

# Низкоуровневое программирование

## Лекция 1

Введение. Архитектура x86-64.  
Основы языка ассемблера.

# О предмете

В рамках предмета рассматриваются различные аспекты выполнения программ центральным процессором.

Структура курса:

1. Основы языка ассемблера. Выполнение программ процессором.
2. Соглашения о вызовах. Правила вызова функций в разных операционных системах.
3. Компиляция и компоновка. Структура исполняемых файлов.
4. Низкоуровневые аспекты безопасности. Какие уязвимости эксплуатирует вредоносное ПО.
5. Функционирование современных компьютерных систем. Устройство современных процессоров, организация памяти, прерывания и исключения, виртуализация. Роль операционных систем в управлении ПК.

# Литература

## **Общая архитектура компьютера:**

Таненбаум Э., Остин Т. Архитектура компьютера.

## **Язык ассемблера:**

Аблязов Р. З. Программирование на ассемблере на платформе x86-64.

Куссвюрм Д. Профессиональное программирование на ассемблере x64 с расширениями AVX, AVX2 и AVX-512

## **Дизассемблирование, вопросы безопасности:**

Касперски К. Искусство дизассемблирования.

Климентьев К.Е. Компьютерные вирусы и антивирусы. Взгляд программиста.

# Полезные ссылки

## [SASM IDE](#)

[Complier Explorer \(online компиляция кода с выводом ассемблерного листинга\)](#)

[Документация по NASM](#)

[Сайт со списком инструкций с их описаниями \(на английском\)](#)

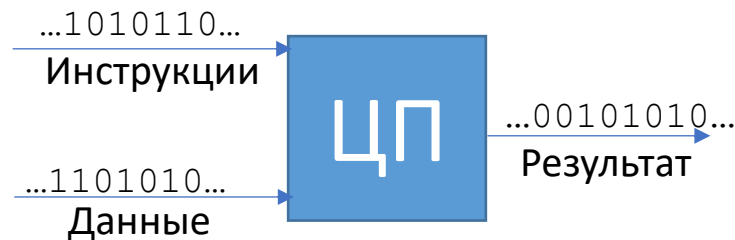
[Сайт со списком инструкций x86 и их описаниями \(на русском, устарел\)](#)

[IDA Free \(дизассемблер\)](#)

# Процессоры

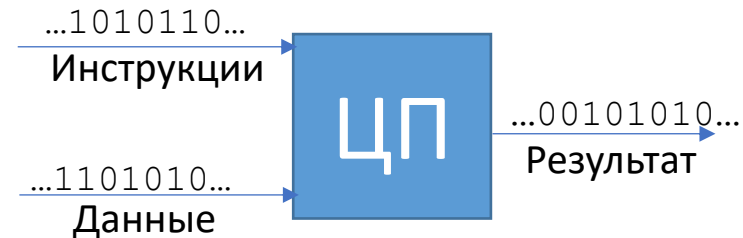
**Центральный процессор** – устройство, предназначенное для выполнения основных действий по обработке информации и управления работой других устройств вычислительной машины.

- ЦП является цифровым устройством => и обрабатываемые данные, и управляющие инструкции кодируются в виде двоичных<sup>1</sup> последовательностей.
- ЦП является синхронным устройством => ЦП работает под управлением специального тактового генератора, задающего ритм его работы. **Тактовая частота** (количество тактов в секунду) является одной из характеристик ЦП и в первом приближении описывает его производительность.
- **Разрядность ЦП** – размер в битах типового элемента данных, которого ЦП может обработать за 1 инструкцию. В рамках курса будут рассматриваться ЦП разрядности 32 и 64 бита.



<sup>1</sup> см. ЭВМ «Сетунь»

# Машинные инструкции



**Машинная инструкция** – двоичная последовательность, однозначным образом определяющая выполняемое процессором действие.

- Инструкция состоит из **опкода (кода операции)**, определяющего выполняемое действие, и списка **операндов**, над которыми действие выполняется.

`add` `eax`, `1`

`10000011111000``000``00000001`

`sub` `ecx`, `2`

`1000001111101``001``00000010`

- Набор всех возможных инструкций задает **машинный язык (машинный код)**.
- Инструкции могут кодироваться разным количеством байт (в рассматриваемых ЦП – от 1 до 15).
- Выполнение инструкции проходит через 4 этапа:
  - чтение;
  - декодирование (определение конкретного действия);
  - выполнение;
  - запись результата (при наличии).

# Оперативная память

**Оперативное запоминающее устройство** (ОЗУ, оперативная память, RAM) – запоминающее устройство, непосредственно связанное с центральным процессором и предназначенное для хранения данных, участвующих в выполнении арифметико-логических операций [[ГОСТ 25492-82](#)].

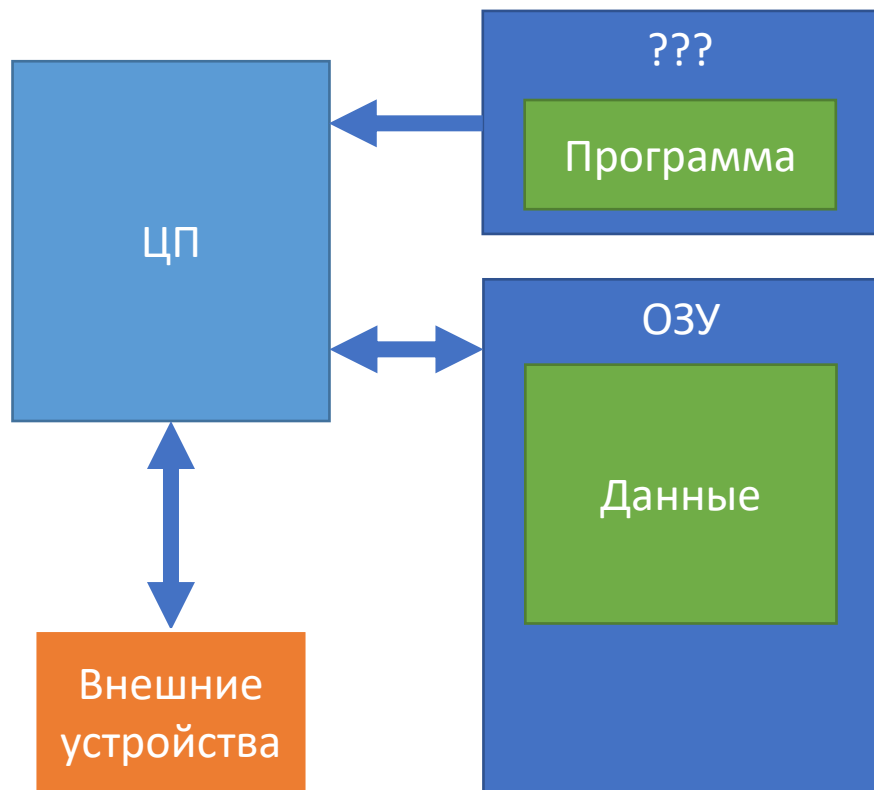
В настоящее время в роли ОЗУ выступает энергозависимая<sup>1</sup> память с произвольным доступом<sup>2</sup>.

