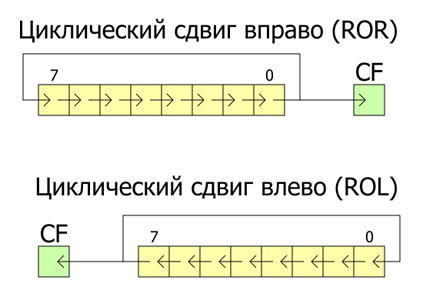
Объявите массив из 8 слов без знака. Сдвиньте первый элемент на 1 бит влево, второй элемент — на 2 бита вправо (логическим сдвигом), третий элемент — на 3 бита влево и т.д. до конца массива. Используйте циклы. Проверьте работу программы в отладчике.

**Часть 19. Циклический сдвиг**

Циклический сдвиг отличается от линейного тем, что выдвигаемые с одного конца биты вдвигаются с другой стороны, то есть движутся по кольцу. В процессора x86 существует 2 вида циклического сдвига: простой и через флаг переноса (CF). У всех команд, рассматриваемых в этой части учебного курса, по 2 операнда, таких же, как у команд линейного сдвига. Первый операнд — сдвигаемое значение и место для записи результата. Второй операнд — счётчик сдвигов, который может находится в регистре CL или указываться непосредственно.

**Простой циклический сдвиг**

Циклический сдвиг вправо выполняется командой ROR, а влево — командой ROL. Схема работы этих команд представлена на рисунке (на примере 8-битного операнда):

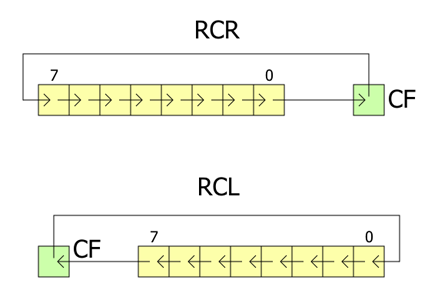


Значение последнего выдвигаемого бита копируется в флаг CF. Для сдвигов на 1 бит устанавливается флаг OF, если в результате сдвига изменяется знаковый бит операнда. Примеры использования команд:

|  |
| --- |
| **rol** **bl**,1 *;Циклический сдвиг BL на 1 бит влево*  **ror** **word**[**si**],5 *;Циклический сдвиг слова по адресу в SI на 5 бит вправо*  **rol** **ax**,**cl** *;Циклический свдиг AX на CL бит влево* |

**Циклический сдвиг через флаг переноса**

Отличие от простого циклического сдвига в том, что флаг CF участвует в сдвиге наравне с битами операнда. При сдвиге на 1 бит выдвигаемый бит помещается в CF, а значение CF вдвигается в операнд с другой стороны. При сдвиге на несколько бит эта операция повторяется многократно. Циклический сдвиг через флаг переноса выполняется командами RCR (вправо) и RCL (влево).



Опять же для сдвигов на 1 бит устанавливается флаг OF, если в результате сдвига изменяется знаковый бит операнда. Примеры использования команд:

|  |
| --- |
| **rcr** **dh**,3 *;Цикл. сдвиг DH на 3 бита вправо через флаг CF*  **rcl** **byte**[**bx**],**cl** *;Цикл. сдвиг байта по адресу в BX на CL бит влево через флаг CF*  **rcl** **dx**,1 *;Цикл. сдвиг DX на 1 бит влево через флаг CF* |

**Пример программы**

В качестве примера напишем программу для подсчёта единичных битов в байте. Для анализа битов используется команда ROL в цикле. Цикл выполняется 8 раз — по числу битов в байте. Если очередной бит равен 1, то выполняем инкремент счётчика единичных битов.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **al**,[x] *;AL = x*  **xor** **bl**,**bl** *;BL = 0 (Здесь будем считать единичные биты)*  **mov** **cx**,8 *;Инициализация счётчика цикла*  lp:  **rol** **al**,1 *;Цилический сдвиг AL на 1 бит влево*  **jnc** bit0 *;Переход, если CF=0*  **inc** **bl** *;Инкремент счетчика единичных битов*  bit0:  **loop** lp *;Команда цикла*  **mov** [n],**bl** *;Сохраняем результат в n*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;----------------------------------------------------------------------*  x **db** 89h *;Байт*  n **db** ? *;Количество единичных битов в байте* |

Эту программу можно немного оптимизировать. Во-первых, использовать вместо условного перехода команду ADC со нулевым вторым операндом. Это позволит прибавить 1, если CF=1 и прибавить 0, если CF=0. Во-вторых, считать единичные биты можно в регистре AH, а обнулить его в начале с помощью команды MOVZX, совместив с загрузкой байта в регистр AL. Ноль для команды ADC можно взять в регистре CH, это делает команду короче и быстрее, чем при использовании непосредственного операнда. CH равен 0 во время выполнения цикла, так как CX изменяется от 8 до 0. Вот что получилось в итоге:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **movzx** **ax**,[x] *;AL = x, AH = 0*  **mov** **cx**,8 *;Инициализация счётчика цикла*  lp:  **rol** **al**,1 *;Цилический сдвиг AL на 1 бит влево*  **adc** **ah**,**ch** *;Прибавляем флаг CF к AH, так как CH = 0*  **loop** lp *;Команда цикла*  **mov** [n],**bl** *;Сохраняем результат в n*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*  *;----------------------------------------------------------------------*  x **db** 89h *;Байт*  n **db** ? *;Количество единичных битов в байте* |

Всего 6 команд для подсчета битов в байте. А попробуйте написать то же самое на языке высокого уровня ;)

**Упражнение**

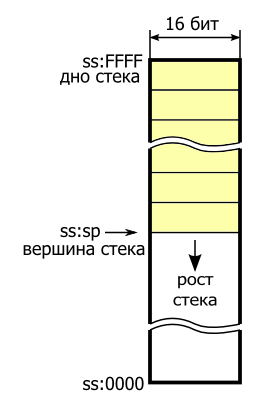
Объявите в программе строку. Длина строки должна быть больше 8 символов и храниться в байте без знака. Напишите цикл для шифрования строки по алгоритму: первый символ циклически сдвигается вправо на 1 бит, второй символ — на 2 бита, …, 7-й — на 7 битов, 8-й — снова на 1 бит, 9-й на 2 бита и т.д. Затем напишите цикл для расшифровки строки и выведите её на экран.

**Часть 20. Стек**

Стеком называется структура данных, организованная по принципу LIFO («Last In — First Out» или «последним пришёл — первым ушёл»). Стек является неотъемлемой частью архитектуры процессора и поддерживается на аппаратном уровне: в процессоре есть специальные регистры (SS, BP, SP) и команды для работы со стеком.

Обычно стек используется для сохранения адресов возврата и передачи аргументов при вызове процедур (о процедурах в следующей части), также в нём выделяется память для локальных переменных. Кроме того, в стеке можно временно сохранять значения регистров.

Схема организации стека в процессоре 8086 показана на рисунке:



Стек располагается в оперативной памяти в сегменте стека, и поэтому адресуется относительно сегментного регистра SS. Шириной стека называется размер элементов, которые можно помещать в него или извлекать. В нашем случае ширина стека равна двум байтам или 16 битам. Регистр SP (указатель стека) содержит адрес последнего добавленного элемента. Этот адрес также называется вершиной стека. Противоположный конец стека называется «дном».

Дно стека находится в верхних адресах памяти. При добавлении новых элементов в стек значение регистра SP уменьшается, то есть стек растёт в сторону младших адресов. Как вы помните, для COM-программ данные, код и стек находятся в одном и том же сегменте, поэтому если постараться, стек может разрастись и затереть часть данных и кода.

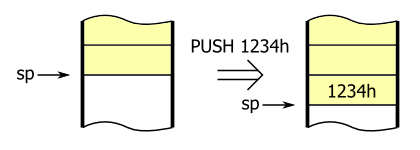
Для стека существуют всего две основные операции:

* добавление элемента на вершину стека (PUSH);
* извлечение элемента с вершины стека (POP);

**Добавление элемента в стек**

Выполняется командой PUSH. У этой команды один операнд, который может быть непосредственным значением, 16-битным регистром (в том числе сегментым) или 16-битной переменной в памяти. Команда работает следующим образом:

1. значение в регистре SP уменьшается на 2 (так как ширина стека — 16 бит или 2 байта);
2. операнд помещается в память по адресу в SP.



Примеры:

|  |
| --- |
| **push** -5 *;Поместить -5 в стек*  **push** **ax** *;Поместить AX в стек*  **push** **ds** *;Поместить DS в стек*  **push** [x] *;Поместить x в стек (x объявлен как слово)*  **push** **word** [**bx**] *;Поместить в стек слово по адресу в BX* |

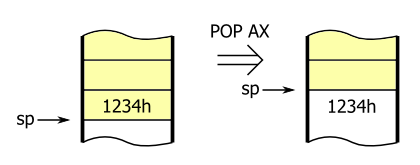
Существуют ещё 2 команды для добавления в стек. Команда PUSHF помещает в стек содержимое регистра флагов. Команда PUSHA помещает в стек содержимое всех регистров общего назначения в следующем порядке: АХ, СХ, DX, ВХ, SP, BP, SI, DI (значение DI будет на вершине стека). Значение SP помещается то, которое было до выполнения команды. Обе эти команды не имеют операндов.

**Извлечение элемента из стека**

Выполняется командой POP. У этой команды также один операнд, который может быть 16-битным регистром (в том числе сегментым, но кроме CS) или 16-битной переменной в памяти. Команда работает следующим образом:

1. операнд читается из памяти по адресу в SP;
2. значение в регистре SP увеличивается на 2.

Обратите внимание, что извлеченный из стека элемент не обнуляется и не затирается в памяти, а просто остаётся как мусор. Он будет перезаписан при помещении нового значения в стек.



Примеры:

|  |
| --- |
| **pop** **cx** *;Поместить значение из стека в CX*  **pop** **es** *;Поместить значение из стека в ES*  **pop** [x] *;Поместить значение из стека в переменную x*  **pop** **word** [**di**] *;Поместить значение из стека в слово по адресу в DI* |

Соответственно, есть ещё 2 команды. POPF помещает значение с вершины стека в регистр флагов. POPA восстанавливает из стека все регистры общего назначения (но при этом значение для SP игнорируется).

**Пример программы**

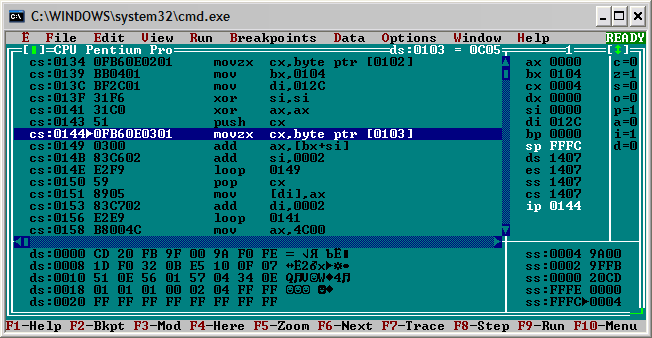
Имеется двумерный массив — таблица 16-битных значений со знаком размером *n* строк на *m* столбцов. Программа вычисляет сумму элементов каждой строки и сохраняет результат в массиве *sum*. Первый элемент массива будет содержать сумму элементов первой строки, второй элемент — сумму элементов второй строки и так далее.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*  **jmp** **start** *;Переход к метке start*  *;----------------------------------------------------------------------*  *; Данные*  n **db** 4 *;Количество строк*  m **db** 5 *;Количество столбцов*  *;Двумерный массив - таблица c данными*  table:  **dw** 12,45, 0,82,34  **dw** 46,-5,87,11,56  **dw** 35,21,77,90,-9  **dw** 44,13,-1,99,32  sum rw 4 *;Массив для сумм каждой строки*  *;----------------------------------------------------------------------*  **start**:  **movzx** **cx**,[n] *;Счётчик строк*  **mov** **bx**,table *;BX = адрес таблицы*  **mov** **di**,sum *;DI = адрес массива для сумм*  **xor** **si**,**si** *;SI = смещение элемента от начала таблицы*    rows:  **xor** **ax**,**ax** *;Обнуление AX. В AX будет считаться сумма*  **push** **cx** *;Сохранение значения CX*    **movzx** **cx**,[m] *;Инициализация CX для цикла по строке*  calc\_sum:  **add** **ax**,[**bx**+**si**] *;Прибавление элемента строки*  **add** **si**,2 *;SI = смещение следующего элемента*  **loop** calc\_sum *;Цикл суммирования строки*    **pop** **cx** *;Восстановление значения CX*  **mov** [**di**],**ax** *;Сохранение суммы строки*  **add** **di**,2 *;DI = адрес следующей ячейки для суммы строки*  **loop** rows *;Цикл по всем строкам таблицы*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы* |

В программе два вложенных цикла: внешний и внутренний. Внешний цикл — это цикл по строкам таблицы. Внутренний цикл вычисляет сумму элементов строки. Стек здесь используется для временного хранения счётчика внешнего цикла. Перед началом внутреннего цикла CX сохраняется в стеке, а после завершения восстанавливается. Такой приём можно использовать для программирования и большего количества вложенных циклов.

**Turbo Debugger**

В отладчике Turbo Debugger стек отображается в нижней правой области окна CPU. Левый столбец чисел — адреса, правый — данные. Треугольник указывает на вершину стека, то есть на тот адрес, который содержится в регистре SP. Если запустить программу в отладчике, то можно увидеть, как работают команды «push cx» и «pop cx».



**Упражнение**

Объявите в программе строку «$!olleH». Напишите код для переворачивания строки с использованием стека (в цикле поместите каждый символ в стек, а затем извлеките в обратном порядке). Выведите полученную строку на экран.

**Часть 21. Простые процедуры**

В этой части учебного курса мы рассмотрим основы создания процедур. Процедура представляет собой код, который может выполняться многократно и к которому можно обращаться из разных частей программы. Обычно процедуры предназначены для выполнения каких-то отдельных, законченных действий программы и поэтому их иногда называют подпрограммами. В других языках программирования процедуры могут называться функциями или методами.

**Команды CALL и RET**

Для работы с процедурами предназначены команды CALL и RET. С помощью команды CALL выполняется *вызов* процедуры. Эта команда работает почти также, как команда безусловного перехода (JMP), но с одним отличием — одновременно в стек сохраняется текущее значение регистра IP. Это позволяет потом вернуться к тому месту в коде, откуда была вызвана процедура. В качестве операнда указывается адрес перехода, который может быть непосредственным значением (меткой), 16-разрядным регистром (кроме сегментных) или ячейкой памяти, содержащей адрес.

Возврат из процедуры выполняется командой RET. Эта команда восстанавливает значение из вершины стека в регистр IP. Таким образом, выполнение программы продолжается с команды, следующей сразу после команды CALL. Обычно код процедуры заканчивается этой командой. Команды CALL и RET не изменяют значения флагов (кроме некоторых особых случаев в защищенном режиме). Небольшой пример разных способов вызова процедуры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    **mov** **ax**,myproc  **mov** **bx**,myproc\_addr  **xor** **si**,**si**    **call** myproc *;Вызов процедуры (адрес перехода - myproc)*  **call** **ax** *;Вызов процедуры по адресу в AX*  **call** [myproc\_addr] *;Вызов процедуры по адресу в переменной*  **call** **word** [**bx**+**si**] *;Более сложный способ задания адреса ;)*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*    *;----------------------------------------------------------------------*  *;Процедура, которая ничего не делает*  myproc:  **nop** *;Код процедуры*  **ret** *;Возврат из процедуры*  *;----------------------------------------------------------------------*  myproc\_addr **dw** myproc *;Переменная с адресом процедуры* |

**Ближние и дальние вызовы процедур**

Существует 2 типа вызовов процедур. *Ближним* называется вызов процедуры, которая находится в текущем сегменте кода. *Дальний* вызов — это вызов процедуры в другом сегменте. Соответственно существуют 2 вида команды RET — для ближнего и дальнего возврата. Компилятор FASM автоматически определяет нужный тип машинной команды, поэтому в большинстве случаев не нужно об этом беспокоиться.

