МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ
И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

СЕВЕРО-КАВКАЗСКИЙ ФИЛИАЛ ОРДЕНА ТРУДОВОГО КРАСНОГО ЗНАМЕНИ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО ОБРАЗОВАТЕЛЬНОГО
УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
СВЯЗИ И ИНФОРМАТИКИ»

Д.А. ЖУКОВСКИЙ
А.Г. ЖУКОВСКИЙ
С.А. ШВИДЧЕНКО

МАШИННО-ЗАВИСИМЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ (ЯЗЫКИ АССЕМБЛЕРА)

Учебно-методическое пособие

Ростов-на-Дону
2022

УДК 004
ББК 32.97
Ж 86

Жуковский Д.А., Жуковский А.Г., Швидченко С.А. МАШИННО-ЗАВИСИМЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ (ЯЗЫКИ АССЕМБЛЕРА). *Учебно-методическое пособие*. – Ростов-на-Дону: СКФ МТУСИ, 2022. – 68 с.

Учебно-методическое пособие предназначено для студентов, изучающих дисциплины «Языки ассемблера» и «Машинно-зависимые языки программирования».

В пособии изложены основные понятия машинно-зависимых языков, сведения об архитектуре микропроцессоров и программ, позволяющих писать и производить отладку программ на языках ассемблера.

Пособие включает в себя ряд разделов, содержащих краткие сведения о программировании, примеры программ определенной тематики и задания для самостоятельного выполнения.

Изложены требования к содержанию отчетов по лабораторным исследованиям, приведен перечень отчетных документов.

Лабораторные исследования позволят студентам, обучающимся по направлениям подготовки бакалавров: 10.03.01 «Информационная безопасность» и 09.03.01 «Информатика и вычислительная техника» более глубоко изучить вопросы, связанные с использованием машинно-зависимых языков программирования и закрепить знания, полученные при прослушивании лекционного курса.

Пособие также будет интересно широкому кругу студентов, изучающих программирование и специалистам, работающим в области информационных технологий.

Рецензент:
Ведущий научный сотрудник «Ростовский-на-Дону НИИ радиосвязи», д.т.н., доцент А.В. Елисеев;

СОДЕРЖАНИЕ

1 **Ассемблер. Основы программирования на Ассемблере**

Ассемблер – это практически самый первый язык программирования. До него было лишь программирование в машинных кодах.

Преимущества знания ассемблера:

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

Глубокое понимание работы компьютера и операционной системы.

Если необходимо написать программу на языке высокого уровня, знание ассемблера поможет понять, как будет выполняться программа, как хранятся переменные, как вызываются функции. А это позволит избежать многих ошибок. Человек, владеющий ассемблером, будет лучше программировать и на других языках.

Максимальная гибкость при работе с аппаратными ресурсами.

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

Оптимизация программ по скорости выполнения.

Современные компиляторы хорошо оптимизируют код, поэтому писать на ассемблере все программы не имеет смысла. Однако, если необходимо написать программу для шифрования или архивации больших файлов, то применение ассемблера позволит в несколько раз увеличить скорость ее выполнения. 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 не важно. 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 «вскрыть» любую программу дизассемблером и изучить механизм её работы. Ассемблер может помочь при отладке при возникновении ошибок в компиляторах.

Язык Ассемблера - это язык программирования низкого уровня. Для начала вы должны ознакомиться с общей структурой процессора, чтобы в дальнейшем понимать, о чем идет речь.

1) Простейшая архитектура компьютера

На рисунке 1 представлена структурная схема микропроцессора 8086, в состав которого входят: устройство управления (УУ), арифметико-логическое устройство (АЛУ), блок преобразования адресов и регистры.

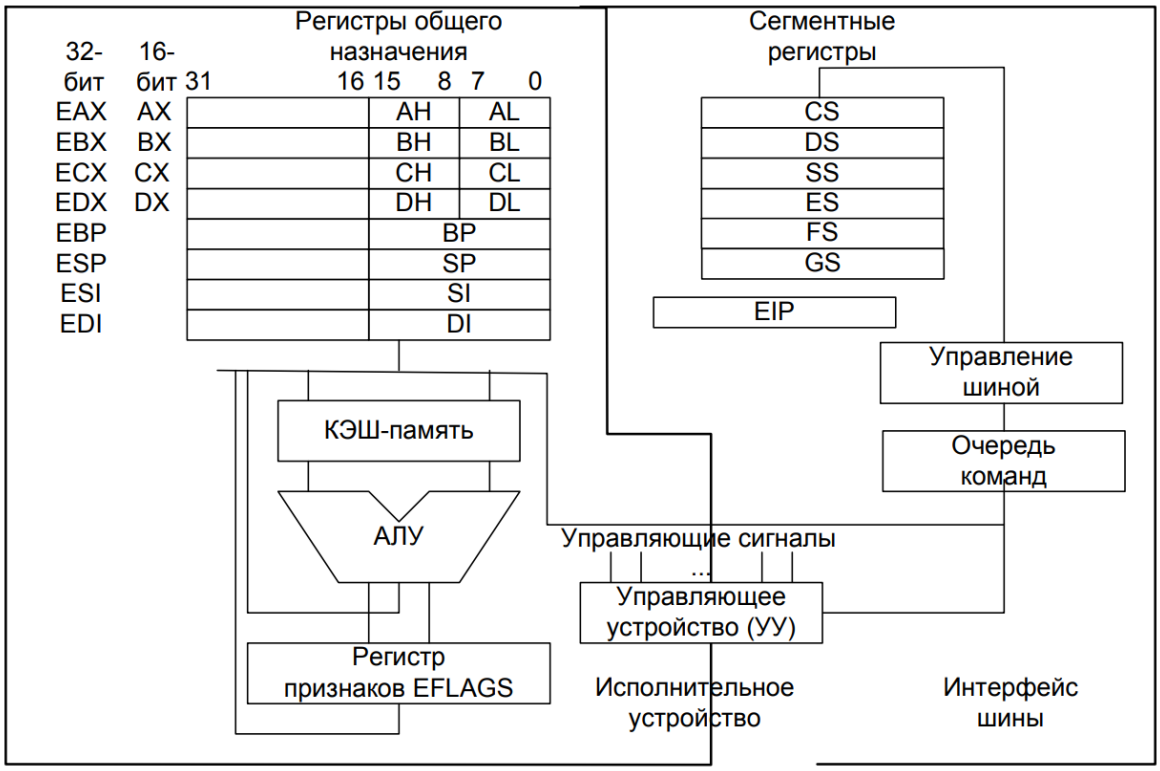


Рисунок 1

Арифметико-логическое устройство (АЛУ) производит операции над двумя величинами для получения результата и выработки ряда признаков (результат меньше нуля, равен нулю, или больше нуля и т. д.).

Аккумулятор – регистр, используемый для размещения подлежащих обработке данных или результатов выполнения операции.

Регистр команд – регистр, служащий для размещения текущей команды.

Регистр адреса – регистр, содержащий адрес ячейки памяти, из которой будет считана команда (операнд) или в которую требуется записать результат выполнения команды.

Регистр(ы) данных – используются в качестве буфера (промежуточного хранилища информации) между памятью и остальными регистрами процессора, через регистр данных пересылаются в процессор команды (операнды) и передаются в память результаты обработки.

Счетчик команд – регистр, указывающий на адрес следующей команды, которая должна быть выполнена после завершения выполнения текущей команды; содержимое счетчика команд увеличивается на единицу в момент выборки из памяти текущей исполняемой команды.

Регистр состояния (флаговый регистр) – регистр, хранящий признаки результата выполнения последней операции; эти признаки используются для организации работы команд перехода.

Устройство управления дешифрирует коды команд и формирует необходимые управляющие сигналы. Арифметико-логическое устройство осуществляет необходимые арифметические и логические преобразования данных. В блоке преобразования адресов формируются физические адреса данных, расположенных в основной памяти. Наконец, регистры используются для хранения управляющей информации: адресов и данных.

Необходимо знать, какие регистры процессора существуют и как их можно использовать. Все процессоры архитектуры x86 (даже многоядерные, большие и сложные) являются дальними потомками Intel 8086 и совместимы с его архитектурой. Это значит, что программы на ассемблере 8086 будут работать и на старших версиях процессоров.

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. (Про стек я подробно расскажу в отдельной части учебного курса)

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 получается путём сдвига содержимого сегментного регистра на 4 бита влево и прибавления к нему смещения (относительного адреса внутри сегмента).

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.