Оперативная память делится на ячейки равного размера - **байты**. Каждый байт имеет свой порядковый номер - **адрес**.

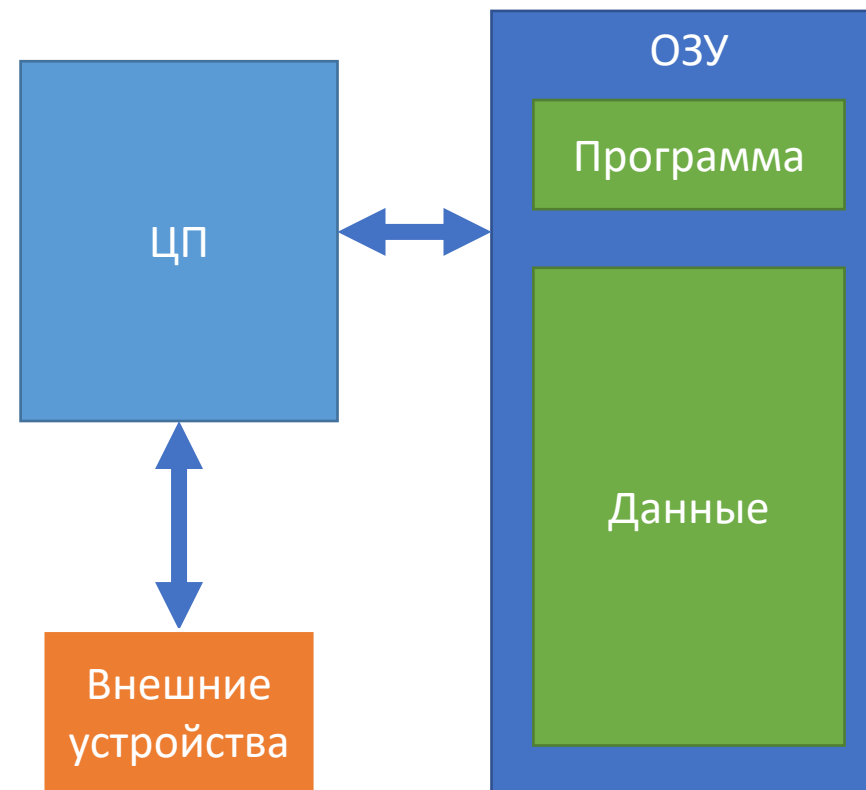
**Байт** – минимальная адресуемая единица информации.

В современных ЭВМ распределением оперативной памяти между программами занимается операционная система. Каждая программа работает в своем собственном *виртуальном* адресном пространстве, и не видит данных других программ.

# Архитектуры ЭВМ



Гарвардская архитектура



**Архитектура фон Неймана**  
(Принстонская архитектура)



# Архитектура набора команд

С каждым ЦП связаны архитектура набора команд и микроархитектура.

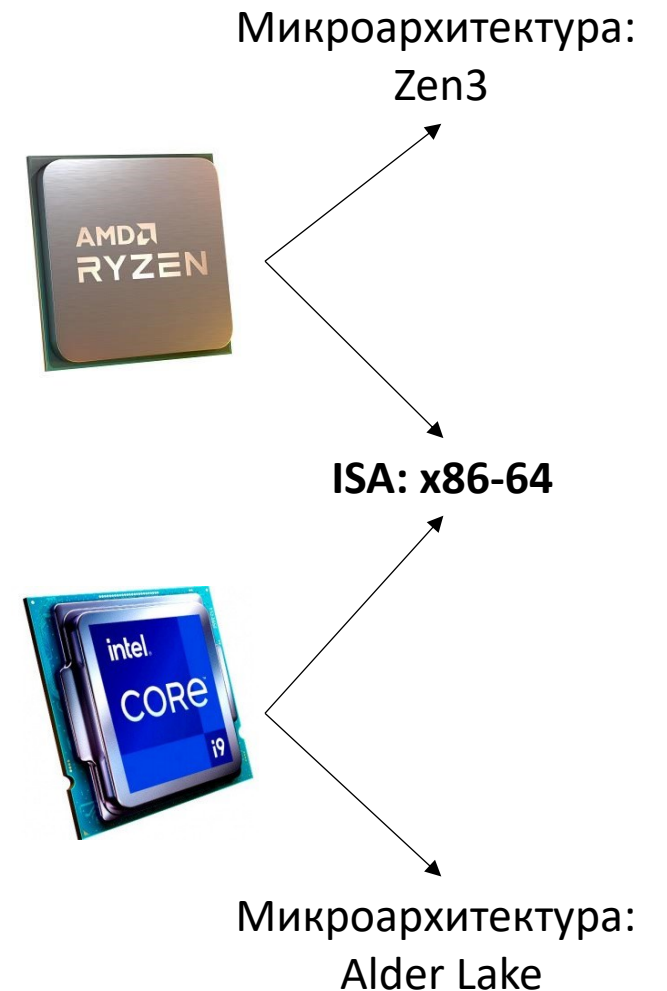
**Архитектура набора команд** (instruction set architecture, ISA) задает:

- машинный язык (регистры ЦП, возможные инструкции + их кодировка);
- модель программирования ЦП (правила, по которым осуществляется выполнение программ).

