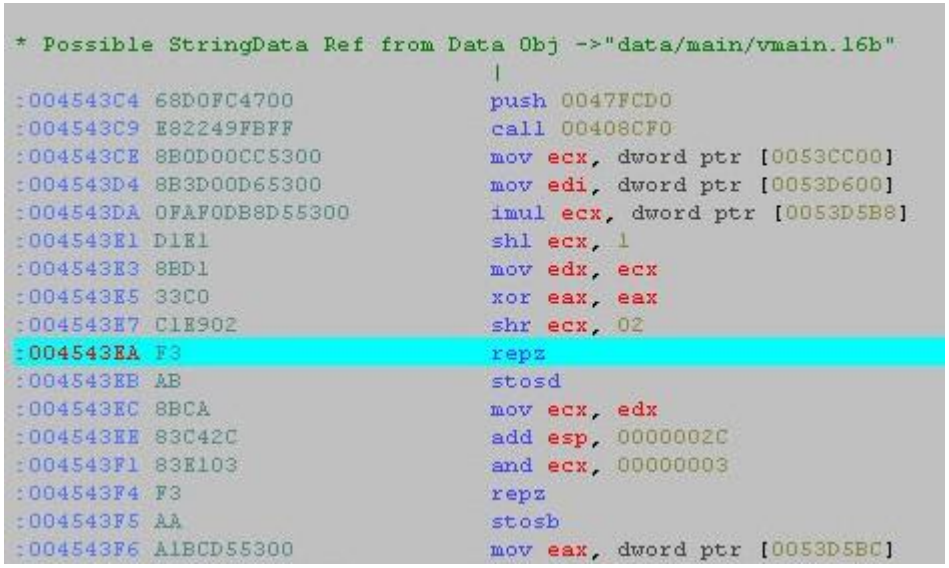
Арифметическиеи логические команды языка Ассемблер. Битовые команды.



```
* Possible StringData Ref from Data Obj ->"data/main/vmain.16b"
                                        |
:004543C4 68D0FC4700                    push 0047FCD0
:004543C9 E82249FBFF                    call 00408CF0
:004543CE 8B0D00CC5300                  mov ecx, dword ptr [0053CC00]
:004543D4 8B3D00D65300                  mov edi, dword ptr [0053D600]
:004543DA 0FAF0DB8D55300                imul ecx, dword ptr [0053D5B8]
:004543E1 D1E1                          shl ecx, 1
:004543E3 8BD1                          mov edx, ecx
:004543E5 33C0                          xor eax, eax
:004543E7 C1E902                        shr ecx, 02
:004543EA F3                            repz
:004543EB AB                            stosd
:004543EC 8BCA                          mov ecx, edx
:004543EE 83C42C                        add esp, 0000002C
:004543F1 83E103                        and ecx, 00000003
:004543F4 F3                            repz
:004543F5 AA                            stosb
:004543F6 A1BCD55300                    mov eax, dword ptr [0053D5BC]
```

Инструкции сложения ADD и вычитания SUB

Команда **ADD** требует двух операндов, как и команда MOV:

```
ADD o1, o2
```

Команда **ADD** складывает оба операнда и записывает результат в o1, предыдущее значение которого теряется.
Точно так же работает команда вычитания — **SUB**:

```
SUB o1, o2
```

Результат, o1-o2, будет сохранен в o1, исходное значение o1 будет потеряно.

```
mov ax, 8    ; заносим в AX число 8
mov cx, 6    ;заносим в CX число 6
mov dx, cx   ;копируем CX в DX, DX = 6
add dx, ax   ;DX = DX + AX
```

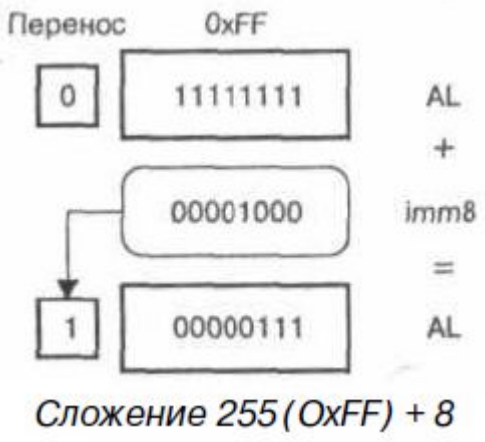
*Команда **ADD** сохранит результат DX + AX в регистре DX, а исходные значения AX и CX останутся нетронутыми.*

