Тема 4

Основы языка Ассемблера

Пример ассемблерной программы (16-битный ассемблер)

```
1)            .model   small
2)            .stack   100h
3)            .data
4)  message   db       'Hello, world!',13,10,'$'
5)            .code
6)            .startup
7)            mov      dx, offset message
8)            mov      ah, 9
9)            int      21h
10)           .exit
11)           end
```

Встраиваемый ассемблерный код

Использование ассемблерных вставок в виде встраиваемого ассемблерного кода – 1-й способ связи Ассемблера и C++.

Синтаксис:

```
_asm  команда_ассемблера  [;комментарий]

_asm {
   команда_ассемблера [;комментарий]
   команда_ассемблера [//комментарий или
                      /*комментарий*/]
   }
```

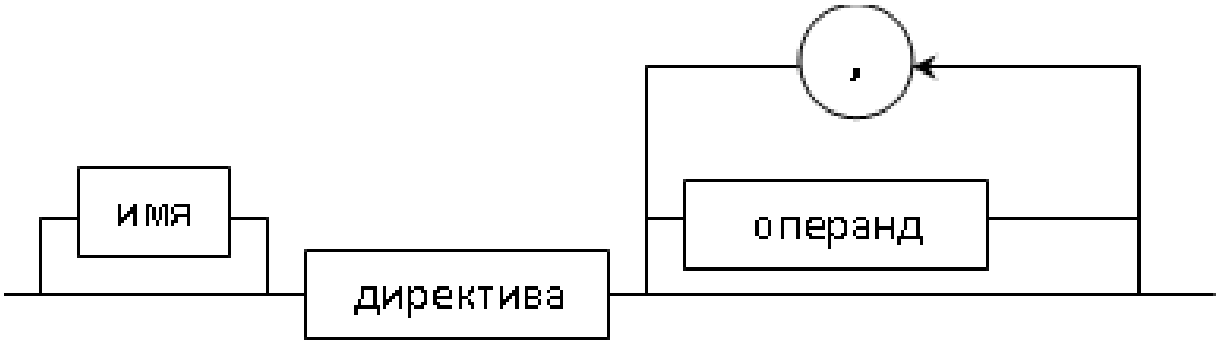
Пример ассемблерной вставки в программу на C++ (MS VS 2010)

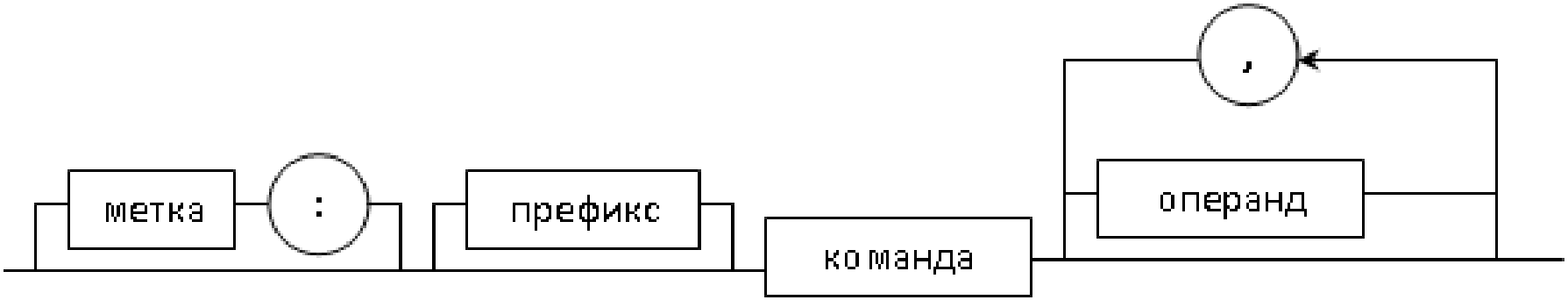
```
#include <iostream>
using namespace std;

void main() {
   int a, b, sum;
   cout << "Enter two integer numbers:\n";
   cin >> a >> b;

   _asm {
       mov          eax, a
       add          eax, b
       mov          sum, eax
   }

   cout << " sum is: " << sum << "\n";
}
```

Категории предложений Ассемблера

- **команды** (инструкции). Команда включает мнемонический код и операнды, разделённые запятыми (от 0 до 3);
- **директивы**, которые являются указанием компилятору на выполнение некоторых действий;
- **макрокоманды** – предложения языка Ассемблера, которые в процессе трансляции замещаются другими предложениями;
- **комментарии** – строки, начинающиеся с символа “;”, перед которым может стоять произвольное число пробелов. В ассемблерных вставках допускаются комментарии в стиле C++.