*Программа, скомпилированная под заданную ISA может выполняться на любом ЦП, реализующем данную ISA.*

**Микроархитектура** процессора описывает устройство процессора на аппаратном уровне.

В рамках курса будут рассматриваться только ISA x86-64 и, поверхностно, ARM.



# Семейство архитектур x86

Семейство ISA x86 берет начало в 1978 г. с появлением ЦП Intel 8086.

В настоящий момент наиболее распространенной в сегменте ПК является архитектура x86-64.

Краеугольным камнем всего семейства x86 является высокая обратная совместимость – ЦП архитектуры x86-64 может выполнять программы, собранные под x86-16 и x86-32.

Архитектуры семейства x86 содержат большое количество машинных инструкций<sup>1</sup>, многие из которых являются вариациями друг друга.

1978 г – архитектура **x86-16**

- [разрядность](#) - 16 бит;
- размер адреса – 20 бит (макс. 1 МБ ОЗУ);
- [FPU](#) – отдельное устройство;
- представитель: Intel 8086 (тактовая частота 5 МГц).

1985 г – архитектура **x86** (официальное название **IA-32**)

- разрядность - 32 бита;
- размер адреса – 32 бита (макс. 4ГБ ОЗУ);
- добавлены механизмы защиты памяти;
- FPU интегрирован в кристалл процессора (начиная с i486);
- представитель: Intel 80386 (тактовая частота - 16 МГц);

2003 г – архитектура **x86-64** (официальное название **amd64**)

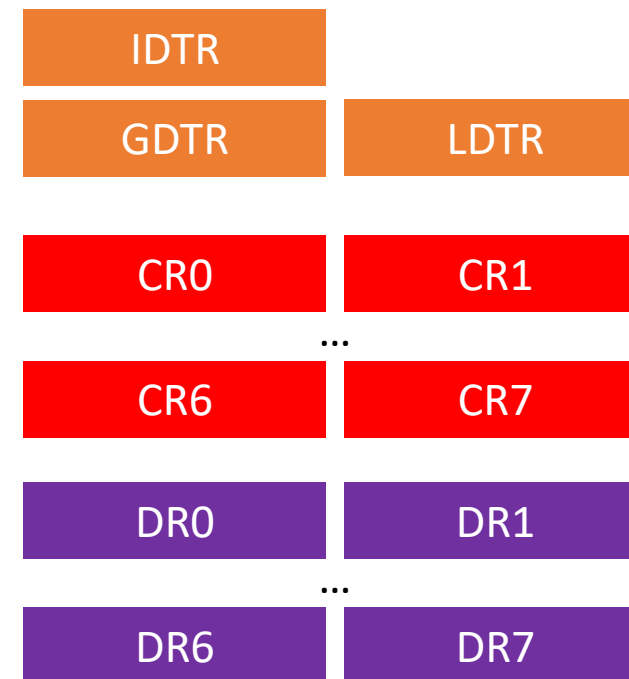
- разрядность – 64 бита
- размер адреса – 64 бит (макс. 16 ЭБ ОЗУ);
- увеличенное число регистров общего назначения;
- новая модель работы с памятью;
- поддержка векторных операций;
- представитель: AMD Opteron (тактовая частота – 1,4 ГГц).

[see also](#)

<sup>1</sup> если не учитывать вариации – то примерно 1500 инструкций, если учитывать – около 6000

# Регистры x86-64

**Регистр** – ячейка памяти фиксированного размера, расположенная внутри процессора



■ регистры общего назначения  
■ индексные регистры  
■ стековые регистры

■ программный счетчик и регистр флагов  
■ сегментные регистры  
■ векторные регистры

■ табличные регистры  
■ управляющие регистры  
■ отладочные регистры

# Части регистров общего назначения

В x86-16 регистры были 16-битными (AX, BX, ..) и состояли из 2 половин (AH|AL, BH|BL, ...).

В x86 регистры были увеличены и переименованы с добавлением префикса E (EAX).

В x86-64 регистры были вновь переименованы с заменой префикса на R (RAX).

***RAX, EAX, AX, AH и AL – это имена частей одного и того же регистра.***

Запись в 4-байтовую часть регистра **обнуляет** старшие 4 байта.

Запись 1-/2-байтовую часть регистра **не меняет** остальную часть регистра.

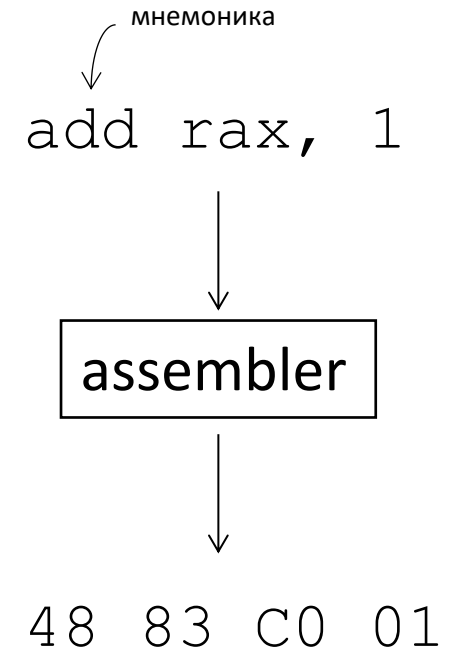


# Язык ассемблера

**Язык ассемблера** – язык программирования, представляющий собой символьную форму записи машинного языка [ГОСТ 19781-90].

**Ассемблер** – программа, осуществляющая преобразование (трансляцию) программы на языке ассемблера в программу на машинном языке.

- Инструкции процессора в языке ассемблера записываются в виде коротких **мнемоник**, за которыми следует список операндов.
- Инструкции на языке ассемблера и на машинном языке взаимно однозначны => возможно *дизассемблирование*.
- Помимо инструкций, язык ассемблера содержит вспомогательные конструкции для описания данных и структуры создаваемого исполняемого файла.
- Поскольку можно задать разные формы записи машинного языка, существуют разные ассемблеры с их собственными языками.