Инструкции сложения ADD и вычитания SUB

```
add ax, 8          ;AX = AX + 8
sub cx, bp         ; CX = CX - BP
add number, 4      ;добавляем значение 4
                   ;к переменной number
sub number, 4      ;number = number — 4
sub number, al     ;вычитаем значение регистра AL
                   ;из "number"
sub ah,al          ;вычитаем AL из AH, результат
                   ;помещаем в AH
```

Что произойдет, если сначала занести в AL (8-разрядный регистр) наибольшее допустимое значение (255), а затем

```
mov al, 255 ; заносим в AL значение 255,
            ;то есть OxFF
add al, 8   ;добавляем 8
```



Сложение 255 (OxFF) + 8

В результате в регистре AL мы получим значение 7. Девятый, ≪потерянный≫, бит скрыт в регистр признаков,
а именно в флаге CF — признак переноса.

Инструкции сложения ADC и вычитания SBB

Команды **ADC** (Add With Carry — сложение с переносом) и **SBB** (Subtract With Borrow — вычитание с займом):

```
ADC o1, o2    ;o1 = o1 + o2 + CF
SBB o1, o2    ;o1 = o1 - o2 - CF
```

Эти команды работают так же, как ADD и SUB, но соответственно добавляют или вычитают флаг переноса CF.

В контексте арифметических операций очень часто используются так называемые **пары регистров**.

Пара — это два регистра, использующихся для хранения одного числа.

Часто используется пара **DX:AX** — обычно при умножении.

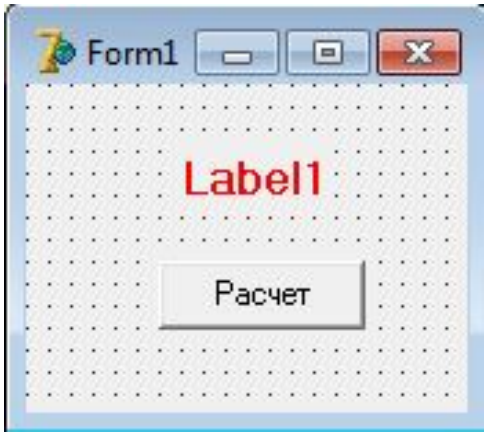
Регистр **AX** хранит младшие 16 битов числа, а **DX** — старшие 16 битов.

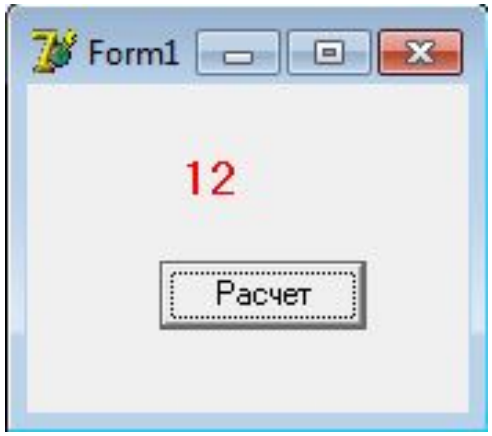
```
mov ax, 0xffff  ;AX = 0xFFFF
mov dx, 0       ;DX = 0
add ax, 8       ;AX = AX + 8
adc dx, 0       ;добавляем 0 с переносом к DX
```

После выполнения **ADC** флаг **CF** будет добавлен к DX (DX теперь равен 1).

Результат сложения 0xFFFF и 8 (0x10007) будет помещен в пару **DX:AX** (DX=1,AX=0007).