Синтаксическая диаграмма для директив и команд Ассемблера





Пояснения

Метка – символьный идентификатор, позволяющий обращаться к первому байту машинной команды, в которую будет преобразована соответствующая команда Ассемблера:

```
s1: mov ax, 10
```

Префикс – символическое обозначение элементов отдельных команд, предназначенных для изменения стандартного действия команды:

```
repne movsb
```

Операнды – части директивы или команды ассемблера, обозначающие объекты, с которыми будут выполняться какие-то действия (в команде) или уточняющие смысл директивы:

```
push dx
```

Ключевые слова Ассемблера

Ключевые слова - служебные слова Ассемблера, которые можно использовать только в строго определенном контексте. К ключевым словам относятся:

- имена команд Ассемблера и имена префиксов;
- имена регистров;
- имена операторов Ассемблера.

Идентификаторы и константы

Идентификаторы - конструкции, предназначенные для обозначения различных объектов в программе.

Константы служат для записи неизменяемых значений, чаще всего числовых.

`25, -3, 0`	целые числа в десятичной системе счисления
`101001b`	целое число в двоичной системе счисления
`35h, 0ffh`	целые числа в 16-ричной системе счисления
`'F'`	число, соответствующее коду указанного символа

Числовые выражения Ассемблера

Выражения позволяют записать цепочку действий с помощью обращений к одно- и двуместным *операторам* Ассемблера.

Простейшими операторами являются:

- арифметические операторы +, -, *, /, MOD;
- побитовые операторы AND, OR, XOR, NOT;
- операторы сравнения EQ, NE, GT, LT, LE, GE;
- операторы сдвига SHR, SHL.

Операторы вычисляются в момент компиляции, поэтому записать выражение eax+3 нельзя!

Адресные выражения Ассемблера

В качестве операндов-примитивов **адресных выражений** могут выступать:

- имена меток операторов Ассемблера;
- имена переменных;
- специальная константа $ - счетчик адреса. Ее значение равно смещению строки, в которой записан счетчик адреса, относительно соответствующего сегмента.

Адресные операторы Ассемблера

Допустимыми являются следующие **адресные операторы**:

- адрес + число, адрес – число *(результат имеет адресный тип)*;
- адрес – адрес *(результат имеет числовой тип и равен количеству байт в указанном промежутке) – не работает в ассемблерных вставках*;
- SEG адрес;
- OFFSET адрес *(оба оператора возвращают числовой тип)*.

Характеристики данных Ассемблера

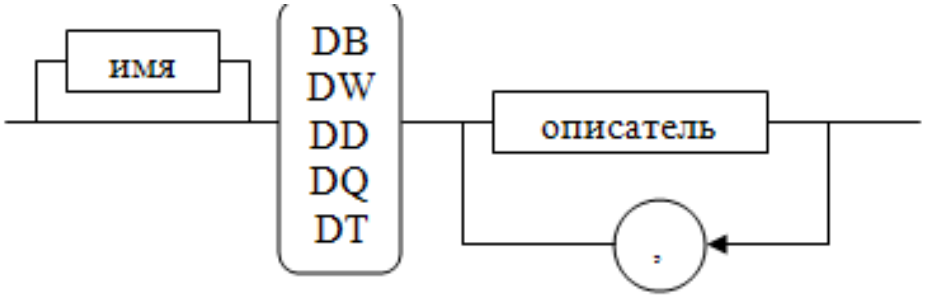
- **тип** данных;
- **длина** данных.