В учебном курсе мы будем использовать только ближние вызовы процедур.

**Передача параметров**

Очень часто возникает необходимость передать процедуре какие-либо параметры. Например, если вы пишете процедуру для вычисления суммы элементов массива, удобно в качестве параметров передавать ей адрес массива и его размер. В таком случае одну и ту же процедуру можно будет использовать для разных массивов в вашей программе. Самый простой способ передать параметры — это поместить их в регистры перед вызовом процедуры.

**Возвращаемое значение**

Кроме передачи параметров часто нужно получить какое-то значение из процедуры. Например, если процедура что-то вычисляет, хотелось бы получить результат вычисления. А если процедура что-то делает, то полезно узнать, завершилось действие успешно или возникла ошибка. Существуют разные способы возврата значения из процедуры, но самый часто используемый — это поместить значение в один из регистров. Обычно для этой цели используют регистры AL и AX.

**Сохранение регистров**

Хорошим приёмом является сохранение регистров, которые процедура изменяет в ходе своего выполнения. Это позволяет вызывать процедуру из любой части кода и не беспокоиться, что значения в регистрах будут испорчены. Обычно регистры сохраняются в стеке с помощью команды PUSH, а перед возвратом из процедуры восстанавливаются командой POP. Естественно, восстанавливать их надо в обратном порядке. Примерно вот так:

|  |
| --- |
| myproc:  **push** **bx** *;Сохранение регистров*  **push** **cx**  **push** **si**  ... *;Код процедуры*  **pop** **si** *;Восстановление регистров*  **pop** **cx**  **pop** **bx**  **ret** *;Возврат из процедуры* |

**Пример**

Для примера напишем процедуру для вывода собщения в рамке и протестируем её работу, выведя несколько сообщений. В качестве параметра ей будет передаватся адрес строки в регистре BX. Строка должна заканчиваться символом ‘$’. Для упрощения процедуры можно разбить задачу на подзадачи и написать соответствующие процедуры. Прежде всего нужно вычислить длину строки, чтобы знать ширину рамки. Процедура *get\_length* вычисляет длину строки (адрес передаётся также в BX) и возвращает её в регистре AX.

Для рисования горизонтальной линии из символов предназначена процедура *draw\_line*. В DL передаётся код символа, а в CX — количество символов, которое необходимо вывести на экран. Эта процедура не возвращает никакого значения. Для вывода 2-х символов конца строки написана процедура *print\_endline*. Она вызывается без параметров и тоже не возвращает никакого значения. Коды символов для рисования рамок можно узнать с помощью таблицы символов кодировки 866 или можно воспользоваться стандартной программой Windows «Таблица символов», выбрав шрифт Terminal.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*  **jmp** **start** *;Переход на метку start*  *;----------------------------------------------------------------------*  msg1 **db** 'Hello!$'  msg2 **db** 'asmworld.ru$'  msg3 **db** 'Press any key...$'  *;----------------------------------------------------------------------*  **start**:  **mov** **bx**,msg1  **call** print\_message *;Вывод первого сообщения*  **mov** **bx**,msg2  **call** print\_message *;Вывод второго сообщения*  **mov** **bx**,msg3  **call** print\_message *;Вывод третьего сообщения*    **mov** **ah**,8 *;Ввод символа без эха*  **int** 21h    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*    *;----------------------------------------------------------------------*  *;Процедура вывода сообщения в рамке*  *;В BX передаётся адрес строки*  print\_message:  **push** **ax** *;Сохранение регистров*  **push** **cx**  **push** **dx**    **call** get\_length *;Вызов процедуры вычисления длины строки*  **mov** **cx**,**ax** *;Копируем длину строки в CX*  **mov** **ah**,2 *;Функция DOS 02h - вывод символа*  **mov** **dl**,0xDA *;Левый верхний угол*  **int** 21h  **mov** **dl**,0xC4 *;Горизонтальная линия*  **call** draw\_line *;Вызов процедуры рисования линии*  **mov** **dl**,0xBF *;Правый верхний угол*  **int** 21h  **call** print\_endline *;Вызов процедуры вывода конца строки*    **mov** **dl**,0xB3 *;Вертикальная линия*  **int** 21h  **mov** **ah**,9 *;Функция DOS 09h - вывод строки*  **mov** **dx**,**bx** *;Адрес строки в DX*  **int** 21h  **mov** **ah**,2 *;Функция DOS 02h - вывод символа*  **mov** **dl**,0xB3 *;Вертикальная линия*  **int** 21h  **call** print\_endline *;Вызов процедуры вывода конца строки*    **mov** **dl**,0xC0 *;Левый нижний угол*  **int** 21h  **mov** **dl**,0xC4 *;Горизонтальная линия*  **call** draw\_line  **mov** **dl**,0xD9 *;Правый нижний угол*  **int** 21h  **call** print\_endline *;Вызов процедуры вывода конца строки*    **pop** **dx** *;Восстановление регистров*  **pop** **cx**  **pop** **ax**  **ret** *;Возврат из процедуры*    *;----------------------------------------------------------------------*  *;Процедура вычисления длины строки (конец строки - символ '$').*  *;В BX передаётся адрес строки.*  *;Возвращает длину строки в регистре AX.*  get\_length:  **push** **bx** *;Сохранение регистра BX*  **xor** **ax**,**ax** *;Обнуление AX*  str\_loop:  **cmp** **byte**[**bx**],'$' *;Проверка конца строки*  **je** str\_end *;Если конец строки, то выход из процедуры*  **inc** **ax** *;Инкремент длины строки*  **inc** **bx** *;Инкремент адреса*  **jmp** str\_loop *;Переход к началу цикла*  str\_end:  **pop** **bx** *;Восстановление регистра BX*  **ret** *;Возврат из процедуры*    *;----------------------------------------------------------------------*  *;Процедура рисования линии из символов.*  *;В DL - символ, в CX - длина линии (кол-во символов)*  draw\_line:  **push** **ax** *;Сохранение регистров*  **push** **cx**  **mov** **ah**,2 *;Функция DOS 02h - вывод символа*  drl\_loop:  **int** 21h *;Обращение к функции DOS*  **loop** drl\_loop *;Команда цикла*  **pop** **cx** *;Восстановление регистров*  **pop** **ax**  **ret** *;Возврат из процедуры*    *;----------------------------------------------------------------------*  *;Процедура вывода конца строки (CR+LF)*  print\_endline:  **push** **ax** *;Сохранение регистров*  **push** **dx**  **mov** **ah**,2 *;Функция DOS 02h - вывод символа*  **mov** **dl**,13 *;Символ CR*  **int** 21h  **mov** **dl**,10 *;Символ LF*  **int** 21h  **pop** **dx** *;Восстановление регистров*  **pop** **ax**  **ret** *;Возврат из процедуры* |

Результат работы программы выглядит:



**Отладчик Turbo Debugger**

Небольшое замечание по поводу использования отладчика. В Turbo Debugger нажимайте F7 (*«Trace into»*), чтобы перейти к коду вызываемой процедуры. При нажатии F8(*«Step over»*) процедура будет выполнена сразу целиком.

**Упражнение**

Объявите в программе 2-3 массива слов без знака. Количество элементов каждого массива должно быть разным и храниться в отдельной 16-битной переменной без знака. Напишите процедуру для вычисления среднего арифметического массива чисел. В качестве параметров ей будет передаваться адрес массива и количество элементов, а возвращать она будет вычисленное значение. С помощью процедуры вычислите среднее арифметическое каждого массива и сохраните где-нибудь в памяти. Выводить числа на экран не нужно, этим мы займемся в следующей части лекций.

**Часть 22. Вывод чисел на консоль**

В качестве примера программирования процедур изучим способы вывода на консоль чисел в различных системах счисления. Проблема эта возникает потому, что в ассемблере нет никаких специальных средств для вывода чисел, а с помощью стандартных функций можно выводить только строки.

Следовательно, задача сводится к тому, чтобы преобразовать двоичное число в строку символов, а затем вывести эту строку на экран. Все процедуры в этой части являются лишь примерами, вы можете использовать их или написать свои собственные процедуры, более удобные для Вас.

Для начала рассмотрим две полезные процедуры, которые будут использоваться в дальнейшем. Чтобы постоянно не обращаться в коде в функции DOS 09h, удобно написать маленькую процедуру для вывода строки:

|  |
| --- |
| *;Процедура вывода строки на консоль*  *; DI - адрес строки*  print\_str:  **push** **ax**  **mov** **ah**,9 *;Функция DOS 09h - вывод строки*  **xchg** **dx**,**di** *;Обмен значениями DX и DI*  **int** 21h *;Обращение к функции DOS*  **xchg** **dx**,**di** *;Обмен значениями DX и DI*  **pop** **ax**  **ret** |

В качестве параметра ей передаётся адрес строки в регистре DI. Строка должна оканчиваться символом ‘$’. Здесь используется команда XCHG, которая выполняет обмен значениями двух операндов.

Вторая полезная процедура — вывод конца строки. Она вызывает первую процедуру для вывода двух символов CR(13) и LF(10). Вызывается без параметров и не изменяет регистры.

|  |
| --- |
| *;Процедура вывода конца строки (CR+LF)*  print\_endline:  **push** **di**  **mov** **di**,endline *;DI = адрес строки с символами CR,LF*  **call** print\_str *;Вывод строки на консоль*  **pop** **di**  **ret**  ...  endline **db** 13,10,'$' |

**Вывод чисел в двоичном виде**

Алгоритм вывода в двоичном виде очень прост. Нужно проанализировать все биты числа и поместить в строку символы ‘0’ или ‘1’ в зависимости от значения соответствующего бита. Удобно делать это с помощью циклического сдвига в цикле. Сдвинутый бит оказывается в флаге CF, а после завершения цикла в регистре то же значение, что и в начале. Процедура *byte\_to\_bin\_str* преобразует байт в регистре AL в строку. Адрес буфера для строки передаётся в регистре DI. Процедура всегда записывает в буфер 8 символов, так как в байте 8 бит.

|  |
| --- |
| *;Процедура преобразования байта в строку в двоичном виде*  *; AL - байт.*  *; DI - буфер для строки (8 символов). Значение регистра не сохраняется.*  byte\_to\_bin\_str:  **push** **cx** *;Сохранение CX*  **mov** **cx**,8 *;Счётчик цикла*    btbs\_lp:  **rol** **al**,1 *;Циклический сдвиг AL влево на 1 бит*  **jc** btbs\_1 *;Если выдвинутый бит = 1, то переход*  **mov** **byte**[**di**],'0' *;Добавление символа '0' в строку*  **jmp** btbs\_end  btbs\_1:  **mov** **byte**[**di**],'1' *;Добавление символа '1' в строку*  btbs\_end:  **inc** **di** *;Инкремент DI*  **loop** btbs\_lp *;Команда цикла*    **pop** **cx** *;Восстановление CX*  **ret** *;Возврат из процедуры* |

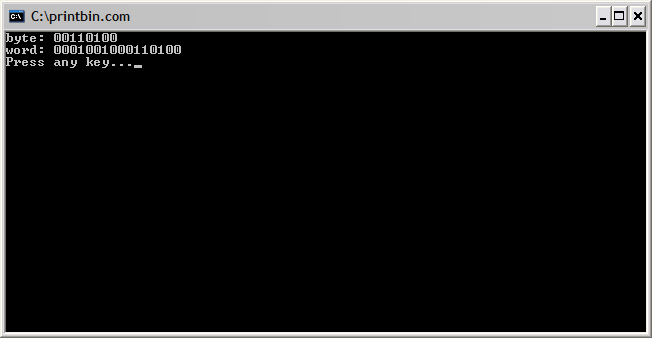
Используя эту процедуру, легко написать ещё одну для вывода слова в двоичном виде:

|  |
| --- |
| *;Процедура преобразования слова в строку в двоичном виде*  *; AX - слово*  *; DI - буфер для строки (16 символов). Значение регистра не сохраняется.*  word\_to\_bin\_str:  **xchg** **ah**,**al** *;Обмен AH и AL*  **call** byte\_to\_bin\_str *;Преобразование старшего байта в строку*  **xchg** **ah**,**al** *;Обмен AH и AL*  **call** byte\_to\_bin\_str *;Преобразование младшего байта в строку*  **ret** |

И наконец приведем пример двух процедур, которые сделают то, что нам нужно. Буфер имеет размер 17 символов, так как в слове 16 бит + символ ‘$’, обозначающий конец строки.

|  |
| --- |
| *;Процедура вывода байта на консоль в двоичном виде*  *; AL - байт*  print\_byte\_bin:  **push** **di**  **mov** **di**,buffer *;DI = адрес буфера*  **call** byte\_to\_bin\_str *;Преобразование байта в AL в строку*  **mov** **byte**[**di**],'$' *;Добавление символа конца строки*  **sub** **di**,8 *;DI = адрес начала строки*  **call** print\_str *;Вывод строки на консоль*  **pop** **di**  **ret**    *;Процедура вывода слова на консоль в двоичном виде*  *; AX - слово*  print\_word\_bin:  **push** **di**  **mov** **di**,buffer *;DI = адрес буфера*  **call** word\_to\_bin\_str *;Преобразование слова в AX в строку*  **mov** **byte**[**di**],'$' *;Добавление символа конца строки*  **sub** **di**,16 *;DI = адрес начала строки*  **call** print\_str *;Вывод строки на консоль*  **pop** **di**  **ret**  ...  buffer rb 17 |

Полный исходный код примера вы можете просмотреть в файле: printbin.asm. Результат работы программы выглядит вот так:



**Вывод чисел в шестнадцатеричном виде**

Этот пример по структуре похож на предыдущий, поэтому для краткости рассмотрим только сами процедуры преобразования числа в строку. Преобразование в шестнадцатеричный вид удобно выполнять группами по 4 бита, то есть по тетрадам. Каждая тетрада будет представлять собой одну шестнадцатеричную цифру. Ниже приведм пример процедуры для преобразования тетрады в символ цифры:

|  |
| --- |
| *;Процедура преобразования числа (0-15) в шестнадцатеричную цифру*  *; вход : AL - число (0-15)*  *; выход: AL - шестнадцатеричная цифра ('0'-'F')*  to\_hex\_digit:  **add** **al**,'0' *;Прибавляем символ '0' (код 0x30)*  **cmp** **al**,'9' *;Сравнение с символом '9' (код 0x39)*  **jle** thd\_end *;Если получилось '0'-'9', то выход*  **add** **al**,7 *;Прибавляем ещё 7 для символов 'A'-'F'*  thd\_end:  **ret** |

Если значение тетрады от 0 до 9, то достаточно только прибавить код символа ‘0’ (0x30). А если значение больше 9, то надо прибавить ещё 7, чтобы получилась буква ‘A’-‘F’.