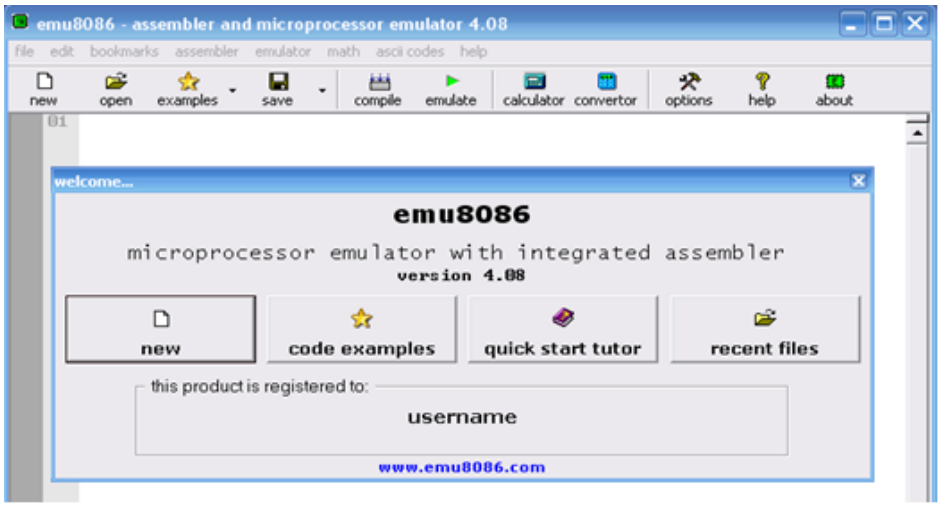
Признаки результата устанавливаются после выполнения арифметических и логических команд:

- 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.

2) Программный эмулятор (виртуальный ПК) Emu8086

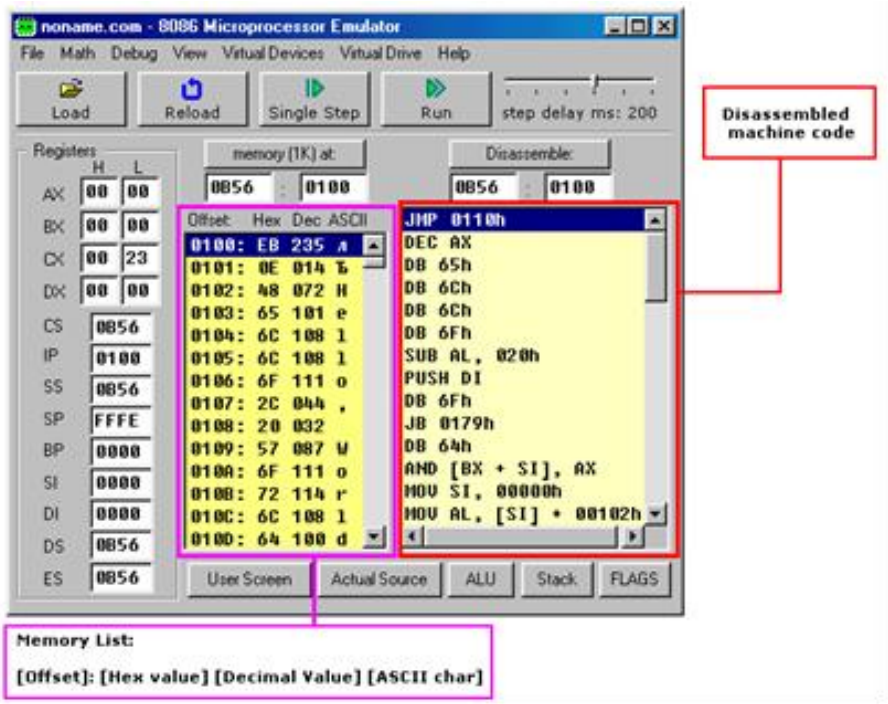
Emu8086 сочетает в себе мощный редактор исходного кода, ассемблер, дизассемблер, программный эмулятор (виртуальный ПК) с отладчиком. Эмулятор выполняет программы на виртуальном ПК, который полностью исключает возможность доступа из вашей программы к реальным аппаратным средствам, таким как жесткие диски и память.

Машинный код 8086 полностью совместим со всеми последовавшими за ним поколениями микропроцессоров Intel, включая Pentium III и Pentium. Это делает код 8086 очень привлекательным, так как он выполняется как на старых, так и на современных компьютерных системах.



3) Использование эмулятора Emu8086

Напечатайте ваш код программы внутри текстовой области. Чтобы загрузить код в эмулятор, щелкните кнопку "Emulate". 



В окне памяти (memory) перечисляют первую строку - смещение, вторая строка - значение hexadecimal, третья строка - десятичное значение, и последняя строка - значение символа ASCII.

Кнопка [Single Step] выполняет команды, один за другим останавливающие после каждой команды.

[Run] кнопка выполняет команды один за другим с задержкой, установленной задержкой шага между командами.

Дважды щелкните на текстовых полях регистра, открывается окно "Extended Viewer " со значением того регистра, преобразованного ко всем возможным формам. Вы можете изменять значение регистра непосредственно в этом окне.

Компиляция кода Ассемблера

Напечатайте ваш код внутри текстовой области и щелкните кнопку [Compile]. Вас спросят, где сохранить откомпилированный файл. После завершения компиляции вы можете щелкнуть кнопку [Emulate] для загрузки откомпилированного файла в эмулятор.

Правила ассемблера

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 эмулятор поддерживает 3 формы записи:

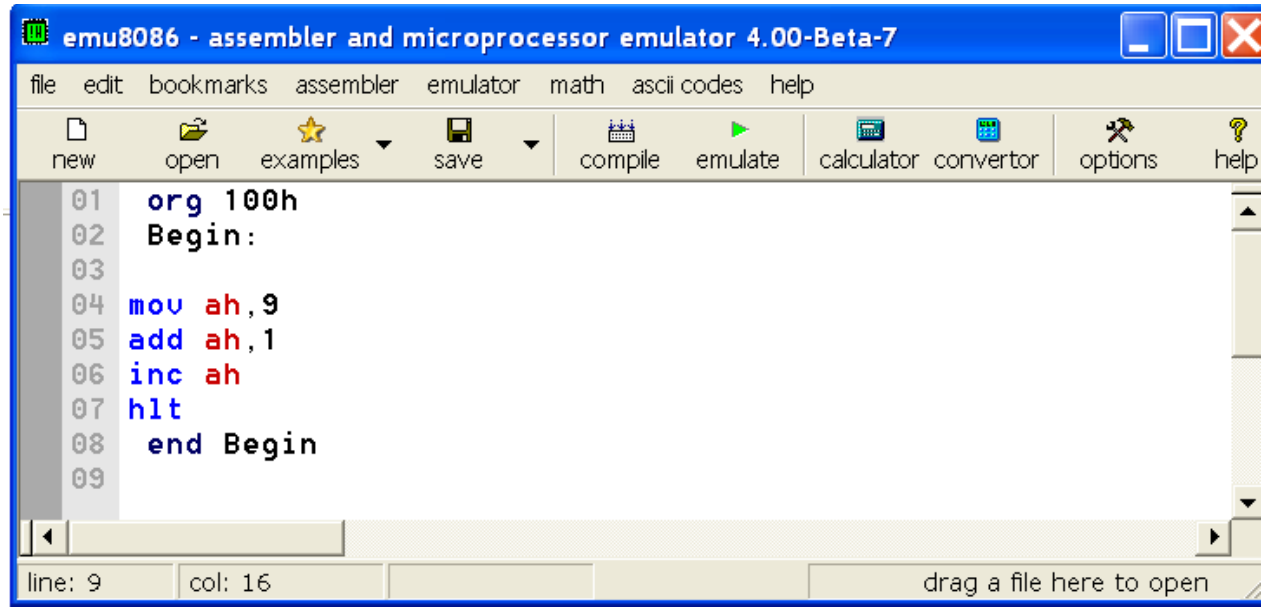
- перед числом записываются символы *'0x'* (как в C/C++);
- перед числом записывается символ *'$'* (как в Pascal);
- после числа записывается символ *'h'*. Если шестнадцатеричное число начинается с буквы, необходимо добавить в начале ноль (иначе непонятно, число это или имя метки).

4) Структура простой программы на Ассемблере

Со строки (02) начинается код программы. Это метка, указывающая Ассемблеру на начало кода. В строке (08) стоят операторы end Begin (Begin англ. начало; end конец). Это конец программы. Вместо слова Begin можно использовать другое. Например, Start:. В этом случае, необходимо было завершать программу командой End Start.

Строка (01) сообщает, что код программы необходимо в памяти компьютера отсчитывать с адреса 100h. По этому адресу в память всегда загружаются программы типа *.com., указывая Ассемблеру при ассемблировании использовать смещение 100h от начала сегмента памяти, в который загружена написанная программа. Сегментные регистры автоматически принимают значение того сегмента, в который загрузилась программа.

Строки (04-07) являются непосредственно программой и содержат команды ассемблера.



5) Операторы языка

Команда MOV

- Копирует **второй операнд** (источник) в **первый операнд** (приемник).
- Операнд-источник может быть непосредственным значением, регистром общего назначения или местоположением памяти.
- Регистр-приемник может быть регистром общего назначения или местоположением памяти.
- Оба операнда должны иметь одинаковый размер байта или слова.