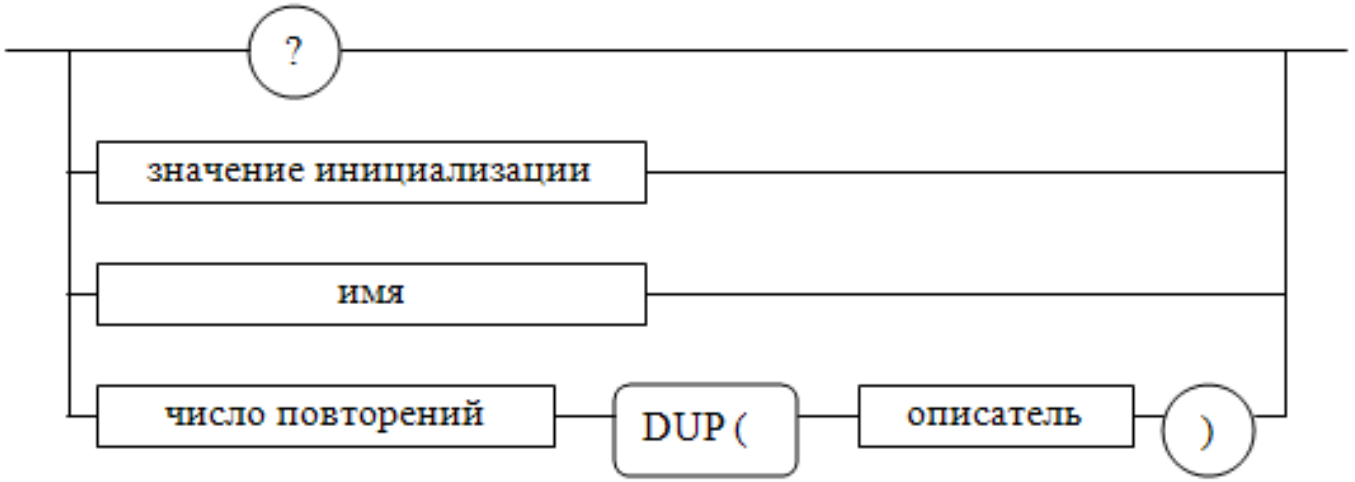
Основные типы данных Ассемблера:

- числовой тип;
- адресный тип.

Длина обрабатываемых данных задаётся в байтах (от 1 до 8).

Директивы описания данных






```
d1 dw 10 dup (5, 5 dup (?))
```

выделяет 60 слов, причем изначально заполняются лишь десять.

Описание данных и совместимость типов C++ и Ассемблера

Intel	Ассемблер	Кол-во байт	C	C++
byte	db	1	[unsinged] char	[unsinged] __int8
word	dw	2	[unsinged] short	[unsinged] __int16
double word	dd	4	[unsinged] int	[unsinged] __int32
quad word	dq	8	[unsinged] long long int	[unsinged] __int64

Операторы преобразования длины

Операторы преобразования длины позволяют явно указать или переопределить длину данных:

- `BYTE PTR выражение` `// 1 байт`
- `WORD PTR выражение` `// 2 байта`
- `DWORD PTR выражение` `// 4 байта`
- `QWORD PTR выражение` `// 8 байт`

Тип операнда может быть любым, тип результата – такой же, как и тип операнда.

Не применяется при регистровой адресации!

Операнды команд Ассемблера

Операнды могут быть:

- именами регистров, и в этом случае данные извлекаются из соответствующих регистров или записываются в них;
- числовыми константами, которые хранятся непосредственно в командах;
- числовыми выражениями, вычисляющимися во время компиляции;
- адресными выражениями.

Адресация

Адресация – информация о том, где находятся обрабатываемые командой данные. Возможно следующее расположение данных:

- в самой команде (*непосредственная адресация*);
- в регистрах (*регистровая адресация*);
- в памяти (*адресация в памяти*).

Как правило, в оперативной памяти располагается не более одного операнда!