# Язык ассемблера NASM

**NASM** – кроссплатформенный ассемблер (есть версии для Windows и Linux).

Особенности языка ассемблера NASM:

- простота – базовое описание можно уместить на несколько страниц (без перечисления всех инструкций процессора).
- регистронезависимость (ADD = add = AdD), исключение – имена меток и секций (см. далее);
- для написания инструкций используется синтаксис Intel;
- отсутствие типов данных (в привычном смысле).
- отсутствие проверок корректности действий программиста ☺;
- есть макросы (%define, %assign, %macro и пр);

# Типы данных и ассемблер

**В языке ассемблера отсутствует понятие типа данных в привычном смысле.**

**То, как интерпретируются данные, зависит от инструкции. Помнить, что лежит в регистре/памяти в данный момент и использовать корректные инструкции – задача программиста.**

При работе с памятью в некоторых случаях требуется указать размер считываемых/записываемых данных. В NASM определены следующие типовые размеры<sup>1</sup>:

Размер	Подходящий тип C/C++	Размер	Спецификатор размера
byte	char	1	db/resb
word	short	2	dw/resw
dword	int/float	4	dd/resd
qword	long long/double	8	dq/resq
tword	long double	10	dt/rest

<sup>1</sup> список неполный

# Структура программы

В языке ассемблера необходимо определить не только программу, но и структуру исполняемого файла в целом.

Всякий исполняемый файл содержит в себе несколько областей (сегментов/секций), хранящих код и данные программы и, в некоторых случаях, служебную информацию. Каждый сегмент имеет свое место загрузки и свои разрешения на доступ к нему.

При написании программы на языке ассемблера основные сегменты программы приходится определять самостоятельно.

Основные сегменты:

**.data** – сегмент глобальных/статических переменных с заданным значением

**.rodata** – сегмент констант

**.bss** - сегмент глобальных/статических переменных без заданного значения (инициализируются нулем).

**.text** – сегмент кода.

```
section .data
    a: db 5
    b: dq 0xFF
    array: times 16 db 0
```

```
section .rodata
    const: db 7
```

```
section .bss
    c: resq 1
```

```
section .text
    global main
```

```
main:
    xor rax, rax
    ret
```



# Метки

**Метка** – символьная строка, представляющая некоторый адрес в памяти.

Метки обычно используются для обозначения переменных, функций или мест внутри функций, соответствующих точкам перехода.

Синтаксис объявления метки:

`<имя метки>:`

При ассемблировании метка заменяется на соответствующий адрес<sup>1</sup>.

Если имя метки начинается с точки, метка является **локальной**.

Локальные метки обычно используются в функциях для организации циклов и условных переходов.

Полное имя локальной метки:

`<имя предыдущей обычной метки>.<имя метки>`

<sup>1</sup>или эквивалентное ему смещение (дистанцию от инструкции до цели, на которую указывает метка).

```
section .data
    a: db 5
    b: dq 0xFF
    array: times 16 db 0
```

```
section .rodata
    const: db 7
```

```
section .bss
    c: resq 1
```

```
section .text
    global main
```

```
main:
    xor rax, rax
    .ret:
    ret
```

# Структура программы

```
char a=7;
long long b = 255;
short array[4] {1,2,3,4};
int array2[4] {1,1,1,1};
```

```
const char constant = 7;
const char cstring[] = "assembler";
```

```
long long c[3];
```

```
int main(){
    c[0]=a+b;
    return 0;
}
```

```
section .data
    a: db 7
    b: dq 0xFF
    array: dw 1,2,3,4
    array2: times 4 dd 1
```

```
section .rodata
    constant: db 7
    cstring: db "assembler",0
```

```
section .bss
    c: resq 3
```

```
section .text
    global main
```

```
main:
    movsx rax, byte[a]
    add rax, [b]
    mov [c], rax
    xor rax, rax
    ret
```

# Синтаксис Intel

В нотации Intel инструкции языка ассемблера имеют форму

<мнемоника> (приемник/операнд1), (операнд2), (операнд3)

Если приемник или источник указаны в [ ], то соответствующее значение интерпретируется как *адрес в оперативной памяти*. По этому адресу происходит чтение/запись.

Примеры:

<code>nop</code>	(Нет операции)
<code>neg rcx</code>	$RCX = -RCX$
<code>add rax, rbx</code>	$RAX += RBX$
<code>add rax, [rbx]</code>	$RAX += *(qword*)RBX$
<code>vaddps xmm0, xmm1, xmm2</code>	$XMM0 = XMM1 + XMM2$

# Перемещение данных

**mov <приемник>, <источник>**

- Размер приемника и источника должен быть равен (`mov rax, eax` – нельзя).
- Если один операнд – адрес в памяти, а другой операнд – регистр, то размер перемещаемых данных равен размеру регистра.
- Если размер передаваемых данных нельзя определить неявно, то его нужно указать.

<code>mov ah, al</code>	AH = AL
<code>mov eax, 0x8065</code>	EAX = 0x8065
<code>mov eax, [0x8065]</code>	EAX = *(DWORD*)0x8065
<code>mov [VAR], rax</code>	*(QWORD*)VAR = RAX
<code>mov dword[rcx], 5</code>	*(DWORD*)RCX = 5

# Адресация

Операнд, заключенный в [], является адресным выражением.

Адресное выражение может состоять из 3 частей: **базы, индекса и смещения**.

Индекс может умножаться на 1/2/4/8.

При вычислении адреса **база**, **индекс** и **смещение** складываются (вычитать индекс нельзя, но можно использовать отрицательные числа в регистрах).

В адресном выражении могут быть указаны максимум 2 регистра.

Метки в адресных выражениях эквивалентны константам.

```
mov al, [LABEL + rcx]
```

```
mov al, [rax + rcx]
```

```
mov al, [LABEL + rcx + 1]
```

```
mov bx, [rsi + 2*rax]
```

```
mov ecx, [LABEL + 4*rbx - 1]
```

```
mov rdx, [LABEL + 8*rcx + rbx]
```

# Инструкция LEA (пример)

Инструкция `lea` (**L**oad **E**ffective **A**ddress) выполняет вычисление адреса (без чтения/записи).