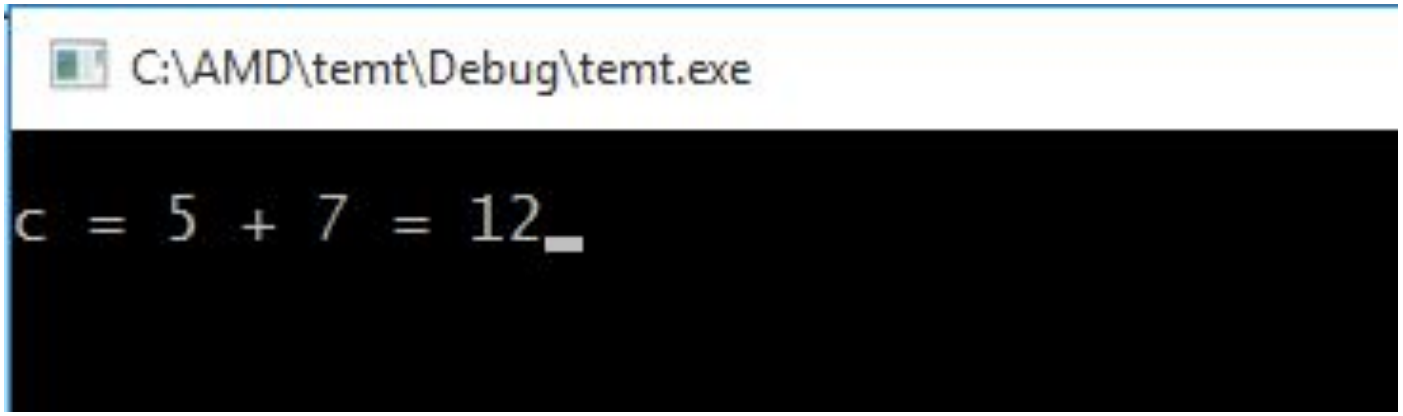
Инструкции сложения ADD и вычитания SUB






```
procedure TForm1.Button1Click(Sender: TObject);
var
  x:integer;
begin
  asm
    mov x, 5
    mov eax, 7
    add eax, x
    mov x, eax
  end;
  Label1.Caption:=IntToStr(x);
end;
```

Инструкции сложения ADD и вычитания SUB




```
int a, b, c;
 __asm
 {
   mov a, 5
   mov b, 7
   mov eax, a
   add eax, b
   mov c, eax
 }
 cout << "c = " << a << " + " << b << " = " << c;
```

Команды инкрементирования INC и декрементирования DEC

Эти команды предназначены для инкрементирования и декрементирования.

Команда **INC** добавляет, а **DEC** вычитает единицу из единственного операнда.

Допустимые типы операнда — такие же, как у команд ADD и SUB, а формат команд таков:

```
INC o1      ;o1 = o1 + 1
DEC o1      ;o1 = o1 - 1
```

Ни одна из этих инструкций не изменяет флаг CF.

```
add al,1    ;AL = AL + 1
inc al      ;AL = AL + 1
Inc number  ;number = number+1
```

Отрицательные числа

Отрицательные целые числа в ПК представлены дополнительном коде.

Один байт может содержать числа в диапазоне от **0** до **255**. Код дополнения
заменяет этот диапазон другим — от **-128** до **127**. Диапазон от **0** до **127** отображается сам на себя, а **отрицательным** числам сопоставляется диапазон от 128 до 255: числу -1 соответствует число **255 (OxFF)**, -2 — **254 (OxFE)** и т.д.

Дополнительный код может быть расширен до 2 байтов (от 0 до 65535). Он будет охватывать диапазон чисел от -32768 до 32767. Если
Число -50 будет представлено как 206.
дополнительный код

Рассмотрим, как это происходит, на примере суммы чисел **-6** и **7** в дополнительном коде из 2 байтов. Число **7** будет отображено само в себя, а число **-6** будет представлено числом **65 536 — 6 = 65 530 (OxFFFA)**.

```
mov ax,OxFFFA ;AX = -6, то есть 65530 или OxFFFA
mov dx,7      ;DX = 7
add ax,dx     ;AX = AX + DX
```

Мы получим результат **65 530 + 7 = 65 537 = 0x10001**, который не помещается в регистре AX, поэтому будет установлен флаг переноса. Но если мы его проигнорируем, то оставшееся в AX значение **будет правильным результатом!**

Отрицательные числа

Ассемблер позволяет указывать отрицательные числа непосредственно, поэтому не нужно преобразовывать их вручную в дополнитель

```
mov ax, -6          ;AX = -6
mov dx, -6          ;DX = - 6
add ax,dx           ;AX = AX + DX
```

Результат: **0xFFF4** (установлен также флаг CF, но мы его игнорируем). В десятичной системе **0xFFF4** = **65 524**. В дополнительном коде мы получим правильный результат: **-12 (65 536 — 65 524 = 12)**.

Механизм дополнительного кода ввели именно для того, чтобы при сложении и вычитании отрицательных чисел не приходилось выполнять дополнительных действий.