Получение эффективного адреса

Эффективный адрес получается как сумма адресов, хранящихся:

- в самой команде;
- в регистрах **RBX/EBX/BX** или **RBP/EBP/BP**;
- в регистрах **RSI/ESI/SI** или **RDI/EDI/DI.**

(для защищенного режима могут использоваться любые регистры общего назначения).

Адрес сегмента при этом находится в одном из сегментных регистров.

Хотя бы одна часть адреса должна присутствовать!

Способы адресации данных

1) Непосредственная - операнд может быть представлен в виде числа, адреса, кода ASCII, а также иметь символьное обозначение:

```
mov     AX, 4C00h          ; Операнд - 16-ричное
                             число
mov     DX,offset mas  ;Смещение массива
                mas  заносится в DX
mov      DL, '!'           ;Операнд - код
                ASCII символа  '!'
```

2) Регистровая - операнд находится в регистре. Способ применим ко всем программно-адресуемым регистрам процессора:

```
mov    BP,SP      ;Пересылка содержимого
                SP в BP
```

Способы адресации данных

3) Адресация памяти

– ***прямая*** - в команде указывается символическое обозначение ячейки памяти, над содержимым которой требуется выполнить операцию :

```
mov  DL,mem1      ;Содержимое байта памяти
                  с символическим именем mem1
                  пересылается в DL
```

Если нужно обратиться к ячейке памяти с известным абсолютным адресом, то этот адрес можно непосредственно указать в качестве операнда:

```
; Настроим сегментный регистр ES на самое начало
  памяти (адрес 0)
    mov       AX, 0
    mov       ES, AX
```

Способы адресации данных

– ***косвенная*** – *е*сли хотя бы одна компонента адреса находится в регистре. При этом имя регистра задается в квадратных скобках.

До 80386 для этого можно было использовать только **BX, SI, DI и BP**, но потом эти ограничения были сняты и адрес операнда разрешили считывать также и из **EAX, EBX, ECX, EDX, ESI, EDI, EBP** и **ESP** (но не из **AX, CX, DX** или **SP** напрямую; надо использовать **EAX, ECX, EDX, ESP** соответственно или предварительно скопировать смещение в **BX, SI, DI** или **BP**).

Например:

```
mov ax,[bx]
```

Типы косвенной адресации

1) ***Базовый и индексный*** – адресация допустима только через регистры **EBX, EBP, ESI** и **EDI**. При использовании регистров **EBX** или **EBP** адресацию называют *базовой*, при использовании регистров **ESI** или **EDI** – *индексной*.

При адресации через регистры **EBX, ESI** или **EDI** в качестве сегментного регистра подразумевается **DS**; при адресации через **EBP** – регистр **SS**. Таким образом, косвенная адресация через регистр **EBP** предназначена для работы со стеком.

```
mov   AL, [EBX]  ;подразумевается DS:[EBX]
mov   AH, [EDI]
```

Типы косвенной адресации

2) ***Базовый и индексный со смещением*** – относительный адрес операнда определяется суммой содержимого регистра (**EBX, EBP, ESI** или **EDI**) и указанного в команде числа, которое называют *смещением (сдвигом):*

```
mov ax, [ebp]+2
```

или

```
mov ax, 2[ebp]
```

3) ***Базово-индексный (адресация по базе с индексированием)*** – относительный адрес операнда определяется как сумма содержимого пар регистров **EBX, EBP, ESI** или **EDI** *:*

```
mov ax, [ebp+esi]
```

Типы косвенной адресации

4) ***Базово-индексный со смещением*** –относительный адрес операнда определяется как сумма содержимого пар регистров **EBX, EBP, ESI** или **EDI** и *смещения:*

```
mov ax,[ebx+esi+2]
mov ax,[ebx][esi]+2
mov ax,[ebx+2][esi]
mov ax,[ebx][esi+2]
mov ax,2[ebx][esi]
```

Это все записи одного и того же:

В регистр AX помещается слово из ячейки памяти со смещением, равным сумме чисел, содержащихся в EBX и ESI, и числа 2.