Инструкция также может использоваться для вычисления простых математических выражений.

*Данная инструкция не изменяет состояние регистра `FLAGS` (см. далее).*

```
section .data
    array: dd 12,24,36,48

section .text
global main

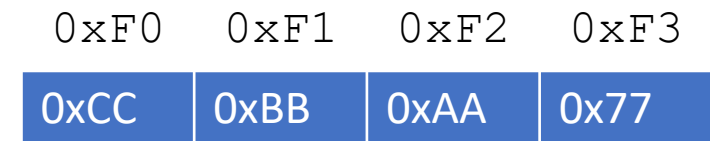
main:
    lea rdi, [array]
    lea rsi, [array+4*rcx+4]
    lea rbx, [rsi-4]

    lea rcx, [rcx+5]
    lea rcx, [2*rdx]
    lea rax, [8*rcx+rdx+5]
    lea rdx, [8*rdx+rdx]
```

# Представление чисел в памяти (пример)

В архитектурах семейства x86 числа хранятся в Little Endian кодировке (т.н. обратный порядок байтов). Младший байт числа будет располагаться по младшему адресу.

```
mov dword[0xF0], 0x77AABBCC
```



```
mov ax, [0xF0]; AX=??
```

**Д/З:** BigEndian и LittleEndian  
(см. «Архитектура компьютера», с.97)

# Дополнительный код

В большинстве современных архитектур отрицательные числа представляются в дополнительном коде (при этом старшие биты отрицательного числа = 1).

Диапазон значений unsigned int  $[0, 2^{32} - 1]$

Диапазон значений signed int  $[-2^{31}, 2^{31} - 1]$

*Дополнительный код позволяет выполнять знаковое и беззнаковое сложение/вычитание одинаковым образом.*

	uint	int
0xFFFFFFFF	$2^{32} - 1$	-1
0xFFFFFFFFD	$2^{32} - 2$	-2
...		
0x80000000	$2^{31}$	$-2^{31}$
0x7FFFFFFF	$2^{31} - 1$	$2^{31} - 1$
...		
0x00000002	2	2
0x00000001	1	1
0x00000000	0	0



# Простые инструкции арифметики (пример)

<b>Сложение</b> add	<b>Вычитание</b> sub (SUBstract)	<b>Изменение знака</b> neg (NEGate)
add ax, 10	sub ax, 10	neg rax
add ebx, ebx	sub ebx, ebx	neg word[rax]
add dx, [rsi]	sub dx, [rsi]	
add [rdi], cl	sub [rdi], cl	

# Битовые операции

<b>И</b> and	<b>Или</b> or	<b>Исключающее или</b> xor	<b>Не</b> not
and ax, 10	or ax, 10	xor ax, 10	not rax
and ebx, ebx	or ebx, ebx	xor ebx, ebx	not qword[rax]
and dx, [rsi]	or dx, [rsi]	xor dx, [rsi]	
and [rdi], cl	or [rdi], cl	xor [rdi], cl	

xor eax, eax – обнуление RAX/EAX

# СДВИГИ

Сдвиг влево/вправо shr, shl (SHift to Right/Left )	Арифметический сдвиг sar, sal [=shl] (Shift Arithmetic to Right/Left )	Циклический сдвиг ror, rol (ROtate Right/Left)
shr rax, 5  shl cx, 5  shl edx, <u>cl</u>	sar rax, 5  sar cx, 5  sar edx, <b>cl</b>	ror rax, 5  rol cx, 5  ror edx, <b>cl</b>
AL = -2 = 0xFE=11111110 shr al, 3 AL = 31 = 0x1F=00011111	AL = -2 = 0xFE=11111110 sar al, 3 AL = -1 = 0xFF=11111111	AL = 0xFE=11111110 ror al, 3 AL = 0xDF=11011111

# Умножение и деление (пример)

`mul <множитель>      div <делитель>`  
`imul <множитель>      idiv <делитель>`

`Mul /div` – беззнаковые операции, `imul/idiv` - знаковые операции.

Инструкции принимают 1 аргумент – множитель/делитель.

Инструкции неявно используют регистры `RAX` и `RDX` (или их меньшие части)\*.

<code>mul rbx</code>	<code>{RDX:RAX} = RAX*RBX</code>	<code>div bx</code>	<code>AX = {DX:AX} / BX</code> <code>DX = {DX:AX} % BX</code>
<code>imul ebx</code>	<code>{EDX:EAX} = EAX*EBX</code>	<code>idiv bl</code>	<code>AL = AX / BL</code> <code>AH = AX % BL</code>
<code>imul bl</code>	<code>AX=AL*BL</code>		

**Д/З:** инструкции `cwd`, `cdq`, `cqo` (пригодятся на л/р)

\* исключение – 1-байтовое умножение и деление, которые используют только регистр `AX`

# Преобразование чисел

**movsx** <приемник>, <источник>

**movzx** <приемник>, <источник>

Инструкции предназначены для корректного расширения числа в представление *большой* разрядности.

- **movsx** – расширение числа с учетом знака.
- **movzx** – расширение числа без учета знака.
- если операнд – адресное выражение, то указание размера обязательно.

<code>movzx ax, al</code>	<code>AX = (unsigned short)AL</code>
<code>movsx eax, byte[rax]</code>	<code>EAX = (int)(*(char*)RAX)</code>
<code>movzx qword[rbx], eax</code>	<code>*RBX = (unsigned long long)EAX</code>
<code>movsx qword[rbx], eax</code>	<code>*RBX = (long long)EAX</code>

# Преобразование чисел

<code>a:dd -2</code>  <code>mov EAX,[a]</code>	Было: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0]  Стало: RAX = 0x00000000FFFFFFFF [RAX>0, EAX<0]	<code>a: dd -2</code>  <code>movsx RAX,dword[a]</code>	Б: RAX = 0x0000000000000000 [RAX=0, EAX=0, AX=0, AL=0]  C: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0, AX<0, AL<0]
<code>a:dw -2</code>  <code>mov AX,[a]</code>	Было: RAX = 0x7777777777777777 [RAX>0, EAX>0, AX>0]  Стало: RAX = 0x7777777777777FFE [RAX>0, EAX>0, AX<0]	<code>a:dw -2</code>  <code>movsx RAX,word[a]</code>	Б: RAX = 0x7777777777777777 [RAX>0, EAX>0, AX>0, AL>0]  C: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0, AX<0, AL<0]
<code>a:db 10</code>  <code>mov AL,[A]</code>	Было: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0, AX<0, AL<0]  Стало: RAX = 0xFFFFFFFFFFFF0A [RAX<0, EAX<0, AX<0, AL>0]	<code>a:db 0x10</code>  <code>movzx RAX,byte[a]</code>	Б: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0, AX<0, AL<0]  C: RAX = 0x00000000000000FA [RAX>0, EAX>0, AX>0, AL>0]

# Выполнение программы. Регистр RIP

По умолчанию инструкции считываются из памяти и выполняются последовательно.