Команда NEG

Используя **NEG**, можно преобразовывать положительное целое число в отрицательное и наоборот. Инструкция **NEG** имеет только один операнд, который может быть регистром или адресом памяти.

```
neg ax      ;изменяет знак числа, сохраненного в AX
neg bl      ;то же самое, но используется 8-битный регистр
bl
neg number  ;изменяет знак переменной number
```

Команды MUL и IMUL

Команда **MUL** может быть записана в трех различных форматах — в зависимости от операнда:

```
MUL r/m8
MUL r/m16
MUL r/m32
```

В 8-разрядной форме операнд может быть любым 8-битным регистром или адресом памяти. Второй операнд всегда хранится в **AL**. Результат (произведение) будет записан в регистр **AX**.

```
(r/m8) * AL -> AX
```

В 16-разрядной форме операнд может быть любым 16-битным регистром или адресом памяти. Второй операнд всегда хранится в **AX**. Результат сохраняется в паре **DX:AX**.

```
(r/m16) * AX -> DX:AX
```

В 32-разрядной форме второй операнд находится в регистре **EAX**, а результат записывается в пару **EDX:EAX**.

```
(r/m32) * EAX -> EDX:EAX
```

Команды MUL и IMUL

Пример 1: умножить значения, сохраненные в регистрах BH и CL, результат сохранить в регистр AX:

```
mov al, bh    ;AL = BH – сначала заносим в AL второй операнд
mul cl        ;AX = AL * CL – умножаем его на CL
```

Результат будет сохранен в регистре AX.

Пример 2: вычислить 486^2, результат сохранить в DX:AX:

```
mov ax, 486 ; AX = 486
mul ax      ; AX * AX -> DX:AX
```

Пример 3: вычислить диаметр по радиусу, сохраненному в 8-битной переменной radius, результат записать в 16-битную переменную diameter:

```
mov al, 2          ; AL = 2
mul radius         ; AX = radius * 2
mov diameter,ax    ; diameter <- AX
```

Команды MUL и IMUL

Команда **IMUL** умножает целые числа со знаком и может использовать один, два или три операнда.

Когда указан ***один операнд***, то поведение **IMUL** будет таким же, как и команды **MUL**, просто она будет работать с числами со знаком.

Если указано два операнда, то инструкция **IMUL** умножит первый операнд на второй и сохранит результат в первом операнде, поэтому первый операнд всегда должен быть **регистром**. Второй операнд может быть регистром, непосредственным значением или адресом памяти.

```
imul edx,ecx       ;EDX = EDX * ECX
imul ebx, sthing   ;умножает 32-разрядную переменную sthing
                   ;на EBX, результат будет сохранен в EBX
imul ecx,6         ;ECX = ECX * 6
```

Если указано три операнда, то команда **IMUL** перемножит второй и третий операнды, а результат сохранит в первый операнд. Первый операнд — только **регистр**, второй может быть **любого типа**, а третий должен быть только **непосредственным значением**:

```
imul ebx,[sthing],9 ;умножаем переменную "sthing" на 9,
                    ;результат будет сохранен EBX
imul ecx,edx,11     ;ECX = EDX * 11
```

Команды DIV и IDIV

Подобно команде **MUL**, команда **DIV** может быть представлена в трех различных форматах в зависимости от типа операнда:

```
DIV r/m8
DIV r/m16
DIV r/m32
```

Операнд служит делителем, а делимое находится в фиксированном месте (как в случае с MUL).

В 8-битной форме переменный операнд (делитель) может быть любым 8-битным регистром или адресом памяти. Делимое содержится в **AX**. Результат сохраняется так: частное — в **AL**, остаток — в **AH**.

```
AX / (r/m8) -> AL, остаток -> AH
```

В 16-битной форме операнд может быть любым 16-битным регистром или адресом памяти. Второй операнд всегда находится в паре **DX:AX**. Результат сохраняется в паре **DX:AX** (DX — остаток, AX — частное).

```
DX:AX / (r/m16) -> AX, остаток -> DX
```

В 32-разрядной форме делимое находится в паре **EDX:EAX**, а результат записывается в пару **EDX:EAX** (частное в EAX, остаток в EDX).

```
EDX:EAX / (r/m32) -> EAX, остаток -> EDX
```

Команды DIV и IDIV

Команда **IDIV** используется для деления чисел со знаком, синтаксис ее такой же, как у команды **DIV**.