Эти типы операндов поддерживаются:
MOV регистр, память
MOV память, регистр
MOV регистр, регистр
MOV память, непосредственное значение (число)
MOV регистр, непосредственное значение

регистры: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

память: [BX], [BX+SI+7], переменная, и т.п...

непосредственное значение: 5, -24, 3Fh, 10001101b, и т.п...

Сложение беззнаковых величин

Под беззнаковым понимается представление заведомо неотрицательных целых чисел, в котором знаковый бит вводить не требуется.

оператор add
Команда ADD производит сложение двух чисел.

Оператор ADD имеет следующий формат: ADD приемник, источник

Примеры использования:
mov al,10 ---> загружаем в регистр AL число 10
add al,15 ---> al = 25; al - приемник, 15 - источник

mov ax,25000 ---> загружаем в регистр AX число 25000
add ax,10000 ---> ax = 35000; ax - приемник, 10000 - источник

mov cx,200 ---> загружаем в регистр CX число 200
mov bx,760 ---> а в регистр BX --- 760
add cx,bx ---> cx = 960, bx = 760 (bx не меняется); cx - приемник, bx - источник

Задание1: Выполните сложение следующих шестнадцатеричных чисел

1C6+223=?	192+258=?	29E+14=?	28F+15D=?
1DF+20E=?	2AA+144=?	1BB+234=?	1CC+224=?
1FF+1F2=?	1EE+204=?	1AB+248=?	1BA+23A=?
1AC+249=?	1CA+22C=?	1AD+24A=?	1DA+21E=?

Директивы объявления данных

Практически любая программа кроме машинных команд содержит также какие-то 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.

Для объявления данных 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	

Синтаксис объявления данных

Чтобы объявить байт со значением 5 достаточно написать:
x db 5
где x — название нашей переменной или константы,
db — директива объявления байта,

5 — значение.

С помощью названия в программе можно будет обращаться к ячейке памяти, содержащей этот байт. Название переменной не обязательно и можно его не писать, если оно не требуется, например: db 5

Просмотрите в эмуляторе результат работы директивы db, сделайте соответствующие выводы.

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 10,20,30,40,50

где array1 — название массива,

10,20,30,40,50 — значения элементов.

Просмотрите в эмуляторе результат работы директивы dw, сделайте соответствующие выводы.

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

array2 db 5 dup(1)

Просмотрите в эмуляторе результат работы директивы, сделайте соответствующие выводы.

Можно объявить массив и таким образом:

array3 dd 4 dup(3,7,0)

Просмотрите в эмуляторе результат работы, сделайте соответствующие выводы.

Объявление строк

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

str1 db '*Hello*'

Для обозначения конца строки используется специальный символ.

str3 db '*Hello*$' ;Для DOS

Резервирование данных (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(?)
```

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

Просмотрите в эмуляторе результат работы директив, сделайте соответствующие выводы.

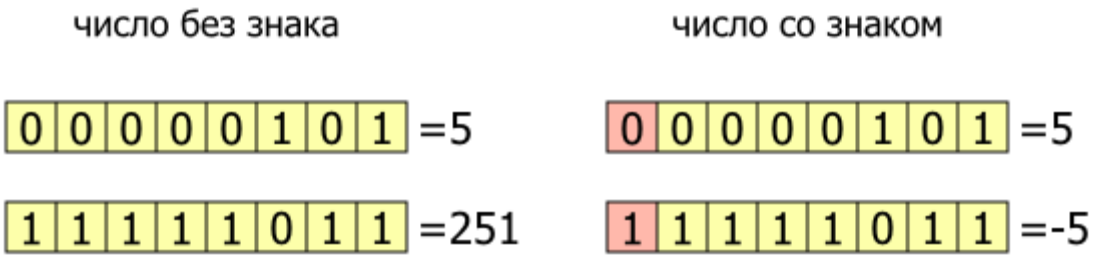
2 Операции со знаковыми и беззнаковыми величинами

1) Учет знаков в операциях

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