Специальный регистр **RIP** указывает на *следующую* инструкцию, которая будет выполнена. Значение RIP автоматически увеличивается после чтения инструкции.

Изменить значение RIP (и тем самым изменить порядок выполнения программы) можно только специальными инструкциями (call, ret, инструкции условного и безусловного перехода.)

выполняется

```
main:
    movsx rax, byte[a]
RIP → add rax, [b]
      mov [c], rax
      xor rax, rax
      ret
```

# Безусловный переход

**jmp <точка назначения>**

Инструкция `jmp` меняет значение регистра RIP на значение аргумента.

Аргумент может быть меткой или адресным выражением.

```
add rax, 8
jmp label
sub rax, 4

label:
RIP → ror eax, cl
```

выполняется



# Регистр FLAGS

Регистр RFLAGS (FLAGS для краткости) содержит **слово состояния программы**. Большинство битов слова состояния указывают на свойства результата последней операции (т.н. **флаги**). Некоторые биты являются управляющими.

Для сохранения регистра флагов на стек используются инструкции `pushf` и `popf`.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

**CF** – флаг переноса.

**SF** – флаг знака.

**PF** – флаг четности.

**OF** – флаг переполнения.

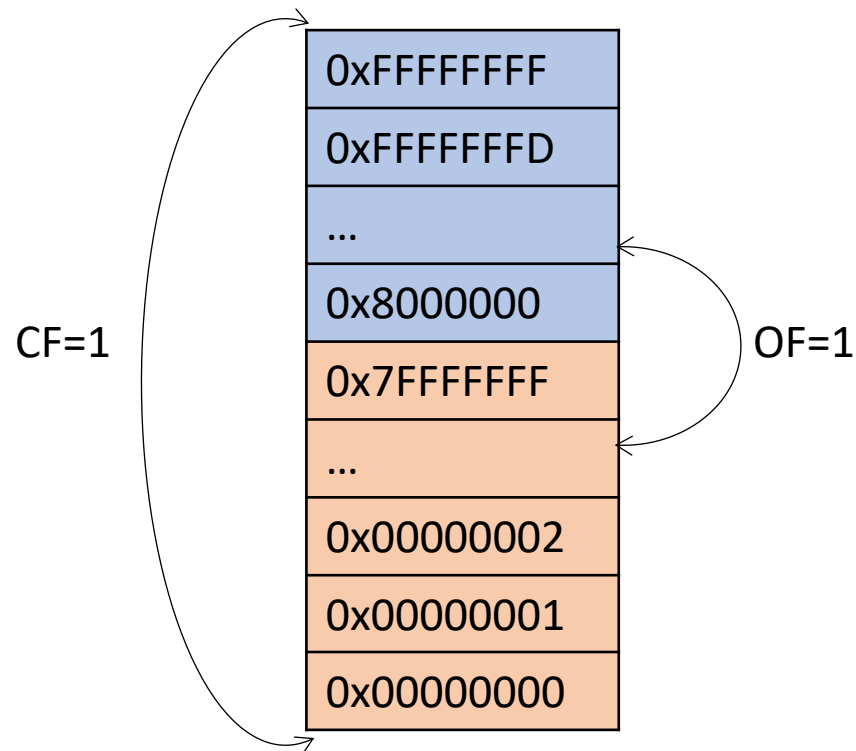
**ZF** – флаг нуля.

# Флаги CF и OF

Флаг **CF** (Carry Flag) равен 1, если в ходе операции произошло *беззнаковое* переполнение.

Флаг **OF** (Overflow Flag) равен 1, если в ходе операции произошло *знаковое* переполнение.

Д/З: инструкции длинного сложения `adc`, `sbb`  
инструкции сдвига с переносом `rcr`, `rcl`



# Инструкции сравнения

`cmp <операнд1>, <операнд2 >`  
`test <операнд1>, <операнд2>`

Инструкции сравнения не изменяют первый операнд, они только меняют регистр FLAGS.

Инструкция `cmp` сравнивает аргументы через *вычитание* с последующим выставлением флагов SF, CF, OF и ZF.

Инструкция `test` сравнивает аргументы через *побитовое И* с последующим выставлением флагов SF, ZF.

<code>mov rax, 10; cmp rax, 10;</code>	SF=0, ZF=1;
<code>mov rax, 10; cmp rax, 11;</code>	SF=1, CF=1, ZF=0;
<code>mov rax, 0; test rax, rax;</code>	ZF=1
<code>mov rax, 10; test rax, rax;</code>	ZF=0

# Операции условного перехода (пример)

Инструкции условного перехода изменяют значение регистра RIP на значение аргумента только биты регистра FLAGS удовлетворяют определенному условию.

Инструкции условного перехода имеют форму `j*`, где \* - символы, задающие условие.

Отрицание условия задается суффиксом `n`:

`je -> jne, jg -> jng.`

Допускается комбинировать условия:

`j(a or e) = jae, j(l or e) = jle`

Инструкция	Значения флагов
<code>je(equal)</code> <code>jz(zero)</code>	ZF=1
<code>jg(greater)</code>	SF=0, ZF=0F
<code>jl(less)</code>	SF=1, ZF!=0F
<code>ja(above, unsigned greater)</code>	CF=0, ZF=0
<code>jb(below, unsigned less)</code> <code>jc(carry)</code>	CF=1
<code>js(sign, less than zero)</code>	SF=1
<code>jo(overflow)</code>	OF=1

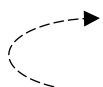
# Циклы (пример)

Циклы в ассемблере организуются через комбинацию [метка + условный переход] (предпочтительно) или через инструкцию `loop` (лучше избегать).