Пример 1: разделить 13 на 2, частное сохранить в BL, а остаток в — BH:

```
mov ax,13 ;AX = 13
mov cl,2  ;CL = 2
div cl    ;делим на CL
mov bx,ax ;ожидаемый результат находится в AX, копируем в BX
```

Пример 2: вычислить радиус по диаметру, значение которого сохранено в 16-битной переменной diameter, результат записать в radius, а остаток проигнорировать.

```
mov ax, diameter        ;AX = diameter
mov bl, 2               ;загружаем делитель 2
div bl                  ;делим
mov radiusl, al         ;сохраняем результат
```

Команда AND

Команда **AND** выполняет логическое умножение двух операндов — **o1** и **o2**. Результат сохраняется в операнде **o1**. Типы операндов такие же, как у команды **ADD**: операнды могут быть 8-, 16- или 32-битными регистрами, адресами памяти или непосредственными значениями.

```
AND o1, o2
```

A	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

Следующий пример вычисляет логическое И логической единицы и логического
нуля (1 AND 0).

```
mov al, 1    ;AL = one
mov bl, 0    ;BL = zero
and al, bl   ;AL = AL and BL = 0
```

Команда **and** производит поразрядное логическое умножение операндов и записывает результат на место первого операнда*:*

```
mov al, 1100b ;al=00001100b
```

Команда OR

Команда **OR** выполняет логическое сложение двух операндов — o1 и o2.

Результат сохраняется в операнде o1. Типы операндов такие же, как у команды **AND**.

A	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

Простой пример установки наименее значимого бита (первый справа) переменной mask в 1.

```
or mask,1
```

Команда **OR** производит поразрядное логическое сложение операндов и записывает результат на место первого операнда.

```
mov al, 1100b ;al=00001100b
or al, 1010b  ;al=00001110b
```

Команда NOT

Используется для инверсии отдельных битов единственного операнда, который может быть регистром или памятью. Соответственно команда может быть записана в трех различных форматах:

```
NOT r/m8
NOT r/ml 6
NOT r/m32
```

A	NOT a
0	1
1	0

Следующий пример демонстрирует различие между операциями NOT и NEG:

```
mov al,00000010b ;AL = 2
mov bl,al        ;BL = 2
not al           ;после этой операции мы получим
                 ;11111101b = 0xFD (-3)
neg bl           ;а после этой операции результат будет
                 ;другим: 11111110 = 0xFE (-2)
```

Команды сдвига

Эти команды перемещают содержимое ячейки **влево** или **вправо**. Одним из операндов этих команд является количество сдвигов cnt. Оно либо равно 1, либо определяется содержимым регистра **CL** (при этом **CL** сохраняет своё содержимое после операции).

Логические сдвиги

Логические сдвиги - команды сдвига, где участвуют все биты первого операнда, при этом бит, уходящий за пределы ячейки, заносится в флаг **CF**, а с другого конца в операнд добавляется ноль.

Логический сдвиг влево (shift left): SHL

Логический сдвиг вправо (shift right): SHR

```
mov al, 01000111b
shl al,1              ;CF=0, al=10001110b
mov al, 01000111b
shr al,1              ;CF=1, al=00100011b
mov bh, 0011100b
mov cl,3
shl bh,cl             ;CF=1, al=11000000b
```

Команды сдвига

Арифметические сдвиги

Арифметические сдвиги предназначены для реализации быстрого умножения и деления знаковых чисел на степени двойки.

Арифметический сдвиг влево (shift arithmetic left): SAL

Арифметический сдвиг вправо (shift arithmetic right): SAR

- сдвиг всех битов операнда вправо на один разряд, при этом выдвигаемый справа бит становится значением флага переноса **cf**;
- одновременно слева в операнд вдвигается не нулевой бит, а значение старшего бита операнда, то есть по мере сдвига вправо освобождающиеся места заполняются значением знакового разряда.
- указанные выше два действия повторяются количество раз, равное значению второго операнда.

```
mov ah,10001110b
sar ah,1                ;CF=0, al=11000111b
mov ah,10001110b
sal ah,1                ;CF=1, ah=00011100b
```

```
mov ax,88
sar ax,2 ;(ax) разделить на 2 в второй степени, то есть на 4
```

Команды сдвига

Циклические сдвиги

Особенность циклических сдвигов в том, что "уходящий" бит не теряется, а возвращается в операнд, но с другого конца.

Циклический сдвиг влево (shift arithmetic left): ROL

Циклический сдвиг вправо (shift arithmetic right): ROR

```
mov ah,11000011b
rol ah,1            ;CF=1, al=10000111b
mov ah,11100010b
ror ah,1            ;CF=0, al=01110001b
```