```
x db -5
y db -25h
z db -77o
k db -101b
```

Со знаковыми и беззнаковыми числами нужно быть внимательным, потому что только вы знаете, какие числа используются в вашей программе! Процессору абсолютно все равно, какие данные он обрабатывает, поэтому 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 знака

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 (11111111_2). Для такой же переменной со знаком максимальным значением будет 127 (01111111_2), а минимальным -128 (10000000_2). Аналогично определяется диапазон для 2- и 4-байтных переменных.

2) Вычитание беззнаковых величин

Вычитание выполняется с помощью команды SUB (subtract — вычесть). В остальном все этапы выполнения вычисления повторяют действия, которые были описаны для операции сложения.

В регистр AX заносится уменьшаемое, а в регистр BX — вычитаемое. Результат выполнения инструкции появится в регистре AX.

Команда SUB имеет следующий формат: SUB приемник, источник

Примеры:
mov al,10
sub al,7 ---> al = 3; al - приемник, 7 - источник

mov ax,25000
sub ax,10000 ---> ax = 15000; ax - приемник, 10000 - источник

mov cx,100
mov bx,15
sub cx,bx ---> cx = 85, bx = 15 (bx не меняется); cx - приемник, bx – источник

Задание 1.
Проверьте, как микропроцессор использует форму двоичного дополнения для представления отрицательных результатов. Выполните вычитание из нуля единицы (т.е., 0 – 1). Какой результат получен и почему?

Команда INC
Команда INC увеличивает на единицу регистр. Она эквивалентна команде
ADD источник, 1
только выполняется быстрее.

Примеры:
mov al,15
inc al ---> теперь AL = 16 (эквивалентна add al,1)

mov dh,39h
inc dh ---> DH = 3Ah (эквивалентна add dh,1)

mov cl,4Fh
inc cl ---> CL = 50h (эквивалентна add cl,1)

Команда DEC
Оператор dec уменьшает значение приемника на 1

Пример:
mov ah,12 ---> AH=12
dec ah ---> AH=11

3) Операции с байтами

В микропроцессорах Intel используются двухбайтовые машинные слова. Каждый регистр общего назначения (AX, BX, CX и DX) может хранить одно машинное слово. Однако имеется возможность оперировать с отдельными байтами этих регистров. В этом случае каждый регистр рассматривается состоящим из старшего (High) и младшего (Low) байтов. Обозначения отдельных байтов из регистров состоят из двух букв. Первая задает имя регистра (A, B, C или D), а вторая указывает, какой это байт регистра. Для обозначения старшего байта используется буква H, а младшего — L. Таким образом, регистр AX можно рассматривать, состоящим из двух однобайтовых регистров AH и AL.

Микропроцессор может выполнять арифметические операции над отдельными байтами.

Задание 2.

Введите в регистр AX число 0102h (два байта) и выполните инструкцию

ADD AH, AL

Сделать выводы каков результат выполнения операции, который будет помещен в регистр AH?

4) Умножение беззнаковых величин

Умножение двух 16-битных чисел может дать 32-разрядный результат, поэтому инструкция умножения MUL (multiply — умножить) размещает результат в двух регистрах DX и AX. Старшие 16 бит помещаются в регистр DX, а младшие в AX.

Замечание.

При выполнении операции умножения одним из множителей всегда является значение из регистра AX.

Задание 3.

Выполните умножение чисел 7C4Bh (в регистр AX) и 100h (BX).

Сделать выводы каков результат операции и почему?

5) Деление беззнаковых величин

Команды микропроцессора предназначены для выполнения целочисленных операций. Так как деление целых чисел нацело происходит

далеко не всегда, то результат деления формируется из двух целых чисел — частного и остатка от деления.

Делимое всегда помещается в пару регистров AX, DX, поэтому в инструкции деления DIV (divide — делить) необходимо указать только регистр с делителем. После выполнения деления регистр AX будет содержать частное, а регистр DX — остаток.

Задание 4.

Выполните деление числа 7C4B12h (DX=007Ch, AX=4B12h) на 0100h (BX).

Каков результат выполнения операции и почему?

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 придумаем какие-нибудь значения.

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

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

Задание 5

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.

Задание 6

Разработайте программу, реализующую указанную формулу, исполнить программу с несколькими наборами исходных данных, проверить правильность результатов.

1. X= A - 5 (B - 2C) + 2
2. X= - 4A + (B + C) / 4 + 2
3. X= 7A - 2B - 100 + C
4. X= - A / 2 + 4 (B + 1) + 3C
5. X= 5 (A - B) - 2C + 5
6. X= (A/ 2 + B) / 4 + C - 1
7. X= - (C + 2A + 4B + B)
8. X= 6C + (B - C + 1) / 2
9. X= 2 - B (A + B) + C / 4
10. X= 2B - 1 + 4 (A - 3C)
11. X= (2A + B) / 4 - C / 2 + 168
12. X= 6 (A - 2B + C / 4) + 10
13. X= 5 (A - B) + C * 4
14. X= - (- (C + 2A) * 4B + 38)
15. X= A - 3 (A + B) + C * 4
16. X= 3(A - 2B) +50 – C / 2
17. X= (3A + 2B) - C / 4 + 217
18. X= 3(C - 2A) + (B - C + 1) / 2
19. X= (2A + B) / 4 - C / 2 + 168
20. X= 6 (A - 2B + C / 4) + 10
21. X= 3 (A - 4B) + C / 4
22. X= - (- (C + 2A) * 5B - 27)
23. X= A / 2 - 3 (A + B) + C * 4
24. X= 3(A - 2B) +50 – C / 2
25. X= 5A + 2B - B / 4 + 131

3 Использование регистра флагов

1) Понятие переполнения регистра

Как и в случае умножения, при выполнении сложения результат может выходить за 16-разрядную сетку (четыре шестнадцатеричных числа). Например, результатом сложения четырехзначных чисел FFFFh и 1h будет пятизначное число 10000h, для записи которого слова (двух байт) недостаточно.

Если результат выполнения операции (над беззнаковыми величинами!) не может быть полностью размещен в регистре, то говорят о возникновении переполнения.

При выполнении сложения беззнаковых чисел суть переполнения (в двоичном представлении) состоит в том, что в результате сложения двух единиц в старшем разряде возникает единица, выходящая за разрядную сетку результирующего регистра. Естественно, что эта единица в регистр помещена быть не может, и при записи в регистр отсекается.

Задание 1.

Выполните сложение чисел FFFFh (AX) и 1h (BX). Каков результат операции?

Регистр флагов.

Флаг - это бит, принимающий значение 1 ("флаг установлен"), если выполнено некоторое условие, и значение 0 ("флаг сброшен") в противном случае. В ПК используется 9 флагов, причем конструктивно они собраны в один 16-разрядный регистр, называемый регистром флагов и обозначаемый как Flags. Эти биты обозначаются буквами C, P, A, Z, S, T, I, D, O. Например, в текстовый редактор загружен текст. Как только вы внесли в текст первое изменение, можно установить в 1 флаг изменений. После сохранения текста значение флага сбрасывается (0). Тогда при выходе из редактора легко проверить, сохранены ли изменения.

Флаг переноса

Если при сложении беззнаковых чисел происходит переполнение (возникает единица переноса за пределы разрядной сетки регистра), то единичка переноса записывается в Carry Flag. В правой половине окна регистров и флагов (Registers) данный флаг обозначается буквой C. Флаг переноса переустанавливается в каждой операции сложения.

Задание 2.

Проследите за изменением состояния флага переноса при последовательном выполнении следующих операций

1. FFFF + 1

2. FF00 + 1

Использование флага переноса

Сложение с использованием флага переноса.

Рассмотренная ранее инструкция сложения ADD выполняет простое сложение двух беззнаковых кодов. **Инструкция ADC** складывает три числа: два операнда из регистров общего назначения, как и раньше, плюс значение бита флага переноса из регистра флагов.

Задание 3.

а) Выполните сложение FFFFh и 1.

б) Затем выполните инструкцию: ADC BX, AX

В результате сложения 1 и 0, в регистре BX будет число 2. (поясните, что останется в регистрах после первой операции).

Вычитание с использованием флага переноса

При выполнении **инструкции SBB** из разности операндов вычитается значение флага переноса.

Задание.

а) Выполните сложение FFFFh и 1.

б) Затем выполните инструкцию: SBB BX, AX.

Поясните результат.

Флаг нуля.

Занесите в регистры BX и AX два равных числа, затем инструкцией SUB произведите вычитание одного числа из другого, в результате чего должен быть установлен флаг нуля Z=1 (Zero Flag).

Флаг знака.

Данный флаг позволяет узнать знак числа. Если вычесть из нуля единицу, то результат будет FFFFh, при этом устанавливается флаг знака S=1 (Sign Flag).

Флаг переполнения.

Флаг переполнения устанавливается в той ситуации, когда этого не должно было произойти. Занесите в регистр AX число 7000h, а в BX 6000h и выполните инструкцию сложения, в результате AX будет содержать число D000h или-12288. Это ошибка, так как результат переполняет слово и является отрицательным, поэтому микропроцессор устанавливает флаг переполнения O=1(Overflow Flag)

Контрольные вопросы

Понятие регистра микропроцессора и машинного слова.

Какая команда позволяет выполнять сложение целых чисел? Где размещаются операнды и результат?

Какова последовательность выполнения инструкции сложения чисел?

Какая инструкция позволяет выполнять вычитание целых чисел? Где размещаются операнды и результат?

В каком виде микропроцессор представляет отрицательные числа? Как будет представлен результат выполнения операции 5h – 8h?

Поясните особенности представления и именования двухбайтовых регистров общего назначения в виде совокупности двух однобайтовых.

Какими особенностями обладает инструкция умножения целых чисел? Где размещаются операнды и результат?

Какими особенностями обладает инструкция деления целых чисел? Где размещаются операнды и результат?

Поясните, что означает термин "переполнение". Как выяснить, что при выполнении операции произошло переполнение?

Что такое флаг, и для чего он нужен?

С помощью какой инструкции, и каким образом происходит сложение с учетом флага переноса?

С помощью какой инструкции, и каким образом происходит вычитание с учетом флага переноса?

Объясните назначение флагов переноса и нуля?

Объясните назначение флагов переполнения и знака?

4 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):

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

5 Организация логических сдвигов

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 будут циклические.

Логический сдвиг вправо

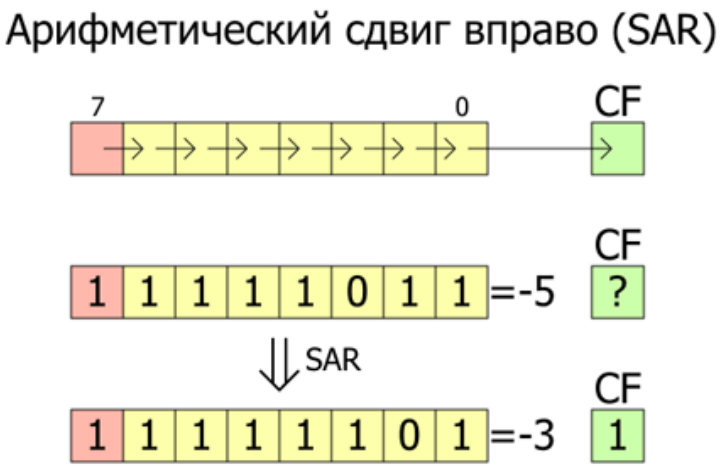
Логический сдвиг всегда выполняется без учёта знакового бита. Для логического сдвига вправо предназначена команда SHR. У этой команды два операнда. Первый операнд представляет собой сдвигаемое значение и на его место записывается результат операции. Второй операнд указывает, на сколько бит нужно осуществить 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 CL 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:

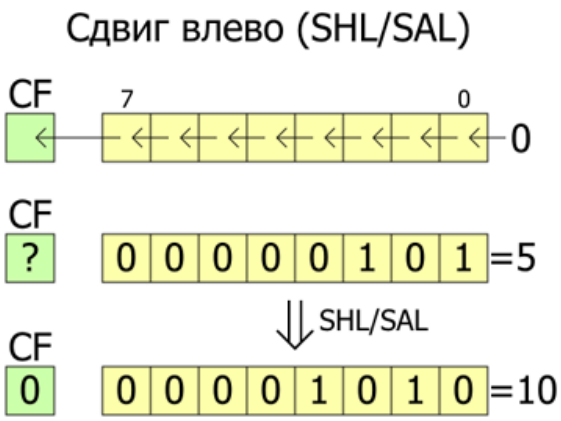


Выдвинутый бит становится значением флага CF. Знаковый бит не изменяется. При сдвиге на 1 бит сбрасывается флаг OF. Эту команду можно использовать для деления целых чисел со знаком на степень 2 (обратите внимание, что «округление» всегда в сторону меньшего числа, поэтому для отрицательных чисел результат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

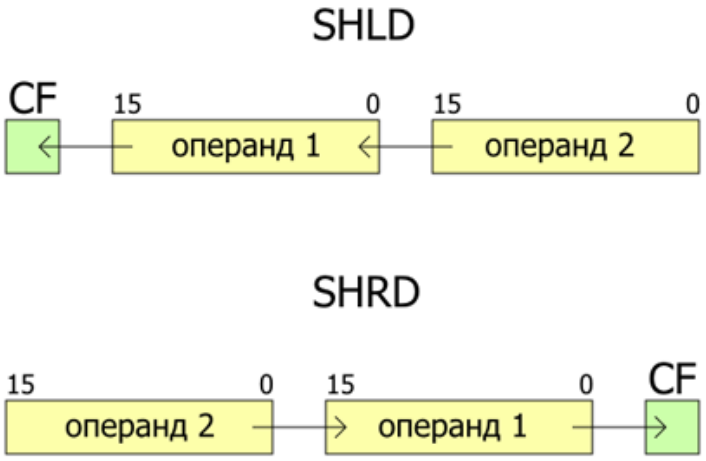


Старший бит становится значением флага CF, а младший получает нулевое значение. С помощью сдвига влево можно быстро умножать числа на степень 2. Но будьте внимательны, чтобы не получить в результате переполнение. Если при сдвиге на 1 бит меняется значение старшего бита, то устанавливается флаг OF. Примеры 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 CF=1 выводим символ ’1′. Проверка битов осуществляется в цикле.

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

6 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 я расскажу о всех возможных режимах адресации и приведу 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]
```