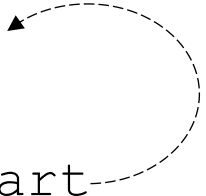
Инструкция `loop` принимает адрес (метку) начала цикла, как аргумент. При исполнении сначала выполняется декремент **ECX**, а потом проверяется его значение. Если `ECX != 0` - происходит прыжок на указанную метку.

```
long long x = 0;
int i = 5;
while(--i > 0)
    x+=10;
```

```
        mov rax, 0
        mov ecx, 5
.cycle_start:
    → add rax, 10
    loop cycle_start
```



```
        mov rax, 0
        mov ecx, 5
.cycle_start:
    add rax, 10
    sub ecx, 1
    jnz cycle_start
```



# Стек вызовов

**Стек вызовов** (программный стек или просто стек) – область памяти, предназначенная для хранения локальных переменных и вспомогательных данных, необходимых для осуществления вызовов функций.

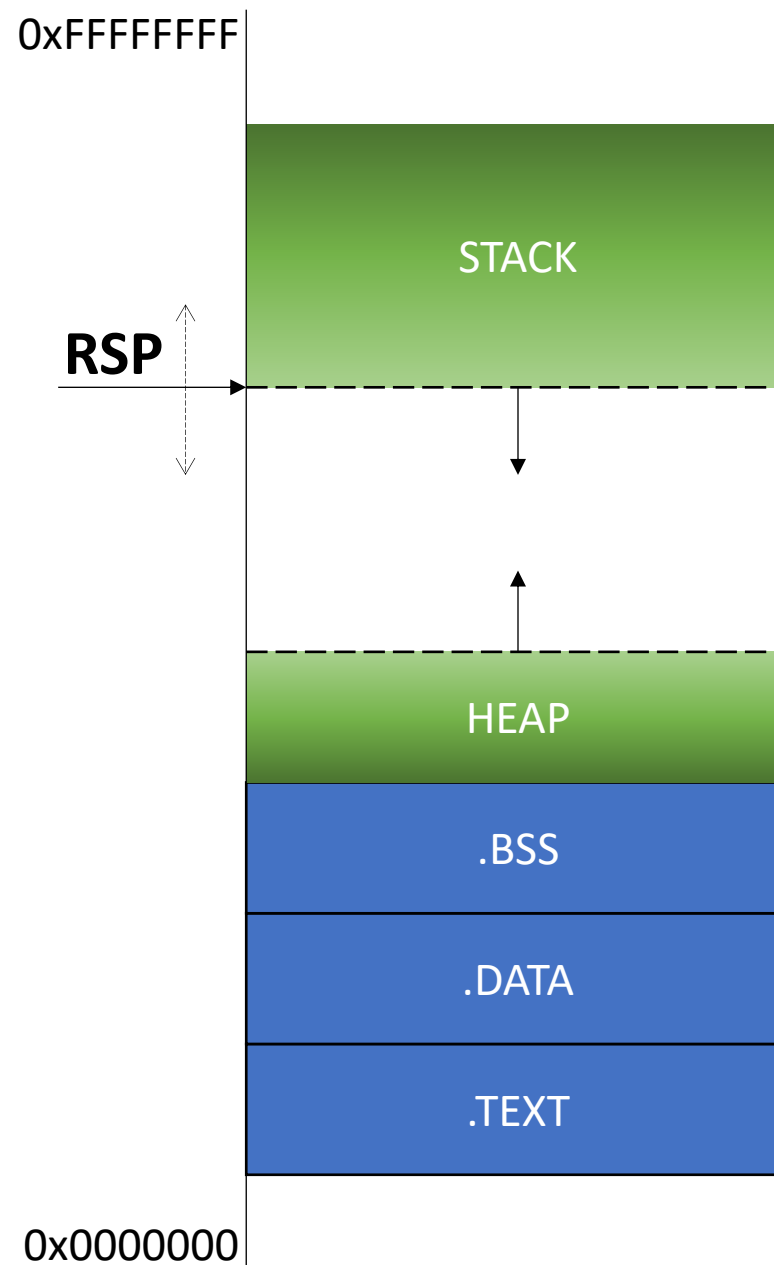
Указатель на вершину стека хранится в регистре **RSP**.

***Стек растет вниз.***

*Вычитание из RSP увеличивает стек.*

*Прибавление к RSP уменьшает стек.*

Поскольку расположение вершины стека непостоянно, меток в стеке быть не может => это задача программиста – помнить, что он разместил `int i` по адресу `[RSP-4]` 😊



# Push и pop

Регистр RSP можно указывать в адресных выражениях для доступа к стеку.

Кроме того, есть специальные инструкции.

Инструкция **push** вычитает из RSP размер операнда и записывает значение на вершину стека.

Инструкция **pop** читает значение из вершины стека и прибавляет к RSP размер операнда.

В x86 **push** и **pop** поддерживают операнды размером **2 или 4** байта\*.

В x86-64 **push** и **pop** поддерживают операнды размером **2 или 8** байт\*.

*\*Примечание: у mov нет такого ограничения, mov [rsp], eax – допустимо*

push ax	⇔	sub rsp, 2 mov [rsp], ax
push qword[rbx]	⇔	sub rsp, 8 mov qword[rsp], [rbx]
push rax	⇔	sub rsp, 8 mov [rsp], rax
pop rbx	⇔	mov rbx, [rsp] add rsp, 8
pop qword[rcx]	⇔	mov qword[rcx], [rsp] add rsp, 8
pop bx	⇔	mov bx, [rsp] add rsp, 2

# Функции (пример)

Функции в ассемблере обозначаются метками.

Для вызова функции используется инструкция `call`. Инструкция сохраняет на стеке текущее значение регистра RIP, и записывает в него же значение аргумента.