Типы косвенной адресации

5) ***Косвенная адресация с масштабированием***:

Идентичен предыдущему, за исключением того, что с его помощью можно прочитать элемент массива слов, двойных слов или учетверенных слов:

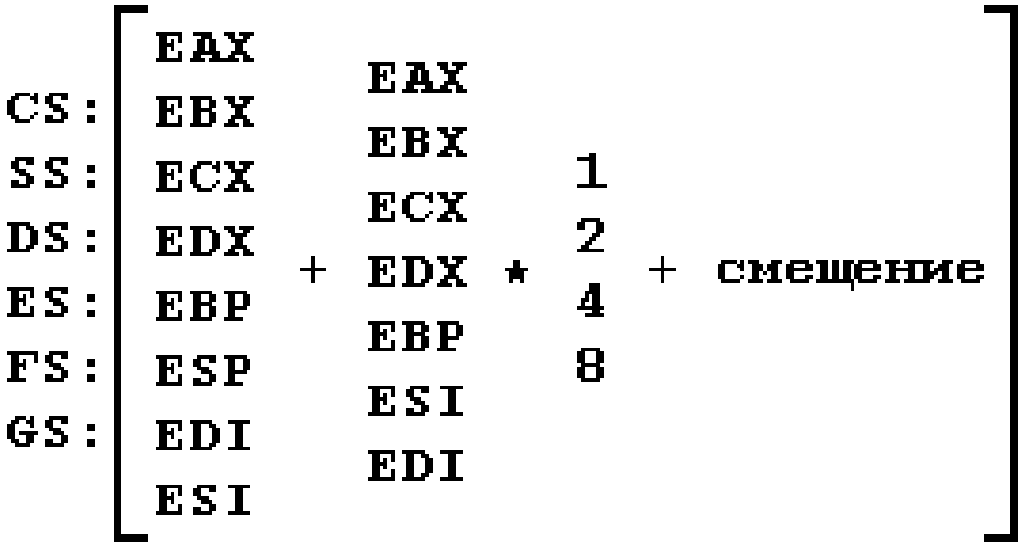
```
mov ax,[esi*2]+2
```

Множитель, который может быть равен 1, 2, 4 или 8, соответствует размеру элемента: байту, слову, двойному слову, учетверенному слову соответственно. Из регистров в этом варианте адресации можно использовать только **EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP**, но не **SI, DI, BP** или **SP**, которые можно было использовать в предыдущих вариантах.

Типы косвенной адресации

6) ***Адресация по базе с индексированием и масштабированием***

Это самая полная возможная схема адресации, в которую входят все случаи, рассмотренные ранее, как частные:



Команда MOV – команда пересылки данных

- **`MOV оп1, оп2`**

переносит содержимое второго операнда (оп2) в первый операнд (оп1). Содержимое второго операнда не меняется.

Ограничения, накладываемые командой MOV на комбинации операндов, следующие:

- длины операндов должны быть равны;
- первый операнд не может быть непосредственным;
- запрещена пересылка память – память;
- запрещена пересылка в регистры CS и EIP (этим должны заниматься команды передачи управления);
- пересылка в сегментные регистры возможна только из регистров общего назначения или из памяти.

Примеры команды MOV

```
mov   ax, bx      ;правильно – содержимое
                  ;регистра bx пересылается в ax
mov   eax, di     ;неправильно – не совпадают
                  ;длины операндов
mov   al, 100001b     ; правильно, длина
          ;определяется по первому операнду
mov   20, 30      ;неправильно – первый операнд
                  ;не может быть числом
mov   dx, offset message ; правильно, второй
     ;операнд  вычисляется при компиляции
mov   ds, es ; неправильно – запрещается
          ; пересылка из одного сегментного
          ; регистра в другой
```

Примеры команды MOV

```
mov  ax, [bx]    ; верно – в регистр ax
  заносится слово из памяти, эффективный
  адрес которой хранится в регистре bx
mov  ax, [bx][si]  ; верно – в регистр ax
  заносится слово из памяти, эффективный
  адрес которой  состоит из двух частей, одна
  хранится в регистре bx, другая – в регистре
  si
mov  ax, [bx+si]   ; эквивалентная запись
  предыдущей команды
mov  [bp], 20      ; неверно – не определена
  длина операндов
mov  byte ptr [bp], 20 ; верно – длина
     первого операнда задана явно
mov  [bp], byte ptr 20 ; можно и так…
```