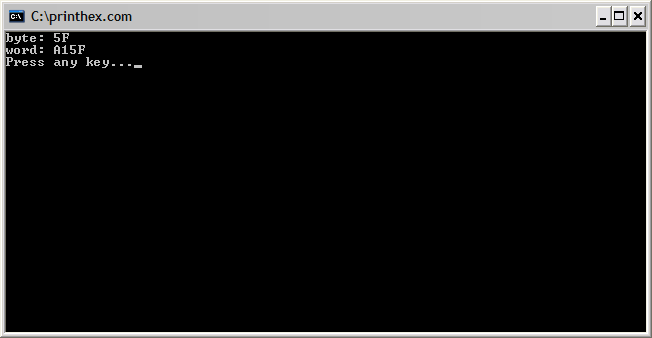
Теперь легко можно преобразовать байт в шестнадцатеричную строку, достаточно каждую из его тетрад заменить соответствующей цифрой:

|  |
| --- |
| *;Процедура преобразования байта в строку в шестнадцатеричном виде*  *; AL - байт.*  *; DI - буфер для строки (2 символа). Значение регистра не сохраняется.*  byte\_to\_hex\_str:  **push** **ax**  **mov** **ah**,**al** *;Сохранение значения AL в AH*  **shr** **al**,4 *;Выделение старшей тетрады*  **call** to\_hex\_digit *;Преобразование в шестнадцатеричную цифру*  **mov** [**di**],**al** *;Добавление символа в строку*  **inc** **di** *;Инкремент DI*  **mov** **al**,**ah** *;Восстановление AL*  **and** **al**,0Fh *;Выделение младшей тетрады*  **call** to\_hex\_digit *;Преобразование в шестнадцатеричную цифру*  **mov** [**di**],**al** *;Добавление символа в строку*  **inc** **di** *;Инкремент DI*  **pop** **ax**  **ret** |

Преобразование слова также не представляет трудности — сначала преобразуем старший байт, затем младший:

|  |
| --- |
| *;Процедура преобразования слова в строку в шестнадцатеричном виде*  *; AX - слово*  *; DI - буфер для строки (4 символа). Значение регистра не сохраняется.*  word\_to\_hex\_str:  **xchg** **ah**,**al** *;Обмен AH и AL*  **call** byte\_to\_hex\_str *;Преобразование старшего байта в строку*  **xchg** **ah**,**al** *;Обмен AH и AL*  **call** byte\_to\_hex\_str *;Преобразование младшего байта в строку*  **ret** |

Полный исходный код примера можно просмотреть в файле: printhex.asm. Результат работы программы выглядит вот так:



**Вывод чисел в десятичном виде**

С десятичными числами немного сложнее. Для начала изучим принципы работы с числами без знака. Чтобы преобразовать число в десятичную строку необходимо в цикле делить его на 10 (это основание системы счисления). Остатки от деления дают нам значения десятичных цифр. Первый остаток — младшая цифра, последний — старшая. Деление продолжается пока частное не равно нулю.

Например, если есть число 125. Делим его на десять: получаем 12, 5 в остатке. Потом делим 12 на десять: получаем 1, 2 в остатке. Наконец, 1 делим на 10: получаем 0, 1 в остатке. Цифры числа, начиная с младшей: 5, 2, 1. Так как обычно десятичные числа пишут, начиная со старшей цифры, то необходимо переставить их наоборот. Для этого можно использовать стек.

В первом цикле производится деление, полученные остатки преобразуются в цифры и помещаются в стек. Во втором цикле символы извлекаются из стека (в обратном порядке) и помещаются в строку. Так как максимальное значение слова без знака 65536 (5 цифр), то в буфер записывается максимум 5 символов.

|  |
| --- |
| *;Процедура преобразования слова в строку в десятичном виде (без знака)*  *; AX - слово*  *; DI - буфер для строки (5 символов). Значение регистра не сохраняется.*  word\_to\_udec\_str:  **push** **ax**  **push** **cx**  **push** **dx**  **push** **bx**  **xor** **cx**,**cx** *;Обнуление CX*  **mov** **bx**,10 *;В BX делитель (10 для десятичной системы)*    wtuds\_lp1: *;Цикл получения остатков от деления*  **xor** **dx**,**dx** *;Обнуление старшей части двойного слова*  **div** **bx** *;Деление AX=(DX:AX)/BX, остаток в DX*  **add** **dl**,'0' *;Преобразование остатка в код символа*  **push** **dx** *;Сохранение в стеке*  **inc** **cx** *;Увеличение счетчика символов*  **test** **ax**,**ax** *;Проверка AX*  **jnz** wtuds\_lp1 *;Переход к началу цикла, если частное не 0.*    wtuds\_lp2: *;Цикл извлечения символов из стека*  **pop** **dx** *;Восстановление символа из стека*  **mov** [**di**],**dl** *;Сохранение символа в буфере*  **inc** **di** *;Инкремент адреса буфера*  **loop** wtuds\_lp2 *;Команда цикла*    **pop** **bx**  **pop** **dx**  **pop** **cx**  **pop** **ax**  **ret** |

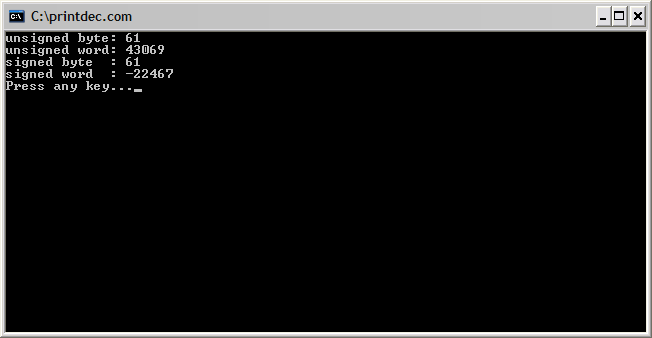
Для вывода байта можно преобразовать его в слово и воспользоваться той же процедурой:

|  |
| --- |
| *;Процедура преобразования байта в строку в десятичном виде (без знака)*  *; AL - байт.*  *; DI - буфер для строки (3 символа). Значение регистра не сохраняется.*  byte\_to\_udec\_str:  **push** **ax**  **xor** **ah**,**ah** *;Преобразование байта в слово (без знака)*  **call** word\_to\_udec\_str *;Вызов процедуры для слова без знака*  **pop** **ax**  **ret** |

Теперь изучим способы работы с числами со знаком. Сначала нужно проверить старший бит числа. Если число положительное, то его можно преобразовать также как число без знака. Если число отрицательное, то добавляем в строку символ ‘-‘, а затем инвертируем число и преобразуем как беззнаковое.

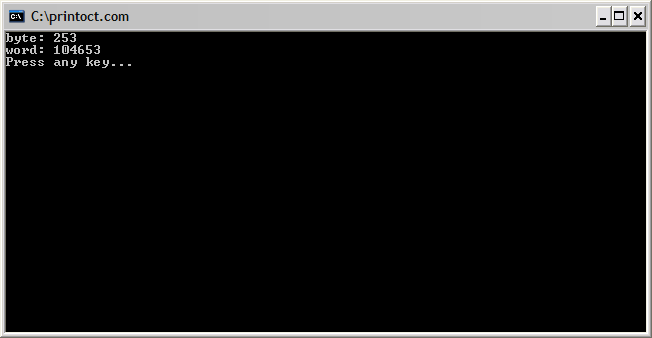
|  |  |
| --- | --- |
| *;Процедура преобразования слова в строку в десятичном виде (со знаком)*  *; AX - слово*  *; DI - буфер для строки (6 символов). Значение регистра не сохраняется.*  word\_to\_sdec\_str:  **push** **ax**  **test** **ax**,**ax** *;Проверка знака AX*  **jns** wtsds\_no\_sign *;Если >= 0, преобразуем как беззнаковое*  **mov** **byte**[**di**],'-' *;Добавление знака в начало строки*  **inc** **di** *;Инкремент DI*  **neg** **ax** *;Изменение знака значения AX*  wtsds\_no\_sign:  **call** word\_to\_udec\_str *;Преобразование беззнакового значения*  **pop** **ax**  **ret** | |
| *;Процедура преобразования байта в строку в десятичном виде (со знаком)*  *; AL - байт.*  *; DI - буфер для строки (4 символа). Значение регистра не сохраняется.*  byte\_to\_sdec\_str:  **push** **ax**  **movsx** **ax**,**al** *;Преобразование байта в слово (со знаком)*  **call** word\_to\_sdec\_str *;Вызов процедуры для слова со знаком*  **pop** **ax**  **ret** |

Полный исходный код примера: printdec.asm. Результат работы программы:



**Вывод чисел в восьмеричном виде**

Выводить числа в восьмеричном виде приходится достаточно редко, поэтому подробно рассматривать программу мы не будем. Можно либо делить число последовательно на 8, либо преобразовывать в цифры группы по 3 бита. Советуем использовать второй вариант. Смотрите код примера в файле: printoct.asm. Результат работы программы:



**Вывод чисел в других системах счисления**

Реализуется также, как вывод в десятичном виде — с помощью алгоритма последовательного деления на основание системы счисления. Например, если нужно вывести число в пятеричной системе счисления, делить надо на 5, а не на 10.

**Упражнение**

Напишите программу для вывода на консоль массива слов со знаком в десятичном виде (например, через запятую). Для вывода чисел можете воспользоваться моим примером или написать свою собственную процедуру.

**Ещё раз напомним ссылки на файлы с примерами**

* printbin.asm — вывод чисел на консоль в двоичном виде
* printoct.asm — вывод чисел на консоль в восьмеричном виде
* printdec.asm — вывод чисел на консоль в десятичном виде (со знаком и без знака)
* printhex.asm — вывод чисел на консоль в шестнадцатеричном виде

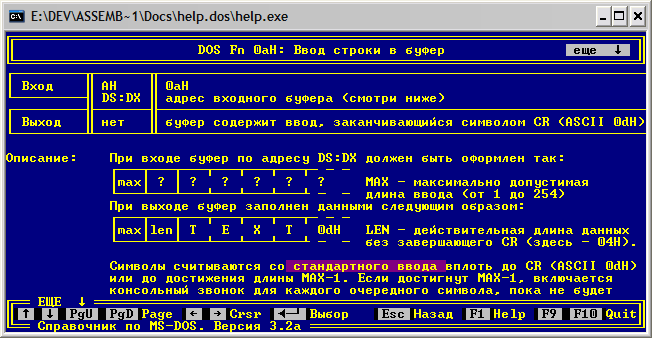
**Часть 23. Ввод чисел с консоли**

В прошлой части мы научились преобразовывать числа в строку и выводить на консоль. А в этой лекции решим обратную задачу — ввод чисел с консоли и преобразованием строки в число. Поскольку ввод в двоичном и восьмеричном виде используется редко, рассмотрим только примеры ввода чисел в десятичном виде (со знаком и без знака) и в шестнадцатеричном.

Вводить числа сложнее, чем выводить, так как помимо преобразования необходимо проверять корректность введённой пользователем строки. Хорошая программа должна устойчиво работать при любых входных данных (в том числе специально введённых так, чтобы нарушить её работу).

**Ввод строки с консоли**

Для ввода строки можно использовать функцию MS-DOS 0Ah. Функция позволяет ввести строку длиной от 1 до 254 символов. При вызове в DX передаётся адрес буфера, первый байт которого должен содержать максимально допустимую длину строки. Длина считается вместе с символом конца строки CR (0dh). В результате работы функции во второй байт буфера записывается фактическая длина введённой строки (не считая символа CR). Начиная с третьего байта в буфер записываются символы строки. Подробнее о работе функции можно узнать в справочнике по DOS:



Чтобы удобнее было использовать эту функцию, можно написать небольшую процедуру. Например, такую:

|  |
| --- |
| *;Процедура ввода строки c консоли*  *; вход: AL - максимальная длина (с символом CR) (1-254)*  *; выход: AL - длина введённой строки (не считая символа CR)*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  input\_str:  **push** **cx** *;Сохранение СX*  **mov** **cx**,**ax** *;Сохранение AX в CX*  **mov** **ah**,0Ah *;Функция DOS 0Ah - ввод строки в буфер*  **mov** [buffer],**al** *;Запись максимальной длины в первый байт буфера*  **mov** **byte**[buffer+1],0 *;Обнуление второго байта (фактической длины)*  **mov** **dx**,buffer *;DX = aдрес буфера*  **int** 21h *;Обращение к функции DOS*  **mov** **al**,[buffer+1] *;AL = длина введённой строки*  **add** **dx**,2 *;DX = адрес строки*  **mov** **ah**,**ch** *;Восстановление AH*  **pop** **cx** *;Восстановление CX*  **ret**  ...  buffer rb 256 |

Процедура использует отдельно объявленный буфер. В качестве единственного параметра ей передаётся максимальная длина строки в регистре AL. После возврата из процедуры в этот регистр записывается фактическая длина строки, а в регистр DX — адрес начала строки. Старшая часть AX сохраняется.

**Ввод десятичных чисел без знака**

Для преобразования числа в строку используется так называемая схема Горнера. Любое число в десятичной системе можно представить в следующем виде:

34710 = 3·102 + 4·101 + 7·100 = (3·10 + 4)·10 + 7

Это очень удобно запрограммировать в цикле: результат сначала умножается на 10, а потом к нему прибавляется очередная цифра и так далее для всех цифр. Кстати, это верно не только для десятичной системы счисления, но и для других, разница только в множителе, который должен быть равен основанию системы счисления.

Следующая процедура преобразует строку в слово в регистре AX. Адрес строки передаётся в DX, длина строки передаётся в AL. Если строка не корректна, процедура возвращает 0 и устанавливает флаг CF. Ошибка возвращается в следующих случаях:

* строка имеет нулевую длину, то есть пустая строка;
* строка содержит любые символы кроме десятичных цифр;
* число находится вне границ диапазона представления чисел (для слова без знака 0…65535).

|  |
| --- |
| *;Процедура преобразования десятичной строки в слово без знака*  *; вход: AL - длина строки*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  *; выход: AX - слово (в случае ошибки AX = 0)*  *; CF = 1 - ошибка*  str\_to\_udec\_word:  **push** **cx** *;Сохранение всех используемых регистров*  **push** **dx**  **push** **bx**  **push** **si**  **push** **di**    **mov** **si**,**dx** *;SI = адрес строки*  **mov** **di**,10 *;DI = множитель 10 (основание системы счисления)*  **movzx** **cx**,**al** *;CX = счётчик цикла = длина строки*  **jcxz** studw\_error *;Если длина = 0, возвращаем ошибку*  **xor** **ax**,**ax** *;AX = 0*  **xor** **bx**,**bx** *;BX = 0*    studw\_lp:  **mov** **bl**,[**si**] *;Загрузка в BL очередного символа строки*  **inc** **si** *;Инкремент адреса*  **cmp** **bl**,'0' *;Если код символа меньше кода '0'*  **jl** studw\_error *; возвращаем ошибку*  **cmp** **bl**,'9' *;Если код символа больше кода '9'*  **jg** studw\_error *; возвращаем ошибку*  **sub** **bl**,'0' *;Преобразование символа-цифры в число*  **mul** **di** *;AX = AX \* 10*  **jc** studw\_error *;Если результат больше 16 бит - ошибка*  **add** **ax**,**bx** *;Прибавляем цифру*  **jc** studw\_error *;Если переполнение - ошибка*  **loop** studw\_lp *;Команда цикла*  **jmp** studw\_exit *;Успешное завершение (здесь всегда CF = 0)*    studw\_error:  **xor** **ax**,**ax** *;AX = 0*  **stc** *;CF = 1 (Возвращаем ошибку)*    studw\_exit:  **pop** **di** *;Восстановление регистров*  **pop** **si**  **pop** **bx**  **pop** **dx**  **pop** **cx**  **ret** |

Для установки флага CF используется команда STC. Сбросить флаг CF можно командой CLC. В коде данной процедуры она не используется, так как в случае успешного завершения цикла флаг CF всегда будет равен 0.

