

Низкоуровневое программирование

Лекция 1

Введение. Архитектура x86-64.
Основы языка ассемблера.

О предмете

Предмет посвящен рассмотрению низкоуровневых аспектов, которые в дальнейшем потребуются для освоения курсов «Системное программирование», «Операционные системы», «Безопасность операционных систем».

Структура курса:

1. Основы языка ассемблера. Выполнение программ процессором.
2. Компиляция и компоновка. Чем занимаются компилятор и компоновщик.
3. Функционирования современных компьютерных систем. Организация памяти, прерывания и исключения, виртуализация.
4. Низкоуровневые аспекты безопасности. Какие уязвимости эксплуатирует вредоносное ПО.

Литература

Общая архитектура компьютера:

Таненбаум Э., Остин Т. Архитектура компьютера.

Язык ассемблера:

Юров В.И. Ассемблер: учебник для вузов [устаревшее]

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

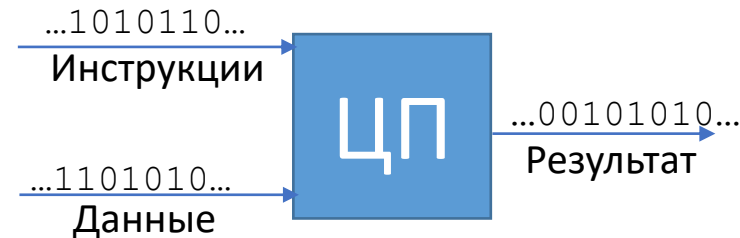
[Документация по NASM](#)

[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.

Машинная инструкция – битовая строка, однозначным образом определяющая выполняемое процессором действие.

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

Набор всех возможных инструкций задает **машинный язык**.

Каждая инструкция выполняется за 1 или несколько **тактов** – дискретных промежутков времени между импульсами внутреннего генератора синхросигнала, используемого для синхронизации работы элементов ЦП. **Тактовая частота** (количество тактов в секунду) является одной из характеристик ЦП.

add eax, 1

10000011111000000000000001

sub ecx, 2

1000001111110100100000010

Оперативная память

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

В настоящее время в роли ОЗУ выступает энергозависимая¹ память с произвольным доступом².

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

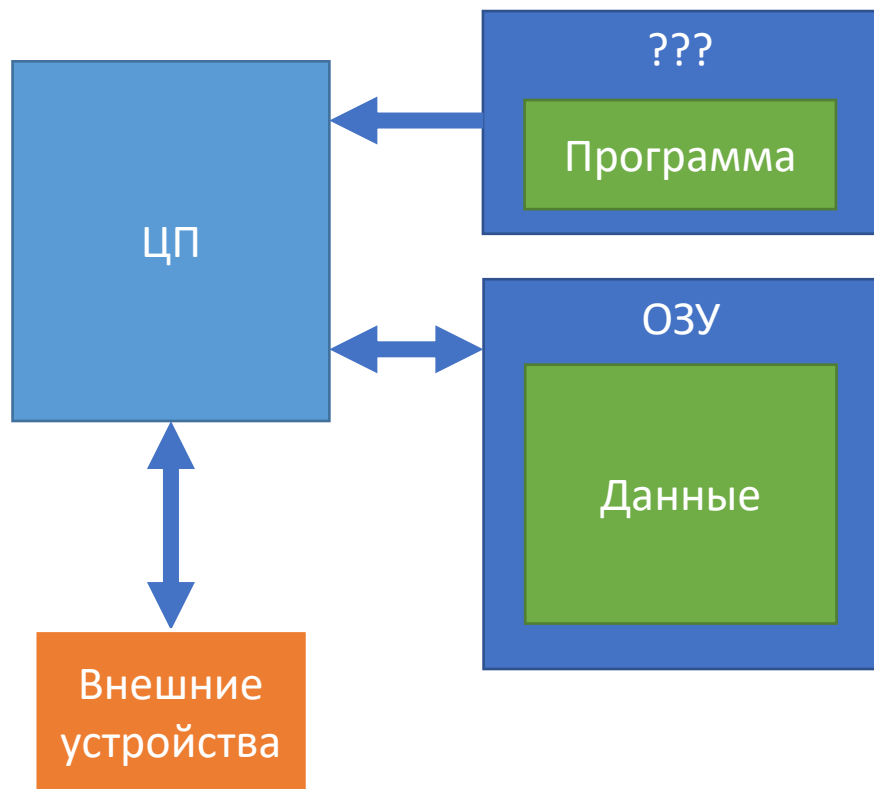
Байт – минимальная адресуемая единица информации.

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.

