

Лабораторная работа 1

Ассемблерные вставки в Visual
studio

Базовые типы данных C(++)

Тип	Размерность
char	8 бит
short	16 бит
int	32 бита
long	64 бита

Строго говоря, размерность типов может отличаться от платформы к платформе. На практике их размерность соответствует указанной в таблице.

По умолчанию все типы являются знаковыми. Для того, чтобы явно указать знаковый/беззнаковый тип, можно использовать ключевые слова `signed/unsigned`, например: «`unsigned int`».

Тип `char` является исключением, т.к. стандарт C/C++ точно не определяет, является ли он по умолчанию знаковым или беззнаковым. Рекомендуется явно указывать `signed/unsigned`.

Шаблон для лабораторной работы (Visual Studio)

```
#include <stdio.h> // import printf

int main() {
    // Some example code; may vary
    char a = 'a';
    int b;

    __asm {
        // assembler text here;
    }
    printf ( "%d\n",  a);
    return 0;
}
```

Ассемблерные вставки Visual C++

- Можно
 - Обращаться к регистрам
 - Обращаться к меткам и переменным C++
 - Обращаться к параметру функции по имени
 - Использовать операторы PTR, LENGTH, SIZE, TYPE
 - Загружать адрес командой LEA
- Нельзя
 - определять данные директивами
 - использовать операторы кроме разрешённых выше
 - определять адрес директивой OFFSET
 - использовать макроопределения
 - обращаться к сегментам по имени

Ввод/вывод в Visual studio

Делаем с помощью функционала C/C++ :)

Можете использовать `#include <cstdio>`
(`printf`, `scanf` и т.д.),
можете использовать `#include <iostream>`
(`std::cin`, `std::cout`), не принципиально.

Гугл в помощь. Уроков валом, разберётесь, я в вас верю. Будут вопросы – помогу.

Отладчик

Он есть. Лучше им пользоваться.

Шаблон для лабораторной работы (SASM)

Здесь всё проще, SASM при создании нового файла вставляет шаблон по умолчанию. Но писать на Си не получится.

```
section .text

global main
main:
    ; assembler text here

    xor eax, eax
    ret
```

Объявление переменных в NASM (упрощённо)

<имя>: <размер> <данные>

Размер – одна из псевдоинструкций:

Псевдоинстр.	Размерность
db	8 бит
dw	16 бит
dd	32 бита
dq	64 бита

Задание строки:

msg: db 'Hello there!', 10, 0

Задание числа:

var: dd 42

Лучше объявлять
переменные в сегменте .data
(далее будет пример)

Немного документации можно найти здесь:

https://www.opennet.ru/docs/RUS/nasm/nasm_ru3.html

Полезные макросы

В SASM для ввода-вывода существует набор макросов. Их описание можно найти в разделе «помощь».

Для их использования нужно в начале файла добавить строку:

```
%include "io.inc"
```

В хэлпе если что написано :)

Пример с переменными и макросами в SASM

```
%include "io.inc"
```

```
section .data
```

```
msg: db 'Hello there!', 10, 0
```

```
var: dd 42
```

```
section .text
```

```
global main
```

```
main:
```

```
    PRINT_STRING msg
```

```
    PRINT_DEC 4, var
```