На основе этой процедуры несложно написать ещё одну для ввода чисел размером 1 байт. Сначала строка преобразуется в слово без знака, а затем выполняется проверка старшей части на равенство нулю. Обратите внимание, что команда TEST всегда сбрасывает флаг CF.

|  |
| --- |
| *;Процедура преобразования десятичной строки в байт без знака*  *; вход: AL - длина строки*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  *; выход: AL - байт (в случае ошибки AL = 0)*  *; CF = 1 - ошибка*  str\_to\_udec\_byte:  **push** **dx** *;Сохранение регистров*  **push** **ax**  **call** str\_to\_udec\_word *;Преобразование строки в слово (без знака)*  **jc** studb\_exit *;Если ошибка, то возвращаем ошибку*  **test** **ah**,**ah** *;Проверка старшего байта AX*  **jz** studb\_exit *;Если 0, то выход из процедуры (здесь всегда CF = 0)*  **xor** **al**,**al** *;AL = 0*  **stc** *;CF = 1 (Возвращаем ошибку)*  studb\_exit:  **pop** **dx**  **mov** **ah**,**dh** *;Восстановление только старшей части AX*  **pop** **dx**  **ret** |

Следующие две процедуры совмещают ввод строки с преобразованием строки в число. Для слова нужно ввести максимум 5 символов, а для байта — максимум 3.

|  |  |
| --- | --- |
| *;Процедура ввода слова с консоли в десятичном виде (без знака)*  *; выход: AX - слово (в случае ошибки AX = 0)*  *; CF = 1 - ошибка*  input\_udec\_word:  **push** **dx** *;Сохранение DX*  **mov** **al**,6 *;Ввод максимум 5 символов (65535) + конец строки*  **call** input\_str *;Вызов процедуры ввода строки*  **call** str\_to\_udec\_word *;Преобразование строки в слово (без знака)*  **pop** **dx** *;Восстановление DX*  **ret** | |
| *;Процедура ввода байта с консоли в десятичном виде (без знака)*  *; выход: AL - байт (в случае ошибки AL = 0)*  *; CF = 1 - ошибка*  input\_udec\_byte:  **push** **dx** *;Сохранение DX*  **mov** **al**,4 *;Ввод максимум 3 символов (255) + конец строки*  **call** input\_str *;Вызов процедуры ввода строки*  **call** str\_to\_udec\_byte *;Преобразование строки в байт (без знака)*  **pop** **dx** *;Восстановление DX*  **ret** |

**Ввод десятичных чисел со знаком**

Ввод чисел со знаком немного труднее. Необходимо проверить первый символ строки: если это символ ‘-‘, то число отрицательное. Кроме того, нужно внимательно проверить диапазон представления (для слова со знаком -32768…32767). Строку без символа ‘-‘ можно преобразовать процедурой для беззнакового числа.

|  |
| --- |
| *;Процедура преобразования десятичной строки в слово со знаком*  *; вход: AL - длина строки*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  *; выход: AX - слово (в случае ошибки AX = 0)*  *; CF = 1 - ошибка*  str\_to\_sdec\_word:  **push** **bx** *;Сохранение регистров*  **push** **dx**    **test** **al**,**al** *;Проверка длины строки*  **jz** stsdw\_error *;Если равно 0, возвращаем ошибку*  **mov** **bx**,**dx** *;BX = адрес строки*  **mov** **bl**,[**bx**] *;BL = первый символ строки*  **cmp** **bl**,'-' *;Сравнение первого символа с '-'*  **jne** stsdw\_no\_sign *;Если не равно, то преобразуем как число без знака*  **inc** **dx** *;Инкремент адреса строки*  **dec** **al** *;Декремент длины строки*  stsdw\_no\_sign:  **call** str\_to\_udec\_word *;Преобразуем строку в слово без знака*  **jc** stsdw\_exit *;Если ошибка, то возвращаем ошибку*  **cmp** **bl**,'-' *;Снова проверяем знак*  **jne** stsdw\_plus *;Если первый символ не '-', то число положительное*  **cmp** **ax**,32768 *;Модуль отрицательного числа должен быть не больше 32768*  **ja** stsdw\_error *;Если больше (без знака), возвращаем ошибку*  **neg** **ax** *;Инвертируем число*  **jmp** stsdw\_ok *;Переход к нормальному завершению процедуры*  stsdw\_plus:  **cmp** **ax**,32767 *;Положительное число должно быть не больше 32767*  **ja** stsdw\_error *;Если больше (без знака), возвращаем ошибку*    stsdw\_ok:  **clc** *;CF = 0*  **jmp** stsdw\_exit *;Переход к выходу из процедуры*  stsdw\_error:  **xor** **ax**,**ax** *;AX = 0*  **stc** *;CF = 1 (Возвращаем ошибку*  stsdw\_exit:  **pop** **dx** *;Восстановление регистров*  **pop** **bx**  **ret** |

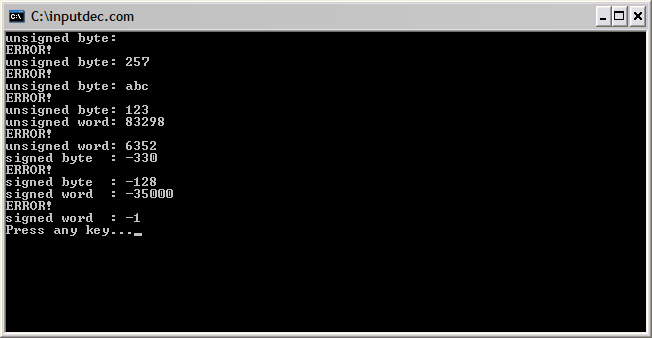
Обратите внимание, что для перехода после проверки диапазона используется команда JA (как для сравнения чисел без знака). Ввод байта со знаком реализуется с помощью той же процедуры и дополнительной проверки диапазона значения.

|  |
| --- |
| *;Процедура преобразования десятичной строки в байт со знаком*  *; вход: AL - длина строки*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  *; выход: AL - байт (в случае ошибки AL = 0)*  *; CF = 1 - ошибка*  str\_to\_sdec\_byte:  **push** **dx** *;Сохранение регистров*  **push** **ax**  **call** str\_to\_sdec\_word *;Преобразование строки в слово (со знаком)*  **jc** stsdb\_exit *;Если ошибка, то возвращаем ошибку*  **cmp** **ax**,127 *;Сравнение результата с 127*  **jg** stsdb\_error *;Если больше - ошибка*  **cmp** **ax**,-128 *;Сравнение результата с -128*  **jl** stsdb\_error *;Если меньше - ошибка*  **clc** *;CF = 0*  **jmp** studb\_exit *;Переход к выходу из процедуры*  stsdb\_error:  **xor** **al**,**al** *;AL = 0*  **stc** *;CF = 1 (Возвращаем ошибку)*  stsdb\_exit:  **pop** **dx**  **mov** **ah**,**dh** *;Восстановление только старшей части AX*  **pop** **dx**  **ret** |

Наконец, ещё две процедуры совмещают ввод строки с преобразованием её в слово и байт со знаком.

|  |  |
| --- | --- |
| *;Процедура ввода слова с консоли в десятичном виде (со знаком)*  *; выход: AX - слово (в случае ошибки AX = 0)*  *; CF = 1 - ошибка*  input\_sdec\_word:  **push** **dx** *;Сохранение DX*  **mov** **al**,7 *;Ввод максимум 7 символов (-32768) + конец строки*  **call** input\_str *;Вызов процедуры ввода строки*  **call** str\_to\_sdec\_word *;Преобразование строки в слово (со знаком)*  **pop** **dx** *;Восстановление DX*  **ret** | |
| *;Процедура ввода байта с консоли в десятичном виде (со знаком)*  *; выход: AL - байт (в случае ошибки AL = 0)*  *; CF = 1 - ошибка*  input\_sdec\_byte:  **push** **dx** *;Сохранение DX*  **mov** **al**,5 *;Ввод максимум 3 символов (-128) + конец строки*  **call** input\_str *;Вызов процедуры ввода строки*  **call** str\_to\_sdec\_byte *;Преобразование строки в байт (со знаком)*  **pop** **dx** *;Восстановление DX*  **ret** |

Полный исходный код примера вы можете загрузить из файла: inputdec.asm. В случае некорректной строки программа выводит сообщение об ошибке и повторяет запрос ввода числа:



**Ввод шестнадцатеричных чисел**

Преобразование шестнадцатеричной строки в число несколько проще. Удобно реализовать в виде отдельной процедуры преобразование одной цифры. Процедура воспринимает символы ‘A’-‘F’ независимо от регистра. Так как перед вычитанием выполняются проверки, флаг CF всегда будет равен нулю после успешного преобразования.

|  |
| --- |
| *;Процедура преобразования шестнадцатеричной цифры в число*  *; вход: DL - символ-цифра*  *; выход: DL - значение цифры (0-15, в случае ошибки DL = 0)*  *; CF = 1 - ошибка*  convert\_hex\_digit:  **cmp** **dl**,'0' *;Сравнение с символом '0'*  **jl** chd\_error *;Если меньше, возвращаем ошибку*  **cmp** **dl**,'9' *;Сравнение с символом '9'*  **jg** chd\_a\_f *;Если больше, то возможно это буква a-f или A-F*  **sub** **dl**,'0' *;Преобразование цифры в число*  **ret** *;Возврат из процедуры (здесь всегда CF = 0)*    chd\_a\_f:  **and** **dl**,11011111b *;Преобразование буквы в верхний регистр*  **cmp** **dl**,'A' *;Сравнение с символом 'A'*  **jl** chd\_error *;Если меньше, возвращаем ошибку*  **cmp** **dl**,'F' *;Сравнение с символом 'F'*  **jg** chd\_error *;Если больше, возвращаем ошибку*  **sub** **dl**,'A'-10 *;Преобразуем букву в число*  **ret** *;Возврат из процедуры (здесь тоже всегда CF = 0)*    chd\_error:  **xor** **dl**,**dl** *;DL = 0*  **stc** *;CF = 1*  **ret** *;Возврат из процедуры* |

Теперь легко можно написать преобразование шестнадцатеричной строки в слово. Вместо умножения на 16 в процедуре используется сдвиг на 4 бита влево, а вместо сложения — операция ИЛИ. Проверки диапазона значения не нужны, достаточно проверить длину строки и преобразовать цифры.

|  |
| --- |
| *;Процедура преобразования шестнадцатеричной строки в слово*  *; вход: AL - длина строки*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  *; выход: AX - слово (в случае ошибки AX = 0)*  *; CF = 1 - ошибка*  str\_to\_hex\_word:  **push** **cx** *;Сохранение регистров*  **push** **dx**  **push** **si**    **movzx** **cx**,**al** *;CX = счётчик цикла = длина строки*  **jcxz** sthw\_error *;Если длина строки = 0, возвращаем ошибку*  **cmp** **cx**,4  **jg** sthw\_error *;Если длина строки больше 4, возвращаем ошибку*  **xor** **ax**,**ax** *;AX = 0*  **mov** **si**,**dx** *;SI = адрес строки*    sthw\_lp:  **mov** **dl**,[**si**] *;Загрузка в DL очередного символа строки*  **inc** **si** *;Инкремент адреса строки*  **call** convert\_hex\_digit *;Преобразование шестнадцатеричной цифры в число*  **jc** sthw\_error *;Если ошибка, то возвращаем ошибку*  **shl** **ax**,4 *;Сдвиг AX на 4 бита влево*  **or** **al**,**dl** *;Добавление преобразованной цифры*  **loop** sthw\_lp *;Команда цикла*  **jmp** sthw\_exit *;CF = 0*    sthw\_error:  **xor** **ax**,**ax** *;AX = 0*  **stc** *;CF = 1*    sthw\_exit:  **pop** **si** *;Восстановление регистров*  **pop** **dx**  **pop** **cx**  **ret** |

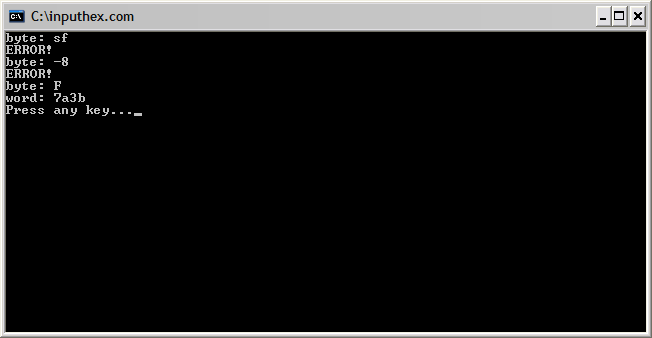
Для ввода байта используется та же процедура, но дополнительно проверяется длина строки — она должна быть не больше 2.

|  |
| --- |
| *;Процедура преобразования шестнадцатеричной строки в байт*  *; вход: AL - длина строки*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  *; выход: AL - байт (в случае ошибки AL = 0)*  *; CF = 1 - ошибка*  str\_to\_hex\_byte:  **push** **cx** *;Сохранение CX*  **mov** **cx**,**ax** *;Сохранение AX в CX*  **cmp** **al**,2 *;Проверка длины строки*  **jg** sthb\_error *;Если больше 2, возвращаем ошибку*  **call** str\_to\_hex\_word *;Преобразование строки в слово*  **jnc** sthb\_exit *;Если нет ошибки, то переход к выходу из процедуры*  sthb\_error:  **stc** *;CF = 1*  sthb\_exit:  **mov** **ah**,**ch** *;Восстановление AH*  **pop** **cx** *;Восстановление CX*  **ret** |

Ещё две процедуры для ввода и преобразования строки, также как для десятичного ввода:

|  |  |
| --- | --- |
| *;Процедура ввода слова с консоли в шестнадцатеричном виде*  *; выход: AX - слово (в случае ошибки AX = 0)*  *; CF = 1 - ошибка*  input\_hex\_word:  **push** **dx** *;Сохранение DX*  **mov** **al**,5 *;Ввод максимум 4 символов (FFFF) + конец строки*  **call** input\_str *;Вызов процедуры ввода строки*  **call** str\_to\_hex\_word *;Преобразование строки в слово*  **pop** **dx** *;Восстановление DX*  **ret** | |
| *;Процедура ввода байта с консоли в шестнадцатеричном виде*  *; выход: AL - байт (в случае ошибки AL = 0)*  *; CF = 1 - ошибка*  input\_hex\_byte:  **push** **dx** *;Сохранение DX*  **mov** **al**,3 *;Ввод максимум 2 символов (FF) + конец строки*  **call** input\_str *;Вызов процедуры ввода строки*  **call** str\_to\_hex\_byte *;Преобразование строки в байт*  **pop** **dx** *;Восстановление DX*  **ret** |

Полный исходный код примера: inputhex.asm. Как и в примере с десятичными числами, программа повторяет запрос ввода, пока не будут введены корректные данные:



**Упражнение**

Напишите программу для ввода байта с консоли в двоичном виде. Желательно с проверкой корректности ввода

**Ещё раз напомним ссылки на файлы с примерами:**

* inputdec.asm — ввод десятичных чисел с консоли (со знаком и без)
* inputhex.asm — ввод шестнадцатеричных чисел с консоли

**Часть 24. Команды управления флагами**

В предыдущей части учебного курса мы использовали флаг CF, чтобы вернуть из процедуры информацию об ошибке. Чтобы у вас сложилась полная картина, рассмотрим команды управления флагами.