В данном примере в регистр 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. Я намеренно использовал разные режимы адресации, хотя тоже самое можно написать проще.

```
1  use16                    ;Генерировать 16-битный код
2  org 100h                 ;Программа начинается с адреса 100h
3
4        sub ax,ax          ;AX = 0
5        cwd                ;DX = 0
6        mov si,ax;SI = 0 - смещение элемента от начала массива
7        mov bx,array       ;Помещаем в BX адрес начала массива
8        mov di,n           ;Помещаем в DI адрес n
9        mov cx,[di]        ;CX = n
10 lp1:
11       add ax,[bx+si]     ;Прибавление младшего слова
```

```
12        adc dx,[bx+si+2]   ;Прибавление старшего слова
13        add si,4           ;Увеличиваем смещение в SI на 4
14        loop lp1           ;Команда цикла
15
16        idiv word[di]      ;Делим сумму на количество элементов
17        cwd                ;DX:AX = AX
18        mov word[m],ax     ;\ Сохраняем
19        mov word[m+2],dx   ;/ результат
20
21        mov ax,4C00h       ;\
22        int 21h            ;/ Завершение программы
23 ;-------------------------------------------------------
24 n      dw 10
25 array  dd 10500,-7500,-15000,10000,-8000
26        dd 6500,11500,-5000,10500,-20000
27 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.

7 Прерывания DOS. Организация циклов

1) Обработка символьной информации с помощью функций DOS

Прерыванием (interrupt), называется способ общения центрального процессора с периферийными устройствами. Периферийное устройство (клавиатура, дисковод и др.) посылает запрос на установление сеанса передачи, процессор прерывает выполнение основной программы и переходит на выполнение программы обработки запроса от периферийного устройства. Эта программа, *драйвер устройства*, предварительно загружена в память (резидентная программа) и её адрес известен процессору. Такая обработка запросов называется аппаратными прерываниями.

Программные прерывания и системные вызовы

Операционная система MS_DOS, как известно, является однозадачной операционной системой, т.е. одновременно может исполнять только одну задачу. Вместе с тем имеется необходимость во время выполнения основной задачи производить некоторые вспомогательные действия. Подход, основанный на предварительной загрузке резидентных программ, использованный в аппаратных прерываниях, которые можно вызывать в момент выполнения основной программы, например, переключение раскладки клавиатуры или обращение к дисководу, оказался очень продуктивным, так как делает MS-DOS псевдомногозадачной системой.

Вызовы резидентных программ, адрес которых заранее известен процессору, стали называть, по аналогии с аппаратными прерываниями, программными прерываниями, хотя никаких прерываний на самом деле не происходит. В систему команд процессора ввели команду вызова программного прерывания, которая вызывает соответствующую резидентную программу.

Команда вызова программного прерывания имеет вид

int <номер прерывания>

<номер прерывания> - число, обычно в шестнадцатеричное, в диапазоне 00h – 0FFh, определяет адрес вызываемой резидентной программы.

Некоторые резидентные программы, выполняющие низкоуровневое общение с периферийными устройствами записаны в ROM BIOS (Read Only Memory Base Input/Output System) и поставляются вместе с системной платой, например, учёт системного времени, форматирование секторов на дорожке диска и т.д., и не зависят от применяемой операционной системы.

Резидентные программы, использующие низкоуровневую систему резидентов BIOS и выполняющие более сложные задачи, например, файловые операции с диском, подгружаются в память при загрузке операционной системы. Их принято называть функциями операционной системы или системными вызовами.

Наибольшее число различных системных функций в MS-DOS сосредоточено в резидентной программе с номером прерывания 21h – диспетчер функций MS-DOS. В зависимости от значения, содержащегося при вызове прерывания в регистре **ah**, MS-DOS выполняет одну из нескольких десятков функций MS-DOS.

Все функции BIOS и DOS описаны в специальных справочниках с указанием для каждой функции набора входных и выходных параметров, передаваемых через регистры, а также перечнем возможных ошибок. В работе будут описаны функции прерывания 21h относящиеся к работе с клавиатурой и экраном ПЭВМ.

Описание функций работы с клавиатурой и дисплеем диспетчера функций MS-DOS

Для вызова функции прерывания DOS 21h необходимо проделать следующие действия:

- выбрать функцию, выполняющую требуемые действия;
- занести номер функции в регистр ah;
- подготовить другие регистры (если это необходимо);
- написать команду int 21h;
- прочесть результаты или состояние из регистров, указанных в описании данной функции.

Ниже следует описание некоторых функций 21H.

Функции 01H

Выполняет ввод с клавиатуры одного символа и отображает его на экране.

- Вызов: ah = 01h
- Возвращаемое значение: al = код ASCII введенного символа

Примечание. Введенный символ отображается на экране (выполняется эхо-отображение). Комбинация клавиш Ctrl/C (или Ctrl/Break) прекращает выполнение программ пользователя.

Функции 02H

Выполняет отображение символа на стандартный вывод (дисплей).

- Вызов: ah = 02h

dl = отображаемый символ

- Возвращаемое значение: нет

Примечание. Символ отображается на стандартный вывод. Комбинация клавиш Ctrl/C (или Ctrl/Break) прекращает выполнение программ пользователя.

Функция 05H

Выполняет отображение символа на принтер.

- Вызов:

ah = 02h

dl = символ для принтера

- Возвращаемое значение:

нет

Примечание. Символ отображается на принтер. Комбинация клавиш Ctrl/C (или Ctrl/Break) прекращает выполнение программ пользователя. Эта функция не возвращает ошибки состояния принтера.

Функция 07H

Выполняет ввод с клавиатуры одного символа.

- Вызов:

ah=07h

- Возвращаемое значение:

al = код ASCIL введенного символа

Примечание. Введенный символ не отображается на экране (не выполняется эхо-отображения). Комбинация клавиш Ctrl/C (или Ctrl/Break) прекращает выполнение программы пользователя.

Функция 08H

Выполняет ввод с клавиатуры одного символа.

- Вызов:

ah=08h

- Возвращаемое значение:

al = код ASCII введенного символа

Примечание. Введенный символ не отображается на экране (не выполняется эхо-отображение). Комбинация клавиш Ctrl/C (или Ctrl/Break) прекращает выполнение программы пользователя.

Функция 09H

Выполняет отображение строки на стандартный вывод.

- Вызов:

ah = 09H

ds: dx=указатель на отображаемую строку

Возвращаемое значение:

нет

Примечание: Строка отображается на стандартный вывод. $ признак конца строки, $ не отображается, dx содержит смещение строки, ds - сегментный адрес. Ниже приведены код управления курсором:

- 0dh (13) - перевод курсора в начало текущей строки;
- 0ah (10) - перевод курсора вниз на 1 строку;
- 08h (8) - перевод влево на 1 позицию;
- 07h (7) - звонок.

Пример.

Чтобы вывести на экран с новой строки текст: “Функция 09H для выдачи текста на экран” и затем перевести курсор в следующую строку, следует в сегменте данных описать строку:

beg db 0dh, 0ah, “Функция 09H для выдачи текста на экран”, 0dh,0ah, “$”

а в программном сегменте записать команды:

```
lea dx,beg          ; адрес строки  в dx
mov ah,09h ; номер функции  в ah
int 21h             ; вызов функции
```

Функция 0AH

Выполняет ввод с клавиатуры в буфер строки символов.

- Вызов:

ah= 0ah

ds: dx = адрес буфера ввода

- Возвращаемое значение:

Строка символов по указанному адресу

Примечание. Читается со стандартного ввода. dx содержит смещение буфер вывода, DS - сегментный адрес. Буфер вывода имеет следующую структуру: 0-й байт содержит максимальное количество символов в буфере; 1-й байт содержит количество реально введенных символов; начиная со 2-го размещён буфер для ввода размером не менее указанного в в 1-м байте. Выполняется эхо-отображение. Комбинация клавиш Ctrl/C (или Ctrl/Break) прекращает выполнение программы пользователя. Символы вводятся один за другим, до тех пор, пока не будет введен код ODh (код клавиш “Enter”), завершающий строку. В ходе ввода строки пользователь может редактировать строку, и, в частности, использовать “забой”.

Пример.

Пусть требуется ввести строку длиной не более 10 символов. При этом в сегменте данных можно описать буфер, например, таким образом:

```
buffer          db 11       ; Нулевой байт буфера
entered     db (?)          ; Число введенных символов
string          db 11 dup (?)       ; Введенные символы
```

Сам ввод выполняется командами:

```
lea dx, buffer          ; Адрес буфера  в dx
mov ah, 0ah             ; Номер функции в ah
int 21h                 ; Вызов функции
```

Функция 0Bh

Выполняет опрос состояния буфера клавиатуры.

- Вызов:

ah = 0Bh

- Возвращаемое значение:

al = 00h, если нет символа в буфере клавиатуры;
al = ffh, если есть символ в буфере клавиатуры.

Примечание. Устанавливает значение AL в зависимости от наличия символов в буфере клавиатуры. Часто используются в задачах, действующих при нажатии определенных клавиш. Комбинация клавиш Ctrl/ (или Ctrl/Break) прекращает выполнение программы пользователя.

2) Дополнительные сведения о структуре DOS и BIOS

Прямое обращение к видеопамяти

Видеопамять компьютера любой конфигурации расположена в адресном пространстве оперативного запоминающего устройства (ОЗУ). Это позволяет напрямую адресовать видеопамять одним из косвенных способов адресации памяти. Видеопамять занимает адреса с A000h по BFFFh, что составляет 128 Кбайт. Для увеличения объёмов видеопамяти (до 64 Мбайт), она делится на слои, так что по одному адресу находиться несколько ячеек, которые расположены в разных слоях. Обращение к видеопамяти зависит от видеорежима, который определяет количество точек по горизонтали и вертикали, а так же количество битов, отводимое для хранения кода цвета каждой точки. Графическими режимами управляет видеоадаптер.

Более простым для программирования, допускающим простой доступ к видеопамяти, является символьный режим, который мы и рассмотрим подробнее. Для работы в символьном режиме отводится 16 Кбайт памяти, начиная с адреса B800h. Экран делится на 80 столбцов и 25 строк. Общее количество знакомест 80 x 25= 2000. Для каждого знакоместа в видеопамяти отводится два байта: чётный байт – ASCII код символа, нечётный – байт атрибутов. Счёт строк и колонок идёт из верхнего левого угла экрана, в байте b800h:0000h хранится символ, выводящийся в нулевой строке и нулевой колонке, в байте b800h:0001h хранится атрибут этого символа. В байте b800h:0002h хранится символ,

выводящийся в нулевой строке и первой колонке, в байте b800h:0002h хранится атрибут этого символа и т.д.

Байт атрибутов имеет следующую структуру:

	Фон				Символ			
Атрибут	BL	R	G	B	I	R	G	B
Номер бита	7	6	5	4	3	2	1	0

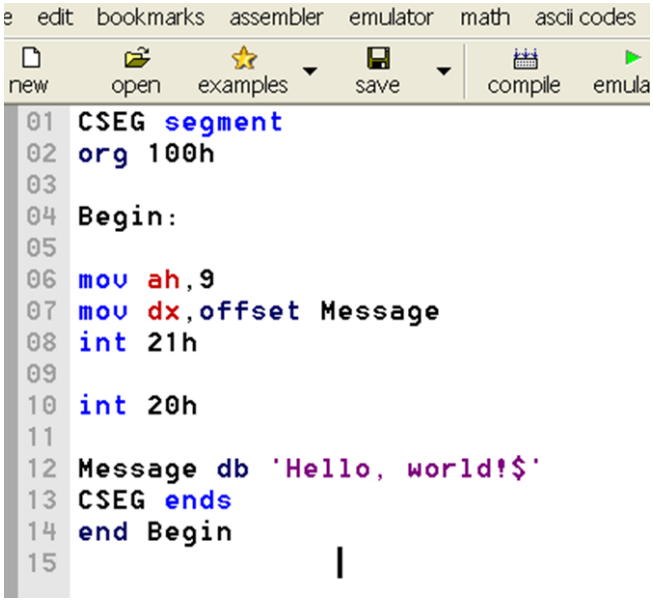
BL – признак мерцания; R – красный цвет;
G – зелёный цвет; B – синий цвет;
I – Интенсивность свечения.

Для доступа к видеопамяти в текстовом режиме можно использовать непосредственно один из сегментных регистров, например, ES:

```
mov ax, 0b800h          ; записать в регистр
mov   es, ax        ; es адрес начала видеопамяти
xor   bx, bx        ; смещение символа от начала видеопамяти
mov   dh, 00010100b     ; атрибуты: на голубом фоне красный символ
mov   dl, 65h           ; ASCII код символа
mov   word ptr es:[bx], dx ; запись в видеопамять символа и атрибута
inc   bx                ; смещение для
inc   bx                ; следующего символа
```

3) Пример использования прерываний

Ниже приведена программа, выводящая на экран строку «Hello world».



Со строки (4) начинается код программы. Это метка, указывающая Ассемблеру на начало кода. В строке (14) стоят операторы **end Begin**. Это конец программы. Вместо слова **Begin** можно было бы использовать что-нибудь другое. Например, **Start:**. В таком случае, нам пришлось бы и завершать программу **End Start** (14).

Строки (6) (8) выводят на экран сообщение Hello, world!

Функция 09h прерывания 21h выводит строку на экран, адрес которой указан в регистре DX.

В строку (7) загружаем в регистр DX адрес сообщения для вывода (в данном примере это будет строка **Hello, world!$**).

Ассемблер подставляет вместо **offset Message** реальный адрес строки с именем **Message** в памяти. Можно, конечно, записать сразу

mov dx, ***адрес строки с именем My_nate в памяти***

Программа будет работать нормально. Но для этого необходимо высчитать самим этот адрес.

Далее, в строке (8), вызываем прерывание MS-DOS, которое и выведет заданную строку на экран.

При выполнении строки (8) программы на Ассемблере вызываем подпрограмму (в данном случае это называется прерывание), которая выводит на экран строку. Для этого помещаем требуемые значения в регистры. Всю необходимую работу (вывод строки, перемещение курсора) берет на себя подпрограмма. Эту строку можно прочитать так: вызываем двадцать первое прерывание (int от англ. interrupt прерывание). Обратите внимание, что после числа 21 стоит буква h. Это шестнадцатеричное число (33 в десятичной системе). Можно заменить строку int 21h на int 33. Программа будет работать корректно. В Ассемблере принято указывать номер прерывания в шестнадцатеричной системе.

В строке (10) вызываем прерывание 20h. Для вызова данного прерывания нет необходимости указывать какие-либо значения в регистрах. Оно выполняет только одну задачу: выход из программы (выход в DOS). В результате выполнения прерывания 20h, программа вернется туда, откуда ее запускали (загружали, вызывали). Например, в Norton Commander или DOS Navigator.

Строка (12) содержит сообщение для вывода. Первое слово (message сообщение) название сообщения. Оно может быть любым (например, mess или string и пр.). Обратите внимание на строку (7), в которой загружен в регистр DX адрес сообщения.

Последний символ в строке Message указывает на конец строки. Если его уберать, то 21h прерывание продолжит вывод до тех пор, пока не встретится где-нибудь в памяти символ $. На экране появятся случайные символы.

4) 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. 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.

5) Создание циклов

Допустим, необходимо выполнить некоторый код программы несколько раз. Возьмем, к примеру, вывод строки функцией 09h прерывания 21h:

Пример 1.