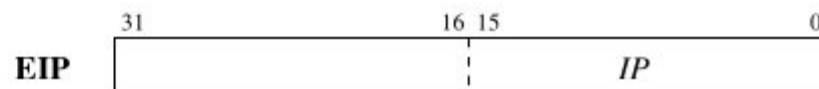
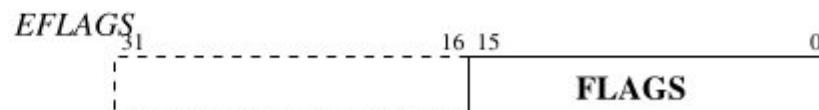
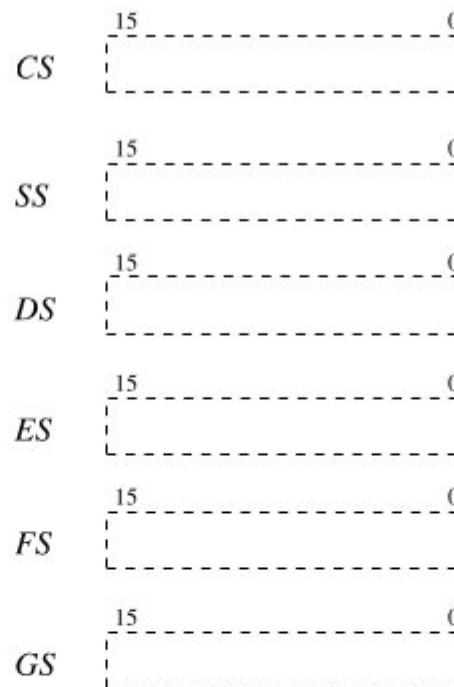
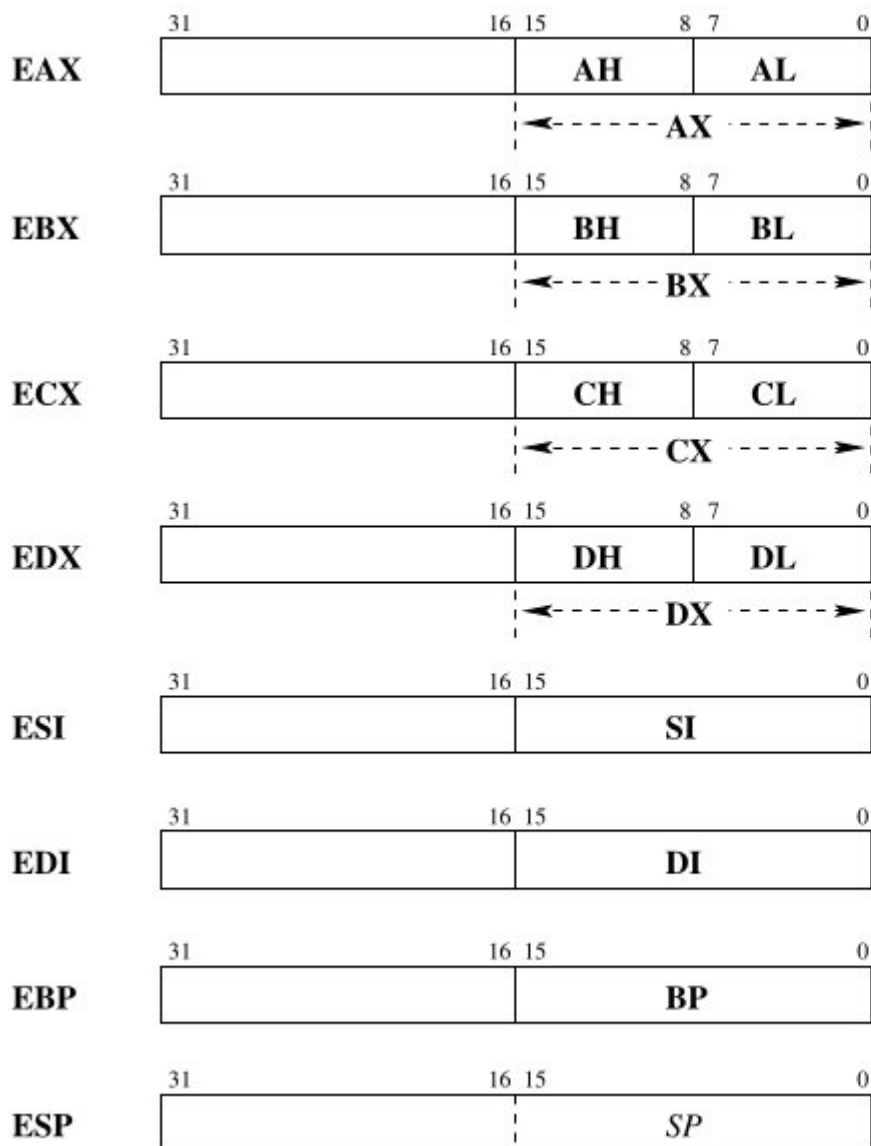
```
    xor eax, eax
```

```
    ret
```

Отладчик

В SASM он тоже есть :)

Регистры x86-32



Инструкция mov

mov <приёмник>, <источник>
; или <куда> <что> записать

Примеры:

```
unsigned char mem = 42;
```

```
mov al, 49          ; al = 49
mov al, 00110011b   ; al = 00110011b
mov al, 42h         ;
mov ax, bx          ; ax = bx
mov al, mem          ; al = mem
mov mem, 12          ; mem = 12
```

ОПЕРАЦИЯ mov mem1, mem2 НЕДОСТУПНА

Эффективное присваивание 0

Вместо того, чтобы использовать инструкцию

```
mov eax, 0
```

МОЖНО ИСПОЛЬЗОВАТЬ

```
xor eax, eax
```

что работает несколько быстрее, чем `mov`.