Примеры команды MOV

Пусть в программе на C++ описаны переменные

```
int a, b, c, min;
```

Тогда

```
mov  eax, a  ; верно
mov  ax, a  ; неверно, несовпадение длин
  операндов
mov  ax, word ptr a ; верно, в ax будут
          ; помещены  младшие разряды a
mov  eax, a+4  ; верно, в eax будет значение b
mov  eax, a+2  ; верно, но бессмысленно…
```

Примеры команды MOV

Пусть идентификатор **message** имеет длину 1

```
mov  ax, word ptr message[bx] ; адрес второго
  операнда берется из команды и регистра
mov  ax, word ptr [bx+message]
mov  ax, word ptr [message+bx] ; эквивалентные
  записи предыдущей команды
mov  ax, message[bx] ; а так писать нельзя –
  несовпадение  длин!
mov  ax, 4[bx] ; верно – противоречия нет
mov  ah, message[bx][si] ; заданы все три части
  адреса
mov  ah, [message+bx+si] ; эквивалентная запись
  предыдущей команды
mov  ax, es:[bx]  ; верно – заменяется
  сегментный регистр
```

Команды пересылки данных

- **XCHG оп1, оп2**

меняет местами данные оп2 и оп1.

```
message db  'Hello, world!',13,10,'$'
…
XCHG BL, BH              ; верно
XCHG DH, message+3       ; тоже верно
XCHG BP, message         ; неверно – несовпадение
                         ; длин
XCHG byte ptr [bx], 2 ; неверно –
            ;использование непосредственного
            ; операнда запрещено
XCHG byte ptr [bx], [si] ; неверно – пересылка
                         ;память-память
```

Команды пересылки данных (продолжение)

- **LEA регистр, адрес**

загружает в регистр вычисленное во время выполнения адресное выражение.

Если выражение может быть вычислено во время компиляции, заменяется машинной командой **MOV**:

```
d2    dw    30
…
lea     bx, d2   ; эта команда эквивалентна
                    mov  bx, offset d2
mov   bx, d2     ; а эта команда пересылает
                 ; содержимое памяти по адресу!
```

Однако, например, команде

```
lea  bx, d2[bx]
```

нет эквивалентной записи команды **MOV.**

Команды пересылки данных (продолжение)

- **LDS регистр, дальний_адрес**

пересылает смещение из второго операнда в указанный регистр, а адрес сегмента – в регистр **DS**.

- **LES регистр, дальний_адрес**
- **LFS регистр, дальний_адрес**
- **LGS регистр, дальний_адрес**

выполняют аналогичные действия для регистров **ES, FS, GS.**

Команды пересылки данных (окончание)

- **PUSH операнд** заносит содержимое операнда в стек. Операнд должен иметь длину в 2 или 4 байта
- **POP операнд** извлекает данные из стека в операнд
- **PUSHF, POPF** занесение и извлечение регистра флагов
- **PUSHA, POPA** занесение и извлечение регистров **AX, CX, DX, BX, SP, BP, SI, DI.** Для регистра **SP** заносится старое значение
- **PUSHAD, POPAD** занесение и извлечение регистров **EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI**

```
push  si
push  bp
; участок программы, изменяющий регистры si и bp
pop   bp
pop   si  ; но не наоборот!
```

Команды пересылки данных (пример)

Задача: поменять местами содержимое двух восьмибайтных чисел

```
void main() {
   long long a=21545633390, b=-1;
       //  5 0438 466E    FFFF FFFF FFFF FFFF

   cout << a << " " << b << "\n"; // 21545633390 -1
   _asm {
       mov          eax, dword ptr a
       xchg         eax, dword ptr b
       mov          dword ptr a, eax
       mov          eax, dword ptr a+4
       xchg         eax, dword ptr b+4
       mov          dword ptr a+4, eax
   }
   cout << a << " " << b << "\n"; // -1 21545633390
   }
```

Команды сложения и вычитания

- **`ADD оп1, оп2`**

складывает оп2 и оп1. Результат помещается в оп1.