```
mov ah,9
mov dx,offset Str
int 21h
mov ah,9
mov dx,offset Str
int 21h
mov ah,9
mov dx,offset Str
int 21h
```

Этот участок кода выведет 3 раза на экран некую строку Str. Код получается громоздким, неудобно читать. Размер программы разрастается. Для выполнения подобных примеров используется оператор 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 приведена программа, которая печатает 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   org 100h                ;Программа начинается с адреса 100h
3
4       mov ah,02h          ;Для вызова функции DOS 02h - вывод
    символа
5       mov dl,'A'          ;Первый выводимый символ
6       mov cx,26           ;Счётчик повторений цикла
7   metka:
8       int 21h             ;Обращение к функции DOS
9       inc dl              ;Следующий символ
10      loop metka          ;Команда цикла
11
12      mov ah,09h          ;Функция DOS 09h - вывод строки
13      mov dx,press        ;В DX адрес строки
14      int 21h             ;Обращение к функции DOS
15
16      mov ah,08h          ;Функция DOS 08h - ввод символа без эха
17      int 21h             ;Обращение к функции DOS
18
19      mov ax,4C00h        ;\
20      int 21h             ;/ Завершение программы
21  ;-------------------------------------------------------
22  press:
23      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 древних телетайпов, когда текст печатался, как на печатной машинке). Так выглядит результат работы программы:

Задание 1.

Измените данную программу таким образом, чтобы она вывела на экран символы английского алфавита в обратном порядке. То же самое проделайте для графических символов ASCII.

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

Результат работы программы выглядит вот так:

Задание 2

Напишите программу для вычисления степени числа 3 по формуле $a = 3^n$. 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). Выведите результат на экран.

8 Организация циклов с фиксированным количеством итераций и условием досрочного завершения

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:

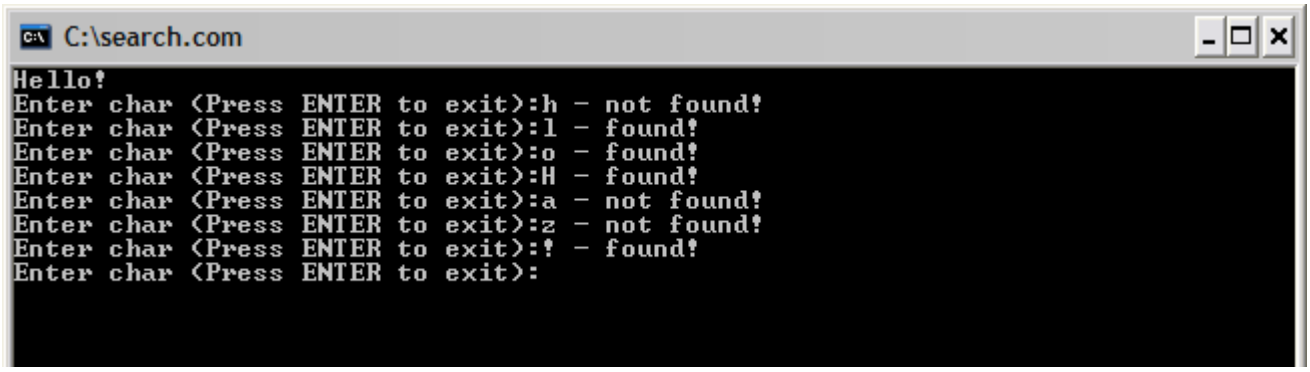
- выполнено требуемое количество итераций;
- выполнено некоторое условие досрочного завершения цикла.

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   use16                       ;Генерировать 16-битный код
2   org 100h                  ;Программа начинается с адреса 100h
3       jmp start            ;Безусловный переход на метку start
4   ;-- Данные ------------------------------------------------
5   string    db 'Hello!',13,10,'$'
6   length    db 6
7   s_entchar db 'Enter char (Press ENTER to exit):$'
8   s_found   db ' - found!',13,10,'$'
9   s_nfound  db ' - not found!',13,10,'$'
10  ;-------------------------------------------------------
11  start:
12      mov ah,09h              ;\
13      mov dx,string           ; > Вывод строки
14      int 21h                 ;/
15  main:
16      mov dx,s_entchar        ;\
17      int 21h                 ;/ Вывод приглашения для ввода
18  символа
19      mov ah,01h              ;\
20      int 21h                 ;/ Ввод символа
21      cmp al,0Dh              ;Нажата клавиша ENTER?
22      je exit                 ;Если да, то переход на метку exit
```

```
23
24      mov bx,string-1      ;BX = (адрес строки - 1)
25      movzx cx,[length]    ;CX = длина строки
26  search:
27      inc bx               ;Инкремент BX
28   mp al,[bx];Сравнение введённого символа с символом строки
29      loopne search        ;Цикл, если не равно.
30      je found             ;Если равно, то символ найден.
31
32      mov dx,s_nfound      ;DX = адрес строки ' - not found!'
33  print_result:
34      mov ah,09h           ;\
35      int 21h              ;/ Вывод результата поиска
36      jmp main             ;Безусловный переход на метку main
37  found:
38      mov dx,s_found       ;DX = адрес строки ' - found!'
39      jmp print_result     ;Безусловный переход на метку
40  print_result
    exit:
41      mov ax,4C00h         ;\
42      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.

9 Безусловные и условные переходы

Трудно представить себе программу без проверки условий и переходов. С их помощью в программе реализуются различные управляющие конструкции, ветвления и даже циклы.

1. 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
```

2. Условные переходы

Условный переход осуществляется, если выполняется определённое условие, заданное флагами процессора (кроме одной команды, которая проверяет CX на равенство нулю). Как вы помните, 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 приведена программа 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   use16                   ;Генерировать 16-битный код
2   org 100h                ;Программа начинается с адреса 100h
3
4       mov al,[x]          ;AL = x
5       add al,[y]          ;AL = x + y
6       jo error            ;Переход, если переполнение
7       mov ah,09h          ;\
8       mov dx,ok_msg       ; > Вывод строки 'OK'
9       int 21h             ;/
10  exit:
11      mov ah,09h          ;\
12      mov dx,pak          ; > Вывод строки 'Press any key...'
13      int 21h             ;/
14
```

```
15      mov ah,08h          ;\
16      int 21h             ;/ Ввод символа
17
18      mov ax,4C00h        ;\
19      int 21h             ;/ Завершение программы
20  error:
21      mov ah,09h          ;\
22      mov dx,err_msg      ; > Вывод сообщения об ошибке
23      int 21h             ;/
24      jmp exit            ;Переход на метку exit
25  ;-------------------------------------------------------
26  x           db -89
27  y           db -55
28  err_msg     db 'Error: overflow detected.',13,10,'$'
29  ok_msg      db 'OK',13,10,'$'
30  pak         db 'Press any key...$'
```

Задание 1.

Проверьте работу программы. Поясните, какие конкретные функции в программе выполняет каждый оператор перехода.

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

Программа 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 участок кода. Обратите внимание, что данные помещены в начале программы, а не в конце. Чтобы данные не выполнились как код, перед ними стоит команда безусловного перехода.

```
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:
```

```
43      mov ax,4C00h            ;\
44      int 21h                 ;/ Завершение программы
```

Скриншот работы программы:

Задание 2.

Проверьте работу программы. Поясните, какие конкретные функции в программе выполняет каждый оператор перехода.

Задание 3.

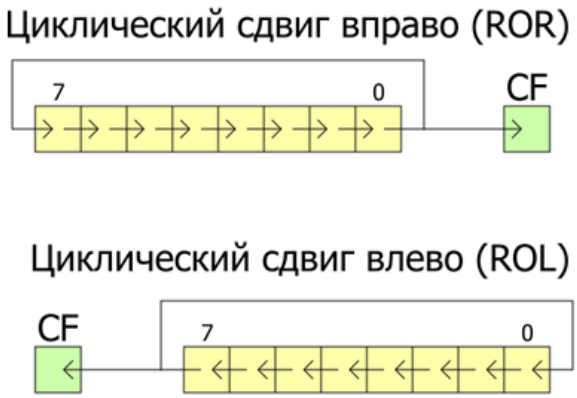
Напишите программу для сравнения двух переменных со знаком *a* и *b*. В зависимости от результатов сравнения выведите «a < b», «a > b» или «a = b». Проверьте работу программы в отладчике.

10 Организация циклических сдвигов

Циклический сдвиг отличается от линейного тем, что выдвигаемые с одного конца биты вдвигаются с другой стороны, то есть движутся по кольцу. В процессоре 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 находиться в регистре CL или указываться непосредственно.

1) 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):

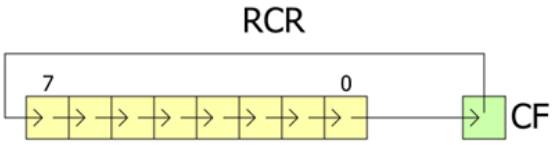


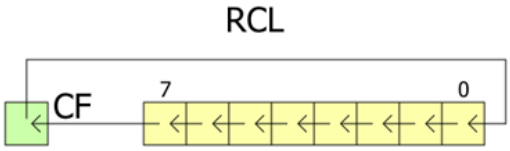
Значение последнего выдвигаемого бита копируется в флаг CF. Для сдвигов на 1 бит устанавливается флаг OF, если в результате сдвига изменяется знаковый бит операнда. Примеры использования команд:

```
    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
```

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

Всего 6 команд для подсчета битов в байте. Значительно сложнее написать то же самое на языке высокого уровня.

Упражнение

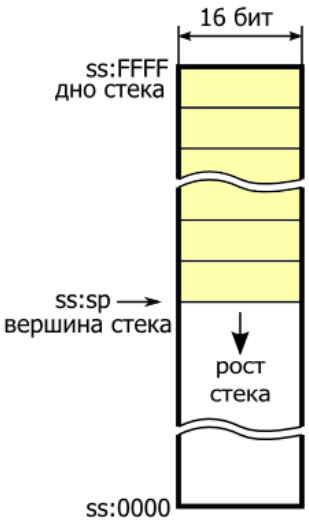
Объявите в программе строку. Длина строки должна быть больше 8 символов и храниться в байте без знака. Напишите цикл для шифрования строки по алгоритму: первый символ циклически сдвигается вправо на 1 бит, второй символ – на 2 бита, …, 7-й – на 7 битов, 8-й – снова на 1 бит, 9-й на 2 бита и т.д. Затем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

11 Организация работы со стеком

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:

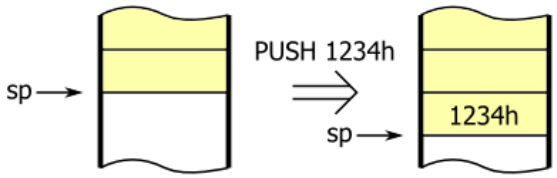
- добавление элемента на вершину стека (PUSH);

- 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
```