Как вы помните флаги изменяются в результате выполнения арифметических и логических команд, а также команд сдвига. Регистр флагов можно сохранить в стек с помощью команды PUSHF и восстановить из стека с помощью команды POPF. Кроме того, в процессоре существуют специальные команды, которые позволяют явно установить или сбросить флаги CF, DF и IF. Это очень простые команды: у них нет операндов и результатом является только изменение значения соответствующего флага.

**Флаг переноса CF**

Команда CLC сбрасывает флаг CF.

Команда STC устанавливает флаг CF в единицу.

Команда CMC инвертирует значение флага CF.

**Флаг направления DF**

Этот флаг определяет направление обработки данных цепочечными командами. Он должен устанавливаться или сбрасываться перед использованием этих команд.

Команда CLD сбрасывает флаг DF.

Команда STD устанавливает флаг DF в единицу.

**Флаг прерывания IF**

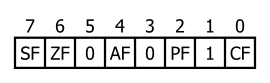
Этот флаг определяет, разрешены в данный момент прерывания или нет.

Команда CLI сбрасывает флаг IF (запрещает прерывания).

Команда STI устанавливает флаг IF в единицу (разрешает прерывания).

**Команды LAHF и SAHF**

Команда LAHF загружает младший байт регистра флагов в AH. Её удобно использовать, когда нужно получить значения сразу нескольких флагов. Порядок расположения флагов представлен на рисунке:



Команда SAHF выполняет обратную операцию — загружает содержимое AH в младший байт регистра флагов. Это позволяет одновременно изменить значения нескольких флагов. При этом биты 1, 3, 5 регистра AH игнорируются.

**Пример программы**

В качестве примера использования команды LAHF напишем процедуру, которая получает и выводит на консоль значения флагов SF, ZF, AF, PF и CF. Подобную процедуру можно использовать в отладочных целях. Достаточно добавить её вызов в то место программы, где хочется проверить состояние флагов.

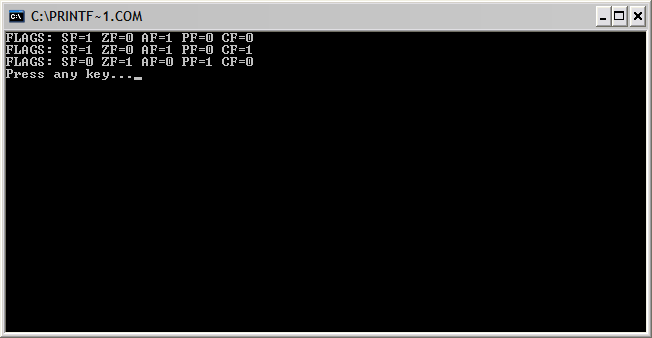
Чтобы выводить значения отдельных битов удобно написать отдельную процедурку для печати флага CF в виде символа. Нужный бит будет помещаться в флаг CF при помощи команды сдвига SHL.

|  |
| --- |
| *;Процедура вывода значения флага CF в виде символа*  print\_cf:  **push** **ax**  **push** **dx**  **mov** **ah**,2 *;Функция DOS 02h - вывод символа*  **mov** **dl**,'0' *;DL = '0'*  **adc** **dl**,0 *;Если CF = 1, то в DL будет символ '1'*  **int** 21h *;Обращение к функции DOS*  **pop** **dx**  **pop** **ax**  **ret** |

Процедура вывода флагов сохраняет регистр флагов и восстанавливает его перед возвратом управления, поэтому можно вызывать её где угодно, не волнуясь, что она нарушит выполнение основной программы.

|  |
| --- |
| *;Процедура вывода состояния флагов на консоль*  print\_flags:  **push** **ax**  **push** **cx**  **push** **dx**  **pushf** *;Сохранение регистра флагов*  **lahf** *;Загрузка младшего байта FLAGS в AH*  **mov** **cl**,**ah** *;CL = AH*  **mov** **ah**,9 *;Функция DOS 09h - вывод строки*  **mov** **dx**,s\_sf *;DX = адрес строки 'FLAGS: SF='*  **int** 21h *;Обращение к функции DOS*  **shl** **cl**,1 *;Сдвиг CL влево на 1 бит*  **call** print\_cf *;Печать выдвинутого бита*  **mov** **dx**,s\_zf  **int** 21h *;Вывод строки ' ZF='*  **shl** **cl**,1 *;Сдвиг CL влево на 1 бит*  **call** print\_cf *;Печать выдвинутого бита*  **mov** **dx**,s\_af  **int** 21h *;Вывод строки ' AF='*  **shl** **cl**,2 *;Сдвиг CL влево на 2 бита*  **call** print\_cf *;Печать выдвинутого бита*  **mov** **dx**,s\_pf  **int** 21h *;Вывод строки ' PF='*  **shl** **cl**,2 *;Сдвиг CL влево на 2 бита*  **call** print\_cf *;Печать выдвинутого бита*  **mov** **dx**,s\_cf  **int** 21h *;Вывод строки ' CF='*  **shl** **cl**,2 *;Сдвиг CL влево на 2 бита*  **call** print\_cf *;Печать выдвинутого бита*  **mov** **dx**,s\_endl  **int** 21h *;Вывод конца строки*  **popf** *;Восстановление регистра флагов*  **pop** **dx**  **pop** **cx**  **pop** **ax**  **ret** |

Полный исходный код примера привед в файле printflags.asm. Вывод программы выглядит так:



**Упражнение**

Чтобы потренироваться в работе с флагами напишите следующую программу. Вычислите сумму значений флагов CF, SF, ZF и выведите на консоль результат сложения. Если результат больше 1, инвертируйте значение флага CF :)

**Часть 25. Передача параметров через стек**

В предыдущих частях учебного курса все параметры передавались процедурам через регистры. В этой части мы рассмотрим другой способ — передачу параметров через стек. Часто этот способ оказывается удобнее. Через регистры можно передать максимум 6-7 параметров, а через стек — сколько угодно. Кроме того можно написать процедуру с переменным количеством параметров.

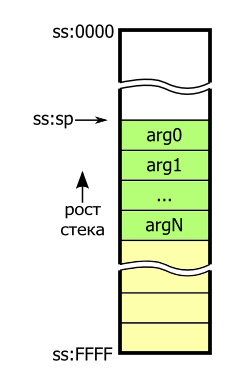
Если через регистры передаётся больше 2-3 параметров, то приходится сохранять регистры внутри процедуры и опять же использовать стек. С другой стороны, обращение к параметрам в стеке происходит медленнее. Если вы оптимизируете программу по скорости выполнения, то имеет смысл передавать параметры через регистры.

**Помещение параметров в стек**

Перед вызовом процедуры параметры необходимо поместить в стек с помощью команды PUSH. Здесь существует два варианта: параметры могут помещаться в стек в прямом или в обратном порядке. Обычно используется обратный порядок и мы будем его использовать в примерах. Параметры помещаются в стек, начиная с последнего, так что перед вызовом процедуры на вершине стека оказывается первый параметр:

|  |
| --- |
| *; Данные*  arg0 **dw** 0  arg1 **dw** 12  argN **dw** 345  *;---------------------------------------------------------------------*  *; Код*  **push** [argN]  **push** ...  **push** [arg1]  **push** [arg0]  **call** myproc |

Перед выполнением команды CALL стек будет иметь следующий вид:

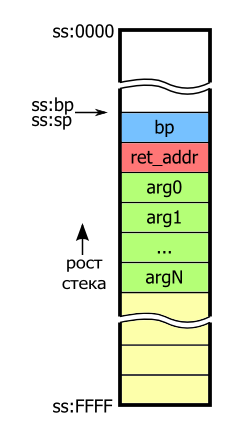


**Обращение к параметрам внутри процедуры**

Для обращения к параметрам внутри процедуры обычно используют регистр BP. В самом начале процедуры содержимое регистра BP сохраняется в стеке и в него копируется значение регистра SP. Это позволяет «запомнить» положение вершины стека и адресовать параметры относительно регистра BP.

|  |
| --- |
| *;Процедура*  myproc:  **push** **bp**  **mov** **bp**,**sp**  ... |

При выполнении кода процедуры стек будет иметь следующую структуру:



Здесь *ret\_addr* обозначает адрес возврата, помещаемый в стек командой вызова процедуры, а *bp* — сохранённое значение регистра BP. В нашем случае стек имеет ширину 16 бит, поэтому первый параметр будет доступен как word[bp+4], второй как word[bp+6] и так далее.

|  |
| --- |
| **mov** **ax**,[**bp**+4] *;AX = arg0*  **mov** **bx**,[**bp**+6] *;BX = arg1*  **add** **ax**,[**bp**+8] *;AX = AX + arg2* |

Полезно представлять себе стек, чтобы правильно указывать смещения относительно регистра BP. Не забудьте перед возвратом из процедуры восстановить значение BP из стека.

**Извлечение параметров из стека**

После того, как процедура выполнилась, необходимо очистить стек, вытолкнув из него параметры. Тут тоже существует 2 способа: стек может быть очищен самой процедурой или кодом, который эту процедуру вызывал. Для первого способа используется команда RET с одним операндом, который должен быть равен количеству байтов, выталкиваемых из стека. В нашем случае он должен быть равен количеству параметров, умноженному на 2.

|  |
| --- |
| **push** [arg1]  **push** [arg0]  **call** myproc  ...  *;----------------------------------------------------------------------*  *;Процедура c двумя параметрами*  myproc:  **push** **bp**  **mov** **bp**,**sp**  ...  **pop** **bp**  **ret** 4 *;Из стека дополнительно извлекается 4 байта* |

Для второго способа нужно использовать команду RET без операндов. Стек восстанавливается после выполнения процедуры путём прибавления значения к SP. С помощью такого способа программируются процедуры с переменным количеством параметров. Процедура не знает, сколько ей будет передано параметров, поэтому очистка стека должна выполняться вызывающим кодом.

|  |
| --- |
| **push** [arg1]  **push** [arg0]  **call** myproc2  **add** **sp**,4 *;Восстановление указателя стека*  ...  *;----------------------------------------------------------------------*  *;Процедура с двумя параметрами (не очищает стек)*  myproc2:  **push** **bp**  **mov** **bp**,**sp**  ...  **pop** **bp**  **ret** |

**Соглашения вызова**

Совокупность таких особенностей, как способ и порядок передачи параметров, механизм очистки стека, сохранение определённых регистров в процедуре и некоторых других называется *соглашениями вызова*. Соблюдение этих соглашений является важным, если вы из своей программы обращаетесь к компонентам, написанным на других языках программирования, вызываете функции ОС, или хотите из других языков вызывать процедуры, написанные на ассемблере. В остальных случаях процедуры на ассемблере можете писать так, как вам удобнее.

**Пример**

В качестве примера рассмотрим процедуру с тремя параметрами: *a*, *b* и *c*. Процедура вычисляет значение выражения (*a*+*b*)/*c*. Параметры передаются через стек в обратном порядке, результат возвращается в регистре AX, стек восстанавливается вызываемой процедурой. Все числа — 16-битные целые со знаком.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*  **jmp** **start** *;Переход на метку start*  *;---------------------------------------------------------------------*  *; Данные*  a **dw** 81  b **dw** 273  x **dw** ?  *;---------------------------------------------------------------------*  **start**:  **push** 3 *;c=3*  **push** [b] *;b*  **push** [a] *;a*  **call** primer *;Вызов процедуры*  **mov** [x],**ax** *;x=(a+b)/c*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*    *;---------------------------------------------------------------------*  *;Процедура c тремя параметрами: a, b, c.*  *;Вычисляет значение выражения (a+b)/c. Результат возвращается в AX.*  primer:  **push** **bp** *;Сохранение регистра BP*  **mov** **bp**,**sp** *;BP=SP*  **push** **dx**    **mov** **ax**,[**bp**+4] *;AX=a*  **add** **ax**,[**bp**+6] *;AX=(a+b)*  **cwd** *;DX:AX=(a+b)*  **idiv** **word**[**bp**+8] *;AX=(a+b)/c*    **pop** **dx**  **pop** **bp** *;Восстановление регистра BP*  **ret** 6 *;Возврат с извлечением параметров из стека* |

**Упражнение**

Напишите любую процедуру с 4-5 параметрами, передаваемыми через стек. Вызовите процедуру в своей программе. Запустите программу в Turbo Debugger и посмотрите, как происходит работа со стеком.

**Часть 26. Локальные переменные**

До этой части учебного курса все переменные в наших программах были только *глобальными* — они создавались и инициализировались при запуске программы и к ним можно было обратиться из любой её части.

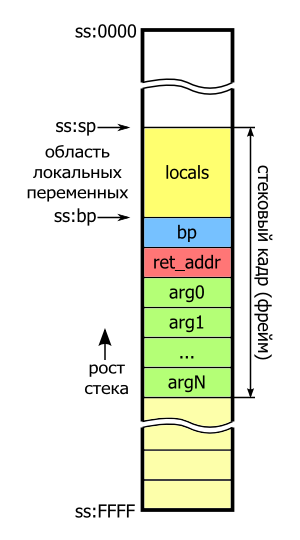
*Локальные переменные* используются для хранения промежуточных результатов во время выполнения процедуры. В отличие от глобальных, эти переменные являются временными и создаются при запуске процедуры. Для локальных переменных существует понятие *области видимости* — так называется область программы, в которой доступна переменная. Обычно в ассемблере область видимости ограничена процедурой, создавшей локальную переменную.

**Создание локальных переменных**

Чтобы создать локальные переменные в процедуре, необходимо выделить для них память. Эта память выделяется в стеке. Сделать это просто — достаточно вычесть из регистра SP значение, равное суммарному размеру всех локальных переменных в процедуре. Так как ширина стека равна 16 бит, то это значение должно быть кратно 2 байтам. При выходе из процедуры нужно восстановить указатель стека. Обычно это выполняется командой mov sp,bp (В bp сохраняется значение sp при входе в процедуру, как в случае с параметрами, передаваемыми через стек). Код процедуры с локальными переменными будет выглядеть следующим образом:

|  |
| --- |
| *;Процедура с локальными переменными*  myproc:  **push** **bp** *;Сохранение BP*  **mov** **bp**,**sp** *;Копирование указателя стека в BP*  **sub** **sp**,locals\_size *;Выделение памяти для локальных переменных*  ...  **mov** **sp**,**bp** *;Восстановление указателя стека*  **pop** **bp** *;Восстановление BP*  **ret** *;Возврат из процедуры* |

Код, выполняемый при входе в процедуру, называют также кодом *пролога*, а код, выполняемый при выходе, — кодом *эпилога*. После выполнения кода пролога стек будет иметь такой вид:



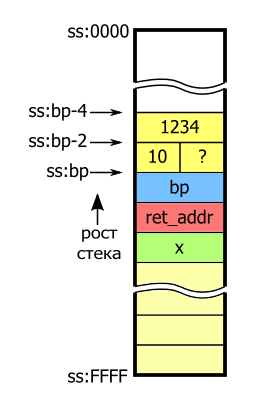
Область стека, включающая в себя параметры процедуры, адрес возврата, локальные переменные и сохранённые регистры, называется *кадром* или *фреймом* стека. Из рисунка понятно, что для обращения к локальным переменным внутри процедуры нужно использовать отрицательные смещения относительно регистра BP. Рассмотрим это на примере.