- **`ADC оп1, оп2`**

складывает оп2 , оп1, флаг CF. Результат помещается в оп1.

- **`SUB оп1, оп2`**

вычисляет оп1 – оп2. Результат помещается в оп1.

- **`SBB оп1, оп2`**

вычисляет оп1 – (оп2+CF). Результат помещается в оп1.

Вычитание реализовано через сложение!

Установка флагов при сложении

При сложении целых чисел не учитывается то, знаковые они или беззнаковые. Вместо этого выставляются флаги **CF, OF, ZF, SF**:

- **CF=1**, если произошёл перенос из старшего разряда (*переполнение беззнаковых чисел*);
- **OF=1,** если результат сложения чисел одного знака имеет противоположный знак *(переполнение знаковых чисел*);
- **ZF=1,** если все биты результата равны нулю;
- **SF=1,** если старший бит результата равен единице (*результат можно трактовать как отрицательное число*)

Команды сложения и вычитания (продолжение)

- **`INC оп1`** увеличивает оп1 на единицу. Флаг CF не изменяется.
- **`DEC оп1`** уменьшает оп1 на единицу. Флаг CF не изменяется.
- **`CMP оп1, оп2`** вычисляет оп1 – оп2. Результат не сохраняется, лишь выставляются флаги.
- **`NEG оп1`** изменяет знак оп1.

Если операнд содержит максимальное по модулю отрицательное число (например, -128, если операнд – байт), то выполнить эту команду нельзя – соответствующего положительного числа не существует. В этом случае значение операнда не меняется, и устанавливается флаг OF.

Команды сложения и вычитания (пример 1)

Задача: сложить два восьмибайтных числа

```
int main() {
  long long a=21545633390, b=-1, rezt;
  //  5 0438 466E     FFFF FFFF FFFF FFFF
  _asm {
      mov          eax, dword ptr a
      mov          ebx, dword ptr a+4
      add          eax, dword ptr b
      adc          ebx, dword ptr b+4
      mov          dword ptr rezt, eax
      mov          dword ptr rezt+4, ebx
  }
  cout << rezt << "\n"; // 21545633389 или 5 0438 466D
  return 0;
}
```

Команды преобразования чисел со знаком

- **CBW**

  преобразует байт, хранящийся в регистре AL, в слово, помещающееся в регистр AX. Знак не меняется!

- **CWD**

  преобразует слово, хранящееся в регистре AX, в двойное слово, помещающееся в регистры DX и AX.

- **CWDE**

  преобразует слово, хранящееся в регистре AX, в двойное слово, помещающееся в регистр EAX.

- **CDQ**

  преобразует двойное слово, хранящееся в регистре EAX, в учетверенное слово, помещающееся в регистры EAX и EDX.

Команды преобразования чисел со знаком

Пример:

Пусть у нас есть описание

```
a   dw   ?
b   db   ?
c   dw   ?
```

и нам надо посчитать значение выражения **`c = a + b`**.

Это делается так:

числа со знаком	*числа без знака*

`mov bx, a`
`mov al, b`
`cbw`
`add bx, ax`
`mov c, bx`
	`mov bx, a`
	`mov ah, 0`
	`mov al, b`
	`add bx, ax`
	`mov c, bx`

Команды сложения и вычитания (пример 2)

Задача: сложить два числа, имеющих разную длину

```
int main() {
   int a=2633390, c;
   short b=-1;

   _asm {
       mov          ax, b
       cwde
       add          eax, a
       mov          c, eax
   }
   cout << c << "\n";
   return 0;
}
```

Команды изменения флагов

- **STC** – установить флаг CF
- **CLC** – сбросить флаг CF
- **STD** – установить флаг DF
- **CLD** – сбросить флаг DF
- **STI** – установить флаг IF
- **CLI** – сбросить флаг IF