Встречается очень часто, поэтому лучше знать, чтобы потом не удивляться.

Обмен значений двух операндов

xchg <операнд 1>, <операнд 2>

Пример:

xchg ax, bx ; tmp=bx, bx=ax, ax=tmp

Сложение и вычитание

add ax, bx ; ax = ax + bx
add ax, mem ; ax = ax + mem
add mem, ax ; mem = mem + ax
add ax, 42 ; ax = ax + 42
add mem, 42 ; mem = mem + 42

sub ax, bx ; ax = ax - bx
sub ax, mem ; ax = ax - mem
sub mem, ax ; mem = mem - ax
sub ax, 42 ; ax = ax - 42
sub mem, 42 ; mem = mem - 42

ОПЕРАЦИЯ add/sub mem1, mem2 НЕВОЗМОЖНА

Умножение

`mul <множитель>` ; для беззнаковых чисел
`imul <множитель>` ; для знаковых чисел

Умножать на число нельзя :(
Только на регистр/переменную.

Неявный множитель	Операнд	Произведение
AL	Reg8/Mem8	AX
AX	Reg16/Mem16	DX:AX
EAX	Reg32/Mem32	EDX:EAX

Пример беззнакового умножения

```
// Example: 2 * 10  
char n = 10;
```

```
mov al, 2  
mul n      ; AX=2*10=20=0014h: AH=00h AL=14h
```

```
// Example: 26 * 10  
n = 10;
```

```
mov al, 26  
mul n      ; AX=26*10=260=0104h: AH=01h AL=04h
```

** В отладчике значения регистров в шестнадцатеричном виде*

Пример умножения со знаком

```
// Example: 8 * -1  
mov  ax,8  
mov  bx,-1  
imul bx
```

$\text{DX:AX} = -8 = \text{ffffffff8h} = 0014\text{h}$: $\text{DX} = \text{ffffh}$ $\text{AX} = \text{fff8h}$

Эффективное умножение

При умножении на степень числа 2 более эффективным является сдвиг влево на требуемое число битов. Логика аналогична умножению на 10 в десятичной системе счисления, где справа дописывается 0. Сдвиг вправо позволит эффективно делить на степень числа 2.

Сдвиг более чем на 1 требует загрузки величины сдвига в регистр CL (в старых системах).

Умножение на 2:

```
shl ax,1
```

Умножение на 8:

```
mov ax,3  
shl ax,cl
```

Деление

`div <делитель>` ; для беззнаковых чисел
`idiv <делитель>` ; для знаковых чисел

Делить на число нельзя :(
Только на регистр/переменную.

Делимое	Операнд	Частное	Остаток
AX	Reg8/Mem8	AL	AH
DX:AX	Reg16/Mem16	AX	DX
EDX:EAX	Reg32/Mem32	EAX	EDX

Пример беззнакового деления

```
mov ax,100  
mov bh,2  
div bh      ; 100 div 2 = 50: ah=0 al=50
```