**Пример процедуры с локальными переменными**

Пример достаточно простой. Внутри процедуры создаются 2 локальных переменных: одна размером слово и вторая размером байт. Чтобы не нарушить стек, нужно выделить 4 байта (один байт не будет использоваться). Процедуре будет передаваться через стек один параметр — *x* (слово без знака). Она будет вычислять выражение 1234+*x*/10 и возвращать его в регистре AX. Числа 1234 и 10 будут нашими локальными переменными. Конечно, в данном случае можно обойтись и без локальных переменных, они здесь только для примера.

|  |
| --- |
| *;Пример процедуры с двумя локальными переменными*  *;вход: параметр x в стеке*  *;выход: AX = вычисленное значение*  simpleproc:  **push** **bp** *;Сохранение BP*  **mov** **bp**,**sp** *;Копирование указателя стека в BP*  **sub** **sp**,4 *;Выделение 4 байт для локальных переменных*    **mov** **word**[**bp**-4],1234 *;Инициализация первой локальной переменной*  **mov** **byte**[**bp**-2],10 *;Инициализация второй локальной переменной*    **mov** **ax**,[**bp**+4] *;AX = x*  **div** **byte**[**bp**-2] *;AL = x/10*  **xor** **ah**,**ah** *;AX = x/10*  **add** **ax**,[**bp**-4] *;AX = 1234+x/10*    **mov** **sp**,**bp** *;Восстановление указателя стека*  **pop** **bp** *;Восстановление BP*  **ret** 2 *;Возврат из процедуры* |

После выполнения кода пролога первая локальная переменная будет находиться по адресу bp-4, а вторая по адресу bp-2. Обратите внимание, что стековые переменные должны быть явно инициализированы. Так как память выделяется в стеке, то изначально в них будет всякий мусор (а вовсе не нули). Структура стека при выполнении процедуры показана на рисунке:



**Команды ENTER и LEAVE**

В системе команд процессоров x86 существуют также специальные команды для работы с кадром стека процедуры: ENTER и LEAVE. Команда ENTER обычно размещается в начале процедуры. У неё два непосредственных операнда: первый операнд — размер памяти, выделяемой под локальные переменные, второй операнд — уровень вложенности. в нашем случае второй операнд будет равен 0. Тогда по действию команда будет аналогична трём следующим командам:

|  |
| --- |
| **push** **bp** *;\*  **mov** **bp**,**sp** *; > или enter locals\_size,0*  **sub** **sp**,locals\_size *;/* |

Второй операнд этой команды позволяет организовывать вложенные области видимости, как в некоторых языках высокого уровня. В ассемблере эти возможности используются редко.

Команда LEAVE не имеет операндов и аналогична по действию двум командам:

|  |
| --- |
| **mov** **sp**,**bp** *;\*  **pop** **bp** *;/ или leave* |

А так будет выглядеть наша процедура, если использовать команды ENTER и LEAVE:

|  |
| --- |
| *;Пример процедуры с двумя локальными переменными*  *;вход: параметр x в стеке*  *;выход: AX = вычисленное значение*  simpleproc2:  **enter** 4,0 *;Создание кадра стека*    **mov** **word**[**bp**-4],1234 *;Инициализация первой локальной переменной*  **mov** **byte**[**bp**-2],10 *;Инициализация второй локальной переменной*    **mov** **ax**,[**bp**+4] *;AX = x*  **div** **byte**[**bp**-2] *;AL = x/10*  **xor** **ah**,**ah** *;AX = x/10*  **add** **ax**,[**bp**-4] *;AX = 1234+x/10*    **leave** *;Освобождение памяти, восстановление BP*  **ret** 2 *;Возврат из процедуры* |

Вроде кажется, что проще, но компиляторы такой вариант практически не используют. Дело в том, что команда ENTER на современных процессорах выполняется гораздо медленнее, чем пролог из 3-х команд. Самый быстрый вариант такой:

|  |
| --- |
| *;Процедура с локальными переменными*  myproc:  **push** **bp** *;Сохранение BP*  **mov** **bp**,**sp** *;Копирование указателя стека в BP*  **sub** **sp**,locals\_size *;Выделение памяти для локальных переменных*  ...  **leave** *;Освобождение памяти, восстановление BP*  **ret** *;Возврат из процедуры* |

**Упражнение**

Напишите улучшенную процедуру для ввода десятичного числа (от 0 до 255) с консоли. Буфер для ввода строки должен быть локальным, то есть выделяться в стеке при вызове процедуры. Процедура должна вызываться без параметров и возвращать введённое число в регистре AL. Ваши результаты, а также вопросы можете писать в комментариях или на форуме.

**Часть 27. Синтаксис объявления меток**

Объявлять метки вы уже научились. Однако, синтаксис FASM не ограничивается объявлением простых меток. В этой части мы рассмотрим дополнительную директиву для создания меток, а также научимся использовать локальные и анонимные метки.

В синтаксисе FASM существует 3 основных способа объявления меток:

**1. Имя метки, после которого ставится двоеточие**. Это самый простой способ. Обычно так объявляются метки в коде.

|  |
| --- |
| exit\_app:  **mov** **ax**,4C00h  **int** 21h |

**2. Использование директив объявления данных**. Имя переменной является по сути той же меткой. Отличие от первого способа в том, что дополнительно с именем метки связывается размер переменной.

|  |
| --- |
| x **db** 5  y **dw** 34,1200,?  z rd 1 |

**3. Объявление метки с помощью специальной директивы *label***. Более сложный, но зато самый гибкий способ. Его мы рассмотрим подробнее.

Директива *label* имеет следующий формат:

|  |
| --- |
| label <имя\_метки> [размер] [**at** адрес] |

У директивы может быть 3 параметра. Обязательным является только первый параметр — имя метки. Второй параметр — оператор размера (*byte*, *word*, *dword* и т.д.). Он связывает с меткой размер переменной, аналогично тому, как это делают директивы объявления данных. Далее может быть указан оператор *at* и адрес метки. Адрес может представлять собой константу, числовое выражение или имя другой метки. Если адрес не указан, то для создания метки используется адрес того места, где она объявлена.

|  |
| --- |
| label m1 *;То же самое, что 'm1:'*  label m2 **byte** *;Похоже на 'm2 db ?', но память не резервируется*  label m3 **dword** *;Похоже на 'm3 dd ?', но память не резервируется*  *;Все 3 метки указывают на один и тот же адрес!* |

Следующий пример показывает, как можно использовать в программе директиву *label*. Допустим, объявлена переменная *x* размером слово. Требуется обнулить старший байт *x*. Воспользовавшись директивой *label*, можно обратиться к старшему байту как к отдельной переменной:

|  |
| --- |
| x **dw** 12345 *;Переменная-слово*  label xh **byte** **at** x+1 *;Объявление метки для обращения к старшему байту*  ...  **start**:  **xor** **al**,**al** *;AL=0*  **mov** [xh],**al** *;xh=0 (старший байт x)*  **mov** **byte**[x+1],**al** *;То же самое без использования метки xh* |

Кроме того, адрес может содержать базовые и индексные регистры для косвенной адресации. Такие метки можно использовать для обращения к параметрам и локальным переменным процедуры. Например:

|  |
| --- |
| label i **word** **at** **bp**-2 *;Локальная переменная*  ...  **inc** [i] *;Инкремент локальной переменной* |

**Локальные метки**

Локальная метка — это метка, имя которой начинается с точки. Во время генерации кода FASM автоматически добавляет к имени локальной метки имя последней объявленной «глобальной» метки. Таким образом, имена локальных меток могут повторяться, если между ними есть хотя бы одна «глобальная» метка.

Локальные метки удобно использовать, например, внутри процедуры. Можно дать им простые, понятные имена и не беспокоиться, что где-то в коде уже объявлена метка с таким именем. В качестве примера добавим локальные метки в процедуру преобразования строки в число:

|  |
| --- |
| *;Процедура преобразования десятичной строки в слово без знака*  *; вход: AL - длина строки*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  *; выход: AX - слово (в случае ошибки AX = 0)*  *; CF = 1 - ошибка*  str\_to\_udec\_word:  **push** **cx** *;Сохранение всех используемых регистров*  **push** **dx**  **push** **bx**  **push** **si**  **push** **di**    **mov** **si**,**dx** *;SI = адрес строки*  **mov** **di**,10 *;DI = множитель 10 (основание системы счисления)*  **movzx** **cx**,**al** *;CX = счётчик цикла = длина строки*  **jcxz** .error *;Если длина = 0, возвращаем ошибку*  **xor** **ax**,**ax** *;AX = 0*  **xor** **bx**,**bx** *;BX = 0*    .lp:  **mov** **bl**,[**si**] *;Загрузка в BL очередного символа строки*  **inc** **si** *;Инкремент адреса*  **cmp** **bl**,'0' *;Если код символа меньше кода '0'*  **jl** .error *; возвращаем ошибку*  **cmp** **bl**,'9' *;Если код символа больше кода '9'*  **jg** .error *; возвращаем ошибку*  **sub** **bl**,'0' *;Преобразование символа-цифры в число*  **mul** **di** *;AX = AX \* 10*  **jc** .error *;Если результат больше 16 бит - ошибка*  **add** **ax**,**bx** *;Прибавляем цифру*  **jc** .error *;Если переполнение - ошибка*  **loop** .lp *;Команда цикла*  **jmp** .exit *;Успешное завершение (здесь всегда CF = 0)*    .error:  **xor** **ax**,**ax** *;AX = 0*  **stc** *;CF = 1 (Возвращаем ошибку)*    .exit:  **pop** **di** *;Восстановление регистров*  **pop** **si**  **pop** **bx**  **pop** **dx**  **pop** **cx**  **ret** |

Локальные метки намного улучшают читаемость кода. Если потребуется обратиться к локальной метке из другого места программы, это можно сделать, указав её полное имя:

|  |
| --- |
| **jmp** str\_to\_udec\_word.error *;Переход к локальной метке* |

Особым образом обрабатываются метки, имя которых начинается с двух точек. Такие метки ведут себя как глобальные, но не становятся префиксом для локальных меток.

**Анонимные метки**

Анонимная метка — это метка с именем *@@*. В программе можно объявлять сколько угодно анонимных меток, но обратиться получится только к ближайшей. Для этого существуют специальные имена: вместо *@b* (или *@r*) FASM подставляет адрес предыдущей анонимной метки, а вместо *@f* — адрес следующей анонимной метки. Этого, как правило, достаточно, чтобы реализовать простой цикл, переход или проверку условия. Таким образом можно избавиться от большого количества «неанонимных» меток. Вот пример той же процедуры с использованием анонимных меток:

|  |
| --- |
| *;Процедура преобразования десятичной строки в слово без знака*  *; вход: AL - длина строки*  *; DX - адрес строки, заканчивающейся символом CR(0Dh)*  *; выход: AX - слово (в случае ошибки AX = 0)*  *; CF = 1 - ошибка*  str\_to\_udec\_word:  **push** **cx** *;Сохранение всех используемых регистров*  **push** **dx**  **push** **bx**  **push** **si**  **push** **di**    **mov** **si**,**dx** *;SI = адрес строки*  **mov** **di**,10 *;DI = множитель 10 (основание системы счисления)*  **movzx** **cx**,**al** *;CX = счётчик цикла = длина строки*  **jcxz** .error *;Если длина = 0, возвращаем ошибку*  **xor** **ax**,**ax** *;AX = 0*  **xor** **bx**,**bx** *;BX = 0*    @@: **mov** **bl**,[**si**] *;Загрузка в BL очередного символа строки*  **inc** **si** *;Инкремент адреса*  **cmp** **bl**,'0' *;Если код символа меньше кода '0'*  **jl** .error *; возвращаем ошибку*  **cmp** **bl**,'9' *;Если код символа больше кода '9'*  **jg** .error *; возвращаем ошибку*  **sub** **bl**,'0' *;Преобразование символа-цифры в число*  **mul** **di** *;AX = AX \* 10*  **jc** .error *;Если результат больше 16 бит - ошибка*  **add** **ax**,**bx** *;Прибавляем цифру*  **jc** .error *;Если переполнение - ошибка*  **loop** @b *;Команда цикла*  **jmp** @f *;Успешное завершение (здесь всегда CF = 0)*    .error:  **xor** **ax**,**ax** *;AX = 0*  **stc** *;CF = 1 (Возвращаем ошибку)*    @@: **pop** **di** *;Восстановление регистров*  **pop** **si**  **pop** **bx**  **pop** **dx**  **pop** **cx**  **ret** |

Рекомендуется локальные и анонимные метки использовать везде, где только возможно. Они делают код программы более понятным и не захламляют пространство имён.

**Упражнение**

В приведённом коде процедуры замените «глобальные» метки на локальные. После этого замените локальные метки на анонимные, где это возможно.

|  |
| --- |
| *;Процедура преобразования байта в строку в двоичном виде*  *; AL - байт.*  *; DI - буфер для строки (8 символов). Значение регистра не сохраняется.*  byte\_to\_bin\_str:  **push** **cx** *;Сохранение CX*  **mov** **cx**,8 *;Счётчик цикла*    btbs\_lp:  **rol** **al**,1 *;Циклический сдвиг AL влево на 1 бит*  **jc** btbs\_1 *;Если выдвинутый бит = 1, то переход*  **mov** **byte**[**di**],'0' *;Добавление символа '0' в строку*  **jmp** btbs\_end  btbs\_1:  **mov** **byte**[**di**],'1' *;Добавление символа '1' в строку*  btbs\_end:  **inc** **di** *;Инкремент DI*  **loop** btbs\_lp *;Команда цикла*    **pop** **cx** *;Восстановление CX*  **ret** *;Возврат из процедуры* |

**Часть 28. Основы создания макросов**

Отличительной особенность FASM является очень гибкая и мощная поддержка макросов. В этом разделе рассмотрим основы создания макросов.

Макросы — это шаблоны для генерации кода. Один раз создав макрос, мы можем использовать его во многих местах в коде программы. Макросы делают процесс программирования на ассемблере более приятным и простым, а код программы получается понятнее. Макросы позволяют расширять синтаксис ассемблера и даже добавлять собственные «команды», которых нет в процессоре.

Обработкой макросов занимается *препроцессор* FASM. Преобразование исходного кода в исполняемый код FASM выполняет в два этапа. Первый этап — *препроцессирование*, а второй — собственно *ассемблирование* или *компиляция*. На первом этапе происходит вычисление всех числовых выражений, вместо констант и названий меток подставляются их фактические значения, вместо макросов подставляется сгенерированный код. На втором этапе все данные и машинные команды преобразуются в соответствующие байты, и в результате получается исполняемый файл требуемого формата.

**Синтаксис создания макроса**

Для создания макроса используется директива *macro*. Эта директива имеет следующий синтаксис:

|  |
| --- |
| macro <название\_макроса> [<список\_параметров>]  {  <тело\_макроса>  } |

После директивы пишется название макроса, а также может быть указан список параметров (операндов). Параметры в списке перечисляются через запятую. Внутри фигурных скобок записывается тело макроса. Кстати, для коротких макросов можно писать всё это в одной строке:

|  |
| --- |
| macro <название\_макроса> [<список\_параметров>] { <тело\_макроса> } |

Тело макроса — это код, который подставляется в то место, где макрос будет вызван. Создание макроса является по сути лишь его объявлением, в этом месте программы никакого кода сгенерировано не будет! Поэтому объявления макросов обычно размещают в самом начале программы или в отдельном файле.

**Примеры макросов**

В качестве примера рассмотрим простой макрос без параметров, который предназначен для завершения программы:

|  |
| --- |
| macro exit\_app  {  **mov** **ax**,4C00h  **int** 21h  } |

После того, как макрос объявлен, в нужном месте программы достаточно написать exit\_app. Туда препроцессор FASM автоматически подставит 2 команды, записанные в теле макроса. Создадим ещё один полезный макрос, предназначенный для такой часто используемой операции, как вывод строки:

|  |
| --- |
| macro print\_str **str**  {  **mov** **ah**,9  **mov** **dx**,**str**  **int** 21h  } |

У этого макроса есть один параметр — адрес строки. При генерации кода вместо str будет подставлен тот параметр, который указан при вызове макроса. Код будет генерироваться в месте каждого вызова макроса! В этом главное отличие макроса от процедуры. Код процедуры содержится в программе только в одном экземпляре, а вызывается она с помощью передачи управления командой CALL.