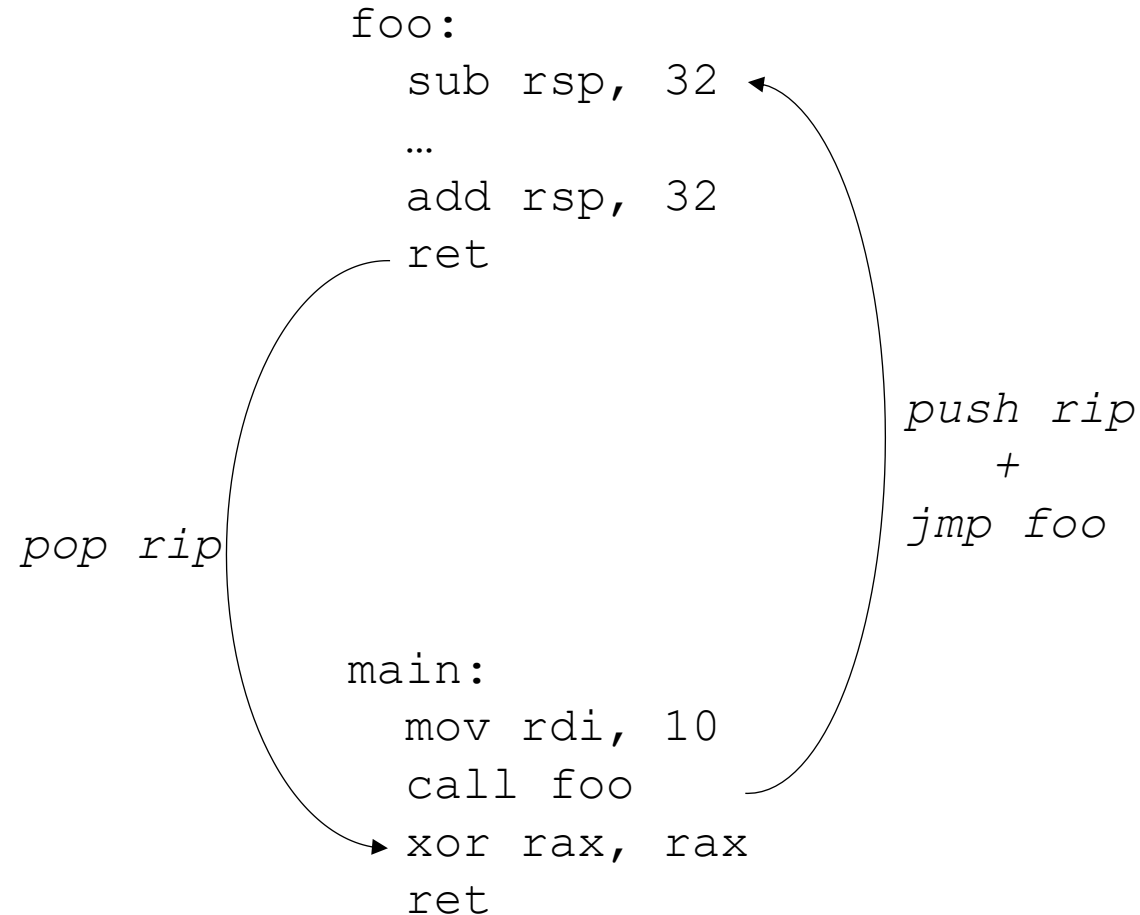
Для возврата из функции в место вызова используется инструкция `ret`. Инструкция считывает со стека адрес возврата в регистр RIP и удаляет этот адрес со стека.

Т.к. адрес возврата считывается со стека,

1. регистр RSP должен иметь то же значение, что и при входе в функцию – если на стеке есть локальные переменные, они должны быть удалены со стека;
2. адрес возврата не должен быть перезаписан;

*Помните, что `main()` – тоже функция.*

*Подробнее – см. лекцию 3.*





# Локальные переменные

Локальные переменные располагаются в стеке вызовов. Локальные переменные в языке ассемблера безымянны – доступ к ним осуществляется через регистр RSP<sup>1</sup>.

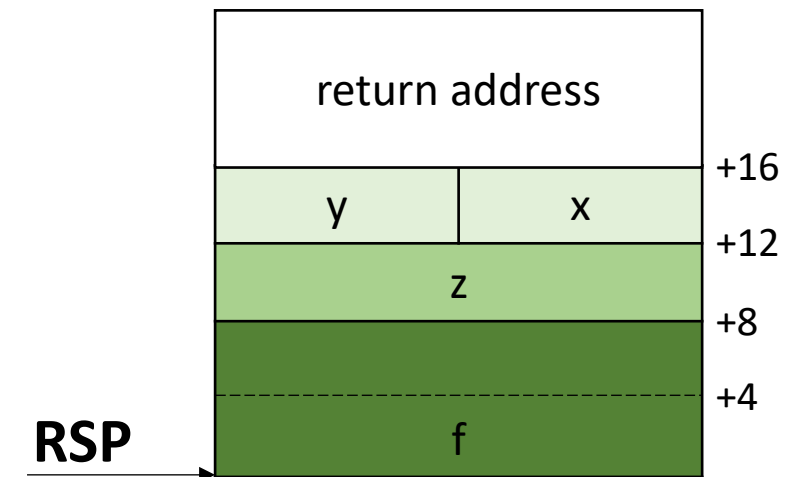
Выделение места под локальные переменные обычно выполняется в виде `SUB RSP, N`, где N – суммарный размер локальных переменных<sup>2</sup>.

*Итоговое расположение переменных в выделенном блоке выбирает программист.*

Для удаления локальных переменных со стека используется обратная инструкция `ADD RSP, N`.

```
int main(){
    short x = 6311;
    short y = 6312;
    int z = 6313;
    long long f = -1;
    /*... */
}
```

```
main:
    sub rsp, 24
    mov QWORD[rsp], -1
    mov DWORD[rsp+8], 6313
    mov WORD [rsp+12], 6312
    mov WORD [rsp+16], 6311
    /*... */
```



<sup>1</sup> или RBP – подробнее см. лекцию 3

<sup>2</sup> данный размер обычно округляют до кратного 8 или 4 в зависимости от разрядности ЦП.