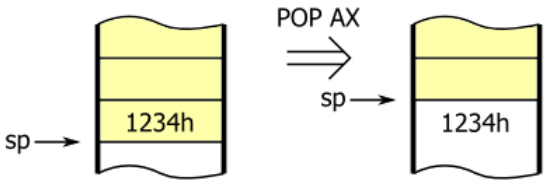
Существуют ещё 2 команды для добавления в стек. Команда PUSHF помещает в стек содержимое регистра флагов. Команда PUSHA помещает в стек содержимое всех регистров общего назначения в следующем порядке: AX, CX, DX, BX, SP, BP, SI, DI (значение DI будет на вершине стека). Значение SP помещается то, которое 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   use16                   ;Генерировать 16-битный код
2   org 100h                ;Программа начинается с адреса 100h
3       jmp start           ;Переход к метке start
4   ;-----------------------------------------------------------
5   ; Данные
6   n   db 4                ;Количество строк
7   m   db 5                ;Количество столбцов
8   ;Двумерный массив - таблица с данными
9   table:
10      dw 12,45, 0,82,34
11      dw 46,-5,87,11,56
12      dw 35,21,77,90,-9
13      dw 44,13,-1,99,32
14  sum rw 4                ;Массив для сумм каждой строки
15  ;-----------------------------------------------------------
16  start:
17      movzx cx,[n]        ;Счётчик строк
18      mov bx,table        ;BX = адрес таблицы
19      mov di,sum          ;DI = адрес массива для сумм
20      xor si,si       ;SI = смещение элемента от начала таблицы
21
22  rows:
23      xor ax,ax       ;Обнуление AX. В AX будет считаться сумма
```

```
24      push cx             ;Сохранение значения CX
25
26      movzx cx,[m]    ;Инициализация CX для цикла по строке
27  calc_sum:
28      add ax,[bx+si]      ;Прибавление элемента строки
29      add si,2            ;SI = смещение следующего элемента
30      loop calc_sum       ;Цикл суммирования строки
31
32      pop cx              ;Восстановление значения CX
33      mov [di],ax         ;Сохранение суммы строки
34      add di,2 ;DI = адрес следующей ячейки для суммы строки
35      loop rows           ;Цикл по всем строкам таблицы
36
37      mov ax,4C00h        ;\
38      int 21h             ;/ Завершение программы
```

Как видите, 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.

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.

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

```
21  ;-----------------------------------------------------------
22  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.

В данном учебном курсе используются 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. 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.

Ниже приведен пример программы:

```
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 передавтся 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   use16                   ;Генерировать 16-битный код
2   org 100h                ;Программа начинается с адреса 100h
3       jmp start           ;Переход на метку start
4   ;-------------------------------------------------------------------------
5   msg1    db 'Hello!$'
6   msg2    db 'asmworld.ru$'
7   msg3    db 'Press any key...$'
8   ;-------------------------------------------------------------------------
9   start:
10      mov bx,msg1
11      call print_message  ;Вывод первого сообщения
12      mov bx,msg2
13      call print_message  ;Вывод второго сообщения
14      mov bx,msg3
15      call print_message  ;Вывод третьего сообщения
16
17      mov ah,8            ;Ввод символа без эха
18      int 21h
19
20      mov ax,4C00h        ;\
21      int 21h             ;/ Завершение программы
22
```

```
23 ;-------------------------------------------------------------------------
24 ;Процедура вывода сообщения в рамке
25 ;В BX передаётся адрес строки
26 print_message:
27     push ax             ;Сохранение регистров
28     push cx
29     push dx
30
31     call get_length     ;Вызов процедуры вычисления длины строки
32     mov cx,ax           ;Копируем длину строки в CX
33     mov ah,2            ;Функция DOS 02h - вывод символа
34     mov dl,0xDA         ;Левый верхний угол
35     int 21h
36     mov dl,0xC4         ;Горизонтальная линия
37     call draw_line      ;Вызов процедуры рисования линии
38     mov dl,0xBF         ;Правый верхний угол
39     int 21h
40     call print_endline  ;Вызов процедуры вывода конца строки
41
42     mov dl,0xB3         ;Вертикальная линия
43     int 21h
44     mov ah,9            ;Функция DOS 09h - вывод строки
45     mov dx,bx           ;Адрес строки в DX
46     int 21h
47     mov ah,2            ;Функция DOS 02h - вывод символа
48     mov dl,0xB3         ;Вертикальная линия
49     int 21h
50     call print_endline  ;Вызов процедуры вывода конца строки
51
52     mov dl,0xC0         ;Левый нижний угол
53     int 21h
54     mov dl,0xC4         ;Горизонтальная линия
55     call draw_line
56     mov dl,0xD9         ;Правый нижний угол
57     int 21h
58     call print_endline  ;Вызов процедуры вывода конца строки
59
60     pop dx              ;Восстановление регистров
61     pop cx
62     pop ax
63     ret                 ;Возврат из процедуры
64
65 ;-------------------------------------------------------------------------
66 ;Процедура вычисления длины строки (конец строки - символ '$').
67 ;В BX передаётся адрес строки.
68 ;Возвращает длину строки в регистре AX.
69 get_length:
70     push bx             ;Сохранение регистра BX
71     xor ax,ax           ;Обнуление AX
72 str_loop:
73     cmp byte[bx],'$'    ;Проверка конца строки
74     je str_end          ;Если конец строки, то выход из процедуры
75     inc ax              ;Инкремент длины строки
76     inc bx              ;Инкремент адреса
77     jmp str_loop        ;Переход к началу цикла
78 str_end:
79     pop bx              ;Восстановление регистра BX
80     ret                 ;Возврат из процедуры
81
82 ;-------------------------------------------------------------------------
83 ;Процедура рисования линии из символов.
84 ;В DL - символ, в CX - длина линии (кол-во символов)
85 draw_line:
```

```
86      push ax             ;Сохранение регистров
87      push cx
88      mov ah,2            ;Функция DOS 02h - вывод символа
89  drl_loop:
90      int 21h             ;Обращение к функции DOS
91      loop drl_loop       ;Команда цикла
92      pop cx              ;Восстановление регистров
93      pop ax
94      ret                 ;Возврат из процедуры
95
96  ;-------------------------------------------------------------------------
97  ;Процедура вывода конца строки (CR+LF)
98  print_endline:
99      push ax             ;Сохранение регистров
100     push dx
101     mov ah,2            ;Функция DOS 02h - вывод символа
102     mov dl,13           ;Символ CR
103     int 21h
104     mov dl,10           ;Символ LF
105     int 21h
106     pop dx              ;Восстановление регистров
107     pop ax
108     ret                 ;Возврат из процедуры
```

Результат работы программы выглядит вот так:

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

Содержание отчета

Отчет по проведенному лабораторному исследованию выполняется в электронном виде в текстовом редакторе. В документе должны быть содержаться следующие элементы:

1. Титульный лист, включающий название дисциплины, название лабораторного исследования, дату выполнения, название учебной группы, Ф.И.О., выполнившего(их) работу.

2. Описание отдельных пунктов лабораторного исследования с соответствующими скриншотами и выводами по каждому пункту проведенного исследования.

3. В конце работы должен прилагаться перечень файлов программ, написанных на ассемблере, которые использовались при проведении исследования.

При сдаче отчета по лабораторному исследованию студент должен быть готов ответить на вопросы преподавателя, возникающие при проверке отчета.

Набор файлов программ на ассемблере и файл отчета в текстовом редакторе сдаются в электронном виде преподавателю, принимающему отчет.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Аблязов Р.З. Программирование на ассемблере на платформе x86-64. Саратов: Профобразование, 2019. — 301 с. — ISBN 978-5-4488-0117-4

2. Гагарина Л.Г., Кононова А.И. Архитектура вычислительных систем и Ассемблер с приложением методических указаний к лабораторным работам: учебное пособие. Москва: СОЛОН-Пресс, 2019. — 368 с. — ISBN 978-5-91359-321-4

3. Юров В.И. Assembler. СПб.: Питер, 2010. — 637 с.

4. Юров В.И. Assembler: практикум. СПб.: Питер, 2002. — 400 с.

5. Зубков С.В. Assembler для DOS, Windows и UNIX. М.: ДМК Пресс, 2004 – 608 с.

6. Магда Ю.С. Ассемблер для процессоров Intel Pentium. СПб.: Питер, 2006. - 410 с.

7. Крупник А. Ассемблер. Самоучитель. – СПб.: Питер, 2005. – 235 с.

8. Калашников О.А. Ассемблер – это просто. Учимся программировать. СПб.: БХВ-Петербург, 2011. – 336 с.

9. Зубарев А.А. Ассемблер для микроконтроллеров AVR: Учебное пособие. – Омск: Изд-во СибАДИ, 2007. – 112 с.