Теперь добавим эти макросы в программу «hello, world!» из части 6 учебного курса:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | *; Макрос выхода из программы*  macro exit\_app  {  **mov** **ax**,4C00h *;Здесь только объявление макроса, код не генерируется*  **int** 21h  }    *; Макрос вывода строки*  macro print\_str **str**  {  **mov** **ah**,9 *;Здесь тоже код не генерируется*  **mov** **dx**,**str**  **int** 21h  }    *;-------------------------------------------------------------------------------*  **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    print\_str hello *;Вывод строки*  exit\_app *;Выход из программы*    *;-------------------------------------------------------------------------------*  hello **db** 'Hello, macro world!$' |

В результате основная программа состоит всего из двух строчек, и это уже не похоже на ассемблер, это — макроассемблер FASM.

**Расширение системы команд**

Макросы можно использовать для расширения системы команд. Например, часто в программе приходится обнулять регистры. Создадим специальный макрос для этой цели:

|  |
| --- |
| *; Макрос - команда обнуления регистра*  macro clr reg { **xor** reg,reg } |

Теперь обнулять регистры в программе можно так:

|  |
| --- |
| clr **ax** *;AX=0*  clr **si** *;SI=0*  clr **bl** *;BL=0* |

**Макросы с переменным количеством параметров**

Возможно также создавать макросы с переменным количеством параметров. В этом случае имя параметра записывается в квадратных скобках. Для генерации кода макрос вызывается столько раз, сколько параметров ему было передано. Например, можно написать специальные макросы для улучшения команд PUSH и POP:

|  |
| --- |
| *; Макрос - улучшенная команда push*  macro **push** [arg] { **push** arg }  *; Макрос - улучшенная команда pop*  macro **pop** [arg] { **pop** arg } |

Несмотря на то, что название макроса совпадает с именем команды, в теле макроса это имя считается именем команды. Данные макросы позволяют помещать в стек и извлекать из стека сразу несколько операндов, что упрощает код программы. Пример использования:

|  |
| --- |
| **push** **ax**,**word**[**si**],5  **pop** **dx**,**cx**,**ax** |

В результате препроцессором будет сгенерирован следующий код:

|  |
| --- |
| **push** **ax**  **push** **word**[**si**]  **push** 5  **pop** **dx**  **pop** **cx**  **pop** **ax** |

**Директива include**

Возможно что Вам будет необходимо написать собственный набор макросов и использовать их в своих программах. В этом случае удобно поместить макросы в отдельный файл и воспользоваться директивой включения файла *include*. Синтаксис директивы *include* очень прост:

|  |
| --- |
| include 'путь/к/файлу' |

Путь к файлу указывается в одинарных кавычках и может быть относительным (по отношению к компилируемому файлу) или полным (начиная от буквы диска или корневого каталога системы). Если включаемый файл находится в той же папке, то достаточно указать только имя файла. Расширение файла может быть любым, но обычно используют «inc» или «asm».

Препроцессор FASM читает указанный файл и подставляет код из него вместо директивы *include*. В отдельный файл можно также вынести часто используемые процедуры, отдельные функциональные блоки программы или даже объявления данных.

Если записать макросы в отдельный файл ‘mymacro.inc’, то программа «hello, world!» станет ещё короче:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | include 'mymacro.inc'    **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*    print\_str hello *;Вывод строки*  exit\_app *;Выход из программы*    *;-------------------------------------------------------------------------------*  hello **db** 'Hello, macro world!$' |

**Упражнение**

Напишите макрос для определения максимального значения. У макроса должно быть 3 операнда: второй и третий сравниваются между собой, больший из них помещается на место первого.

**Часть 29. Макросы PROC и ENDP**

Прктика показывает, часто довольно неудобно обращаться к параметрам и локальным переменным, указывая смещения относительно регистра BP. Такой способ подходит только для совсем простых и маленьких процедур. Поэтому в таких ассемблерах, как TASM и MASM, существуют специальные директивы, позволяющие создавать процедуры быстро и удобно. В FASM таких директив нет! Но они и не нужны — то же самое можно сделать с помощью макросов.

Для начала, нам потребуется заголовочный файл с макросами. Стандартный пакет FASM для Windows, к сожалению, не включает в себя макросы для 16-битных процедур. Однако такие макросы можно найти на официальном форуме FASM или открыть из файла: PROC16.INC. Это переделанная версия файла PROC32.INC с точно таким же синтаксисом.

Заголовочный файл необходимо будет включить в программу с помощью директивы *include*:

|  |  |
| --- | --- |
| 1 | include 'proc16.inc' |

**Базовый синтаксис объявления процедуры**

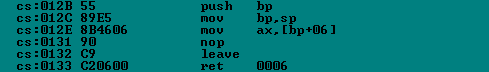
Для создания процедуры используется следующий синтаксис:

|  |
| --- |
| proc <имя\_процедуры>[,][<список\_параметров>]  ...  **ret**  endp |

После *proc* указывается имя процедуры. Далее через запятую список параметров. Между именем процедуры и списком параметров запятую ставить не обязательно (можно просто поставить пробел). Для возврата из процедуры следует использовать команду RET без операндов. Завершается процедура макросом *endp*. Например, объявим процедуру с тремя параметрами:

|  |
| --- |
| proc myproc,a,b,c  **mov** **ax**,[b]  ...  **ret**  endp |

Внутри процедуры обращаться к параметрам можно как к простым переменным — с помощью квадратных скобок! При вызове процедуры параметры должны помещаться в стек, начиная с последнего. Если запустить программу в отладчике, то можно увидеть сгенерированный макросами код (вместо 3-х точек поставим команду NOP):



Макросы создали нужный пролог и эпилог процедуры. Команда RET дополнительно выталкивает 6 байт из стека (это как раз 3 параметра-слова).

**Указание размеров параметров**

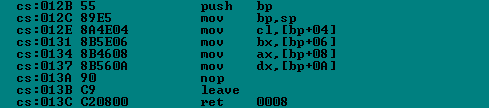
По умолчанию размер параметров считается равным ширине стека, то есть в нашем случае 16-бит. Если требуется передавать процедуре байт или двойное слово, то нужно дополнительно указать размер. Это можно сделать, поставив двоеточие после имени параметра. Возможные операторы размера перечислены в таблице:

|  |  |
| --- | --- |
| **Оператор** | **Размер в байтах** |
| BYTE | 1 |
| WORD | 2 |
| DWORD | 4 |
| PWORD | 6 |
| QWORD | 8 |
| TBYTE | 10 |
| DQWORD | 16 |

Пусть первый параметр процедуры будет иметь размер байт, второй — слово, третий — двойное слово:

|  |
| --- |
| proc myproc,a:**BYTE**,b:**WORD**,c:**DWORD**  **mov** **cl**,[a]  **mov** **bx**,[b]  **mov** **ax**,**word**[c]  **mov** **dx**,**word**[c+2]  ...  **ret**  endp |

В результате будет сгенерирован такой код:



Первый параметр будет занимать в стеке одну ячейку, хотя использоваться будет только младший байт. Третий параметр займёт 2 смежные ячейки стека. Для избежания путаницы, рекомендуется всегда использовать только параметры размером слово (или двойное слово в 32-битном ассемблере). К сожалению поместить отдельный байт в стек всё равно не получится, а *DWORD* придётся «заталкивать» двумя командами, поэтому лучше разделить его на два слова, явно указав младшую и старшую часть.

**Указание соглашений вызова**

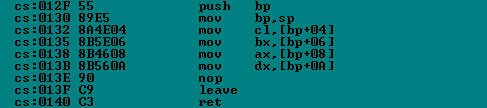
Дополнительно для процедуры можно указать соглашения вызова. Чтобы это сделать, надо добавить специальное ключевое слово после имени процедуры. Доступно всего 2 варианта:

|  |  |  |
| --- | --- | --- |
| **Ключевое слово/ соглашения вызова** | **Порядок помещения параметров в стек** | **Очищает стек** |
| stdcall | обратный | вызываемая процедура |
| c | обратный | вызывающий код |

Буква *c* здесь обозначает соглашения вызова для языка Си. По умолчанию используется *stdcall*. Например, пусть наша процедура не очищает стек:

|  |
| --- |
| proc myproc c,a:**BYTE**,b:**WORD**,c:**DWORD**  **mov** **cl**,[a]  **mov** **bx**,[b]  **mov** **ax**,**word**[c]  **mov** **dx**,**word**[c+2]  ...  **ret**  endp |

В результате получится следующий код (сравните с предыдущим примером и найдите одно отличие!):



**Сохранение и восстановление используемых регистров**

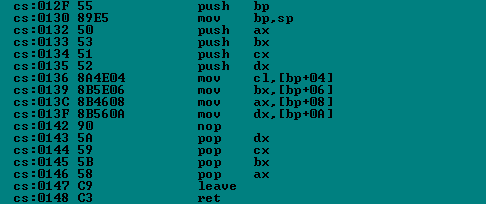
Макросы PROC и ENDP позволяют также организовать сохранение и восстановление регистров, используемых кодом процедуры. Для этого после имени процедуры нужно указать ключевое слово *uses* и список регистров через пробел. Регистры будут помещены в стек при входе в процедуру (в порядке их записи) и восстановлены перед возвратом. Например, добавим сохранение регистров к процедуре:

|  |
| --- |
| proc myproc c uses **ax** **bx** **cx** **dx**,a:**BYTE**,b:**WORD**,c:**DWORD**  **mov** **cl**,[a]  **mov** **bx**,[b]  **mov** **ax**,**word**[c]  **mov** **dx**,**word**[c+2]  ...  **ret**  endp |

Если объявление процедуры получается слишком длинным, можно продолжить его на следующей строке, добавив символ *\* в конец первой строки (это работает и с любыми другими макросами):

|  |
| --- |
| proc myproc4 c uses **ax** **bx** **cx** **dx**,\  a:**BYTE**,b:**WORD**,c:**DWORD** |

Результат:



**Объявление локальных переменных**

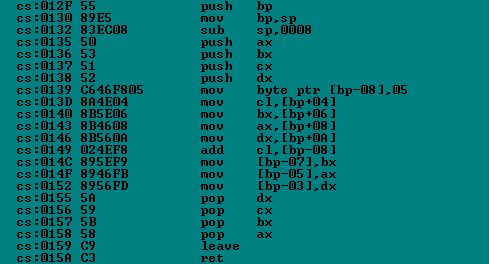
Для объявления локальных переменных существует 3 варианта синтаксиса:

**1. *local* + директивы объявления данных**

Макрос *local* предназначен для создания локальных переменных. После слова *local* локальные переменные объявляются обычными директивами. Можно использовать как инициализированные, так и неинициализированные переменные. Можно объявлять несколько переменных в одной строке через запятую (однако, не получится объявить массив, перечислив значения). Например:

|  |
| --- |
| proc myproc c uses **ax** **bx** **cx** **dx**,\  a:**BYTE**,b:**WORD**,c:**DWORD**  local i **db** 5  local j **dw** ?, k rd 1  **mov** **cl**,[a]  **mov** **bx**,[b]  **mov** **ax**,**word**[c]  **mov** **dx**,**word**[c+2]  **add** **cl**,[i]  **mov** [j],**bx**  **mov** **word**[k],**ax**  **mov** **word**[k+2],**dx**  **ret**  endp |

После объявления обращаться к локальным переменным можно также, как к параметрам и глобальным переменным. Получится код:



В стеке для локальных переменных выделяется 8 байт памяти. Первая переменная находится по самому младшему адресу (ближе к вершине стека). Сгенерировалась также команда, присваивающая значение первой переменной (сразу после сохранения регистров).

**2. Альтернативный синтаксис *local***

Альтернативный синтаксис похож на синтаксис размеров параметров. После имени переменной ставится двоеточие и оператор размера. Вместо оператора размера может быть также имя структуры (это удобно при программировании для Windows).

|  |
| --- |
| local i:**BYTE**  local j:**WORD**, k:**DWORD** |

Инициализация переменных в этом варианте синтаксиса не предусмотрена, её придётся делать вручную. Зато можно легко объявлять массивы — обозначение очень напоминает языки высокого уровня:

|  |
| --- |
| local buffer[256]:**BYTE** *;Локальный буфер размером 256 байт* |

**3. *locals* и *endl***

Третий вариант — объявление локальных переменных в виде блока *locals* — *endl*. Используются обычные директивы объявления данных:

|  |
| --- |
| proc myproc c uses **ax** **bx** **cx** **dx**,\  a:**BYTE**,b:**WORD**,c:**DWORD**  locals  i **db** 5  j **dw** ?  k rd 1  endl  **mov** **cl**,[a]  **mov** **bx**,[b]  **mov** **ax**,**word**[c]  **mov** **dx**,**word**[c+2]  **add** **cl**,[i]  **mov** [j],**bx**  **mov** **word**[k],**ax**  **mov** **word**[k+2],**dx**  **ret**  endp |

Этот способ хорошо подходит, если в процедуре много локальных переменных.

**Макросы для вызова процедур**

В файле PROC16.INC объявлены также макросы для удобного вызова процедур. Эти макросы избавляют от необходимости писать несколько команд PUSH для помещения параметров в стек. Вместо этого достаточно написать одну строку, в которой указывается имя процедуры и список параметров через запятую. Например:

|  |
| --- |
| stdcall <имя\_процедуры>[,<список\_параметров>] |

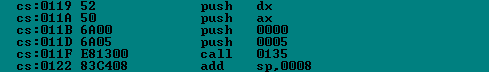
Всего существует 4 разных макроса, они перечислены в таблице. Два последних макроса чаще используются в программировании для Windows. Они выполняют вызов процедуры, адрес которой находится в переменной.

|  |  |
| --- | --- |
| **Макрос** | **Описание** |
| stdcall | Вызов процедуры с соглашениями вызова *stdcall* |
| ccall | Вызов процедуры с соглашениями вызова *c* |
| invoke | То же самое, что stdcall [<имя\_переменной>] |
| cinvoke | То же самое, что ccall [<имя\_переменной>] |

Так как наша процедура использует соглашения вызова *c*, то вызывать её надо макросом *ccall*:

|  |
| --- |
| ccall myproc,5,0,**ax**,**dx** *;3-й параметр находится в DX:AX* |

Макрос дополнительно генерирует код для восстановления указателя стека:

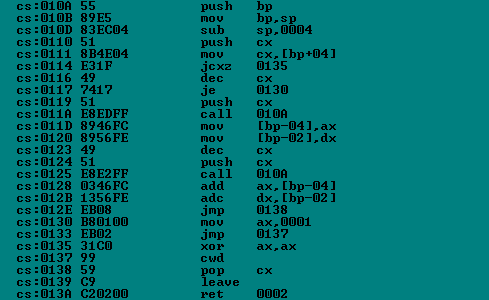


**Важная особенность**

Если была объявлена процедура, но ее ни разу не вызывали, то её код не будет добавлен в исполняемый файл! Это позволяет создать свою библиотеку полезных процедур, сохранить их в отдельном файле и добавлять во все проекты, где они нужны. При этом в исполняемом файле окажутся только те процедуры, которые использует программа.

**Упражнение** (на дизассемблирование)

Проанализируйте процедуру и напишите для неё ассемблерный код с использованием макросов PROC16. Что делает эта процедура? Подумайте, какие недостатки у неё есть и как их можно исправить.