- todo: пример деления где остаток не 0

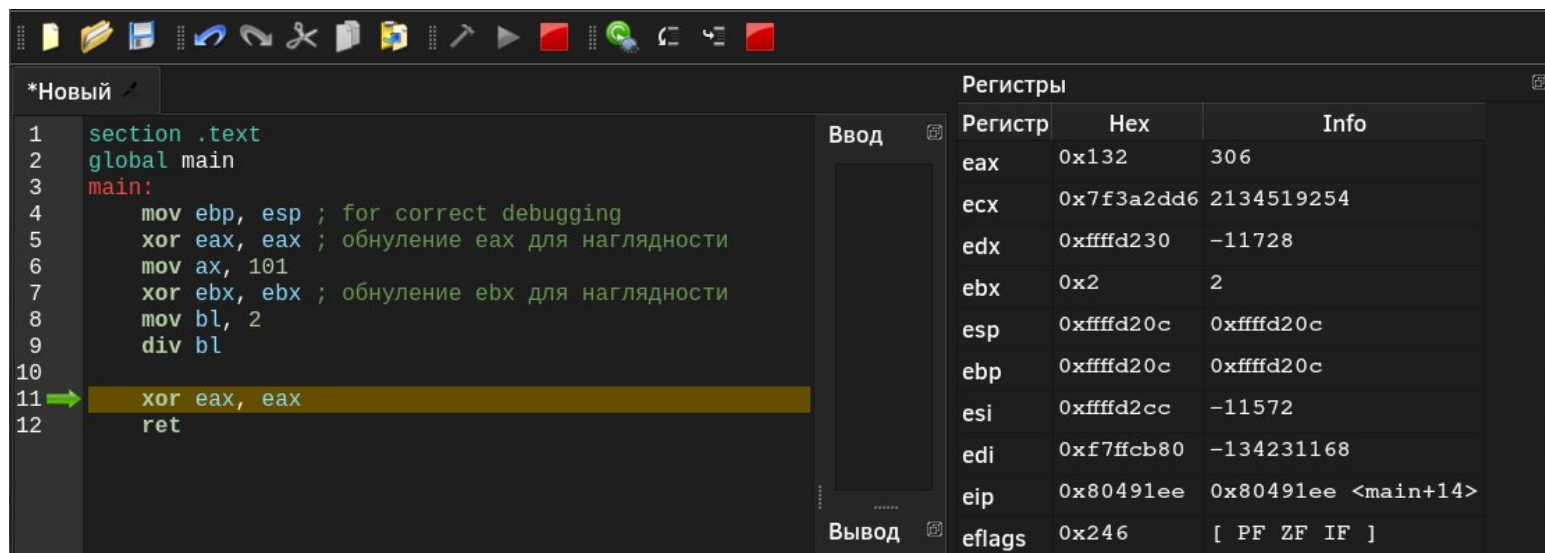
Пример деления с ненулевым остатком

```
xor eax, eax ; обнуление eax для наглядности
mov ax, 101
xor ebx, ebx ; обнуление ebx для наглядности
mov bl, 2
div bl
```

В отладчике можно будет увидеть, что

eax = 0x132

Где второй октет 0x1 – остаток в регистре AH, октет 0x32 – частное в регистре AL.



The screenshot shows a debugger window with the following components:

- Code Window:** Displays assembly code for a new file (*Новый). The code is as follows:

```
1 section .text
2 global main
3 main:
4     mov ebp, esp ; for correct debugging
5     xor eax, eax ; обнуление eax для наглядности
6     mov ax, 101
7     xor ebx, ebx ; обнуление ebx для наглядности
8     mov bl, 2
9     div bl
10
11 → xor eax, eax
12     ret
```

Line 11 is highlighted with a green arrow.
- Registers Window:** Shows the current state of the CPU registers.

Регистр	Hex	Info
eax	0x132	306
ecx	0x7f3a2dd6	2134519254
edx	0xffffd230	-11728
ebx	0x2	2
esp	0xffffd20c	0xffffd20c
ebp	0xffffd20c	0xffffd20c
esi	0xffffd2cc	-11572
edi	0xf7fcb80	-134231168
eip	0x80491ee	0x80491ee <main+14>
eflags	0x246	[PF ZF IF]

Используя команды DIV и особенно IDIV, очень просто вызвать переполнение.

Прерывания приводят (по крайней мере в системе, используемой при тестировании этих программ) к непредсказуемым результатам.

В операциях деления предполагается, что частное значительно меньше, чем делимое.

Деление на ноль всегда вызывает прерывание.

Но деление на 1 генерирует частное, которое равно делимому, что может также легко вызвать прерывание.

Рекомендуется использовать следующее правило:

если делитель - байт, то его значение должно быть меньше, чем левый байт (AH) делителя:

если делитель - слово, то его значение должно быть меньше, чем левое слово (DX) делителя.

Преобразование знака

Инструкция NEG обеспечивает преобразование знака двоичных чисел из положительного в отрицательное и наоборот.

Практически инструкция NEG устанавливает противоположные значения битов и прибавляет 1.

Примеры:

```
neg ax
```

```
neg bl
```

```
neg BINAMT ; байт или слово в памяти
```

Пример

Вычислить значение выражения

$$y = \frac{12 + 3}{8 + 6} * 3 + 12$$

```
mov ax,12
```

```
add ax,3 ; ax = 12 + 3
```

```
mov bl,8
```

```
add bl,6 ; bl = 8 + 6
```

```
div bl ; делим содержимое ax на содержимое bl
```

```
mov ah,0 ; остаток обнуляем
```

```
mov bl,3
```

```
mul bl ; результат в al умножаем на 3
```

```
add ax,12 ; к произведению прибавляем 12
```

```
mov y,ax ; заносим в y
```