**Часть 30. Команды работы с битами**

Работать с отдельными битами операндов можно, используя логические операции и сдвиги. Однако, кроме них в системе команд x86 существуют специальные команды для работы с битами: это команды сканирования битов и команды проверки (и модификации) битов. Впервые они появились в процессоре i386. Так что сейчас вряд ли найдётся процессор, в котором их нет.

**Команды сканирования битов**

Сканирование битов выполняется командами BSF и BSR. Эти команды очень похожи. У них 2 операнда. Первый операнд должен быть 16-битным регистром, в него записывается результат. Второй операнд может быть 16-битным регистром или словом в памяти — это обрабатываемое значение.

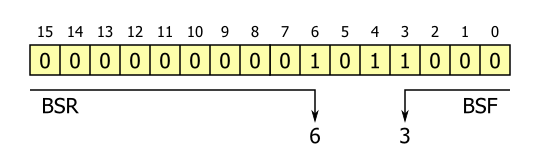
Команда BSF просматривает биты второго операнда от младшего к старшему и помещает индекс первого единичного бита в регистр. Биты нумеруются, начиная с нуля. Если единичный бит найден, то флаг нуля сбрасывается (ZF=0). Если все биты нулевые, то флаг нуля устанавливается (ZF=1), а значение первого операнда будет неопределённым (на разных процессорах может быть по-разному). Например, мой процессор (Athlon XP) в этом случае не изменяет значение в регистре. Пример кода:

|  |
| --- |
| **mov** **ax**,01011000b *;AX=58h*  **bsf** **bx**,**ax** *;BX=3, ZF=0*  **xor** **ax**,**ax** *;AX=0*  **bsf** **bx**,**ax** *;BX=?, ZF=1* |

Команда BSR отличается тем, что просматривает биты от старшего к младшему. Всё остальное также.

|  |
| --- |
| **mov** **ax**,01011000b *;AX=58h*  **bsr** **bx**,**ax** *;BX=6, ZF=0*  **xor** **ax**,**ax** *;AX=0*  **bsr** **bx**,**ax** *;BX=?, ZF=1* |

Картинка для наглядности:



**Команды проверки и модификации битов**

Команда BT копирует значение проверяемого бита в флаг CF. Вот и вся проверка! После этого можно выполнить условный переход командами JC или JNC, в зависимости от значения бита. У команды два операнда: слово в регистре или в памяти и номер бита, который может находиться в регистре или быть непосредственным значением. Примеры использования команды:

|  |
| --- |
| **bt** **ax**,0 *;Проверка младшего бита AX*  **jc** m1 *;Переход, если бит равен 1*  **mov** **cx**,3 *;CX=3*  **bt** **ax**,**cx** *;Проверка 3-го бита AX*  **jnc** m1 *;Переход, если бит равен 0* |

Ещё 3 команды немного отличаются от BT:

* команда BTR проверяет бит и затем сбрасывает его;
* команда BTS проверяет бит и затем устанавливает его в 1;
* команда BTC проверяет бит и затем инвертирует его.

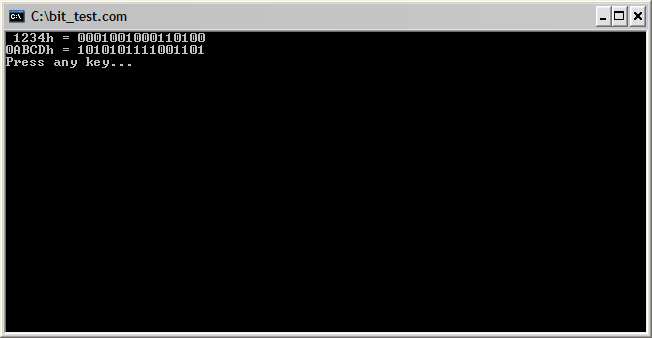
Эти команды удобны тем, что можно совместить проверку бита и присвоение ему нового значения.

**Пример программы**

Команду BT можно использовать в цикле для вывода чисел в двоичном виде. Вместо команды условного перехода рекомендуется использовать команду ADC. Она прибавляет значение бита к коду символа. Получается ‘0’ или ‘1’. В коде используются макросы из предыдущей части.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | include 'proc16.inc'  **use16** *;Генерировать 16-битный код*  org 100h *;Программа начинается с адреса 100h*  **jmp** **start**  *;-------------------------------------------------------------------------------*  *; Данные*  s\_1 **db** ' 1234h = $'  s\_2 **db** '0ABCDh = $'  s\_pak **db** 'Press any key...$'  *;-------------------------------------------------------------------------------*  **start**:  stdcall print,s\_1 *;Вывод строки s\_1*  stdcall print\_bin,1234h *;Вывод числа 1234h*  stdcall endline *;Переход на новую строку*  stdcall print,s\_2 *;Вывод строки s\_2*  stdcall print\_bin,0ABCDh *;Вывод числа 0ABCDh*  stdcall endline *;Переход на новую строку*  stdcall print,s\_pak *;Вывод строки 'Press any key...'*  **mov** **ah**,8 *;\*  **int** 21h *;/ Ввод символа без эха*  **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*    *;-------------------------------------------------------------------------------*  *; Процедура вывода числа в двоичном виде*  proc print\_bin uses **ax** **cx** **bx** **dx**, value  **mov** **bx**,[value] *;Загрузка числа в BX*  **mov** **cx**,15 *;Счетчик битов*  **mov** **ah**,2 *;Для функции DOS 02h*  @@: **mov** **dl**,'0' *;DL='0'*  **bt** **bx**,**cx** *;Проверка бита!*  **adc** **dl**,**ch** *;Прибавление значения бита (CH=0)*  **int** 21h *;Вывод символа*  **dec** **cx** *;Декремент счетчика*  **jns** @b *;Переход, если неотрицательное значение*  **ret** *;Возврат из процедуры*  endp    *;-------------------------------------------------------------------------------*  *; Процедура вывода строки*  proc print uses **ax** **dx**, **str**  **mov** **ah**,9 *;Функция DOS 09h*  **mov** **dx**,[**str**] *;Загрузка адреса строки в DX*  **int** 21h *;Вывод строки*  **ret** *;Возврат из процедуры*  endp    *;-------------------------------------------------------------------------------*  *; Процедура вывода перехода на новую строку*  proc endline  **call** @f *;Подумайте, как это работает*  **db** 13,10,'$'  @@: **call** print  **ret**  endp |

Результат работы программы:



**Упражнение**

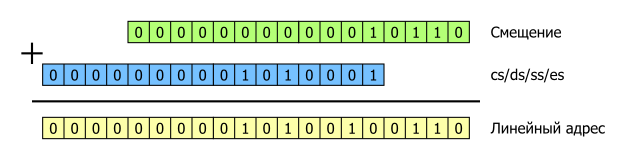
Напишите процедуру, которая сбрасывает старший единичный бит и устанавливает в единицу младший нулевой бит в регистре AX.

**Часть 31. Сегментная адресация**

До этой части учебного курса мы создавали только COM-программы, в которых один и тот же сегмент использовался для кода, данных и стека. Однако, для дальнейшего изложения необходимо подробнее разобраться с сегментной адресацией.

**Формирование адреса в реальном режиме**

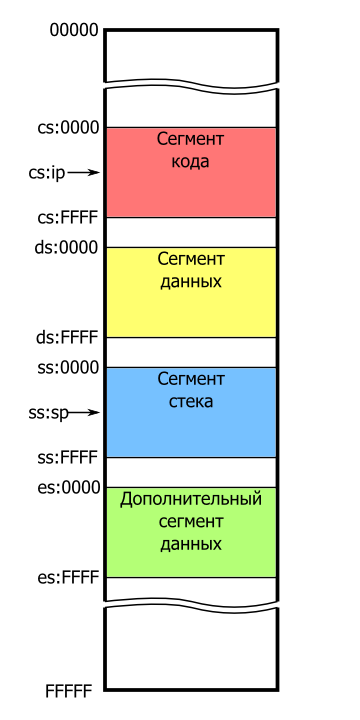
Основная идея сегментной адресации в том, что адрес состоит из двух частей — сегментной и смещения. Обычно их записывают через двоеточие (например 0100:0500). Линейный адрес любой ячейки памяти получается в результате сложения смещения и сегментной части, сдвинутой на 4 бита влево.



Начало сегмента всегда выровнено на границу параграфа (адрес кратен 16 байтам). Максимальный размер сегмента равен 216 = 64 КБайта. А всего можно адресовать 220 = 1 МБайт памяти.

Одна из особенностей сегментной адресации — неоднозначность представления адреса. Допустим, требуется обратиться к ячейке памяти по адресу 00400. Этот адрес может быть представлен как 0000:0400, 0040:0000, 0020:0200 и так далее.

Загруженная в память программа может одновременно работать с четырьмя сегментами. Сегменты могут перекрываться или даже совпадать, как это было в случае с COM-программой.



Для всех команд подразумевается сегмент «по умолчанию». Например, команды PUSH и POP работают с сегментом стека. Если операнд такой команды находится в памяти, то он берётся из сегмента данных. Команды JMP и LOOP вычисляют адрес перехода в сегменте кода.

**Создание DOS EXE**

Возможности сегментной адресации полностью реализуются в исполняемом файле DOS EXE. Не путайте этот формат с исполняемым файлом Windows (PE EXE)! Расширение такое же, но файл имеет совершенно другую структуру.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | format MZ *;Исполняемый файл DOS EXE (MZ EXE)*  entry code\_seg:**start** *;Точка входа*  **stack** 200h *;Размер стека*    *;----------------------------------------------------------------------*  **segment** data\_seg *;Cегмент данных*  hello **db** 'Hello, asmworld!$' *;Строка*    *;----------------------------------------------------------------------*  **segment** code\_seg *;Сегмент кода*  **start**: *;Отсюда начинается выполнение программы*  **mov** **ax**,data\_seg *;\*  **mov** **ds**,**ax** *;/ Инициализация регистра DS*    **mov** **ah**,09h *;\*  **mov** **dx**,hello *; > Вывод строки*  **int** 21h *;/*    **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы* |

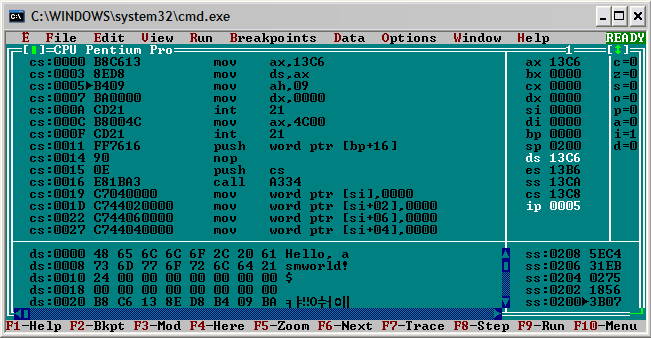
В первой строке после директивы *format* нужно поставить *MZ*, чтобы FASM сгенерировал нужный файл.

Во второй строке указывается точка входа, то есть метка, с которой начинается выполнение программы. Имя метки дополняется названием сегмента, в котором она находится.

После директивы *stack* можно указать требуемый размер сегмента стека в байтах (по умолчанию используется 4096). Дальше файл состоит из сегментов, которые объявляются с помощью директивы *segment*. После директивы записывается название сегмента.

В примере файл состоит из двух сегментов. В первом находятся данные (а точнее строка), во втором — код. Выполнение программы начинается с метки *start* в сегменте кода. После запуска необходимо инициализировать регистр *ds*, чтобы выбрать нужный сегмент данных. Для этого используются 2 команды, так как невозможно напрямую записать значение в сегментный регистр (нет команды MOV *ds*,*значение*).

Работу программы с сегментами можно увидеть в отладчике. Обратите внимание, что *cs*, *ds*, *es* и *ss* имеют разные значения:



**Префиксы переопределения сегментов**

Иногда нужно изменить используемый сегмент данных только для одной команды. Например, необходимо прочитать байт из текущего сегмента кода или записать в сегмент стека. Для этого предназначены префиксы переопределения сегмента. Названия префиксов совпадают с названиями сегментных регистров.

Особенность синтаксиса FASM в том, что префикс пишется внутри квадратных скобок (так как по смыслу является частью адреса):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | format MZ *;Исполняемый файл DOS EXE (MZ EXE)*  entry code\_seg:**start** *;Точка входа*  **stack** 200h *;Размер стека*    *;----------------------------------------------------------------------*  **segment** data\_seg *;Cегмент данных*  hello **db** 'Hello, asmworld!$' *;Строка*    *;----------------------------------------------------------------------*  **segment** code\_seg *;Сегмент кода*  **start**: *;Отсюда начинается выполнение программы*  **mov** **ax**,data\_seg *;\*  **mov** **ds**,**ax** *;/ Инициализация регистра DS*    **mov** **ah**,09h *;\*  **mov** **dx**,hello *; > Вывод строки*  **int** 21h *;/*    **mov** **ax**,eseg *;\*  **mov** **es**,**ax** *;/ Инициализация регистра ES*  **mov** **al**,[**cs**:**start**] *;Чтение байта, с которого начинается код*  **cmp** **al**,0xB8  **jnz** exit  **mov** **word**[**es**:0000h],1234h *;Запись значения в сегмент eseg*    exit:  **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*    *;----------------------------------------------------------------------*  **segment** eseg  rw 1 *;Зарезервировать 1 слово* |

**Дальние переходы, вызовы процедур и возвраты**

Дальними (*far*) называются переходы в другой сегмент кода. При их выполнении меняется содержимое регистра cs. Они могут только безусловными. Ближние (*near*) переходы осуществляются в пределах одного сегмента. Аналогично есть дальние и ближние вызовы процедур, а также дальние и ближние возвраты.

Команда дальнего вызова процедуры сохраняет в стек не только *ip*, но и *cs*, чтобы можно было вернуться в текущий сегмент кода. Команда RET является синонимом ближнего возврата RETN. Дальний возврат осуществляется командой RETF. Она восстанавливает из стека регистры *ip* и *cs*.

Для наглядности пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | format MZ *;Исполняемый файл DOS EXE (MZ EXE)*  entry seg1:**start** *;Точка входа*    *;-------------------------------------------------------------------------*  **segment** seg1 *;Сегмент первый*  hello **db** 'Hello, asmworld!$' *;Строка*    **start**: *;Отсюда начинается выполнение программы*  **push** **cs** *;\*  **pop** **ds** *;/ Инициализация регистра DS*  **jmp** seg2:do\_it    exit:  **mov** **ah**,08h  **int** 21h  **mov** **ax**,4C00h *;\*  **int** 21h *;/ Завершение программы*    *;-------------------------------------------------------------------------*  **segment** seg2 *;Сегмент второй*  do\_it:  **mov** **dx**,hello *;DX = СМЕЩЕНИЕ строки в seg1*  **call** seg3:print\_str *;Дальний вызов (cs,ip в стек)*  **jmp** seg1:exit *;Дальний переход*    *;-------------------------------------------------------------------------*  **segment** seg3  *; Дальняя процедура для вывода строки (ds:dx = адрес строки)*  print\_str:  **mov** **ah**,09h  **int** 21h  retf *;Дальний возврат (восстанавливает ip,cs)* |

Программа состоит из трёх сегментов. Сначала выполняется переход во второй, затем вызов процедуры в третьем сегменте. Кстати, сегмент может содержать код и данные вместе —в примере строка размещена в начало первого сегмента.

**Упражнение**

Напишите программу, которая сравнивает две переменные и выполняет переход в другой сегмент в зависимости от результата сравнения. Если меньше, переход в сегмент 1. Если больше — в сегмент 2. Иначе в сегмент 3.