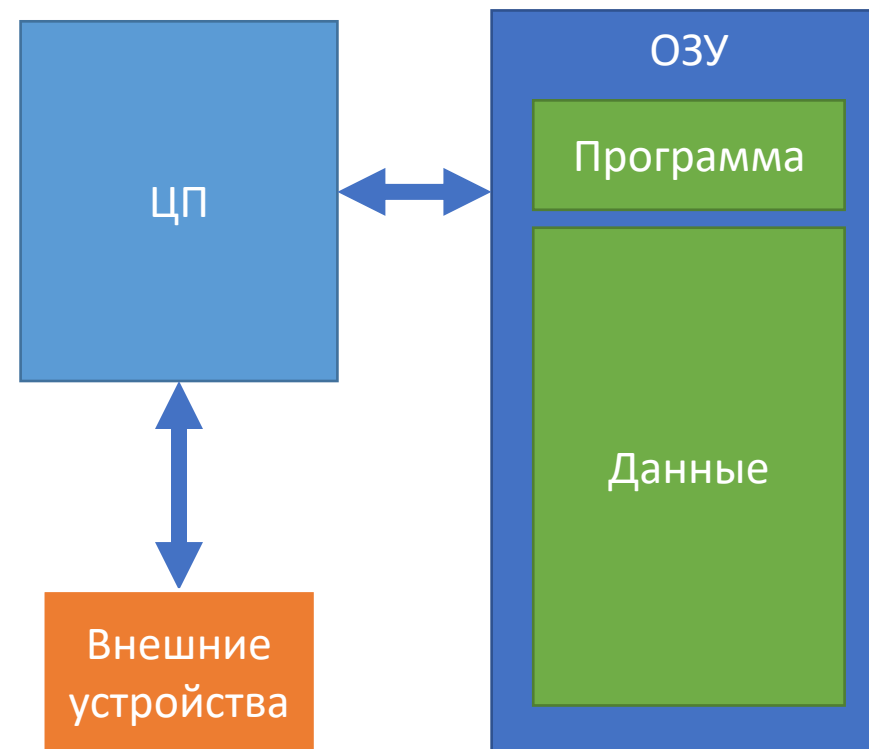
1 - т.е., работает, только пока есть питание

2 - т.е., доступ к любой ячейке памяти занимает одинаковое время

Архитектура ЭВМ



Гарвардская архитектура



Архитектура фон Неймана
(Принстонская архитектура)

Архитектура набора команд

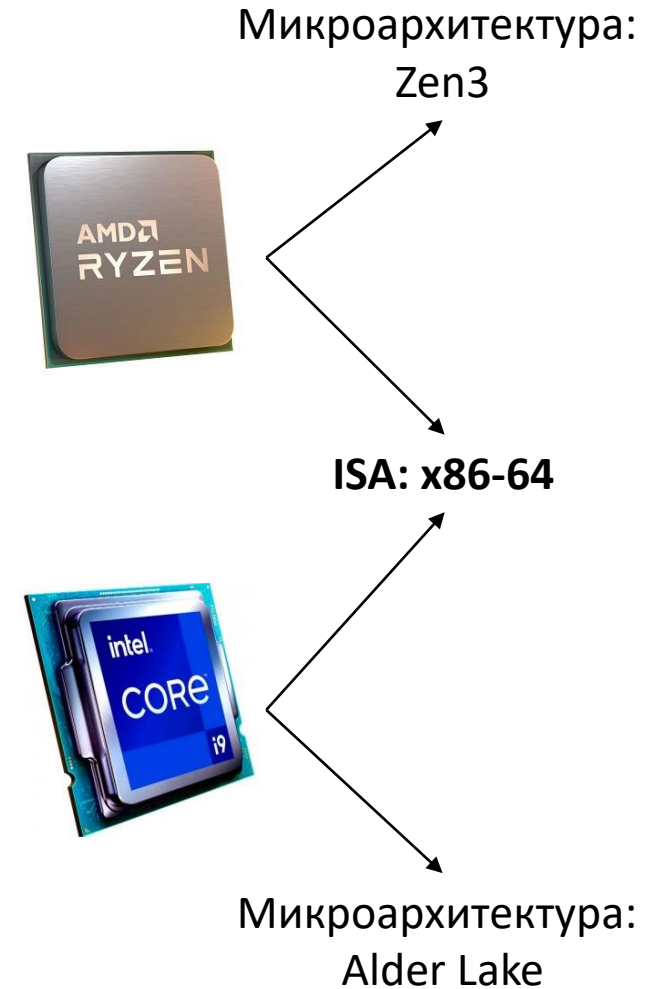
Архитектура набора команд (instruction set architecture, ISA) указывает:

- какие команды процессор может в принципе выполнять и с каким результатом;
- как именно команды кодируются в инструкции.

Программа, скомпилированная под заданную ISA может выполняться на любом совместимом ЦП.

Микроархитектура процессора описывает устройство процессора на аппаратном уровне.

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



Семейство архитектур x86

1978 г – процессор Intel 8086

- [разрядность](#) - 16 бит;
- тактовая частота - 5 МГц;
- шина адреса – 20 бит (макс. 1 МБ ОЗУ);
- [FPU](#) – отдельное устройство (Intel 8087).

1985 г – архитектура **IA-32**¹ (Intel 80386)

- разрядность - 32 бита;
- тактовая частота - 5 МГц;
- шина адреса – 32 бита (макс. 4ГБ ОЗУ);
- FPU интегрирован в кристалл процессора (начиная с i486);

2003 г – архитектура **amd64**² (AMD Opteron)

- разрядность – 64 бита
- тактовая частота - 5 МГц;
- увеличенное число регистров общего назначения;
- шина адреса – до 64 бит (обычно – 48 бит);
- поддержка векторных операций.

¹ - часто называют просто x86

² - синонимы – x86-64, x64

[see also](#)

Регистры x86-64

Регистр – ячейка памяти фиксированного размера, расположенная внутри процессора

RAX	RCX
RBX	RDY

CS	DS
SS	ES

IDTR	
GDTR	LDTR

R8	R9
----	----

...

R14	R15
-----	-----

XMM0	XMM1
------	------

...

XMM6	XMM7
------	------

CR0	CR1
-----	-----

...

CR6	CR7
-----	-----

RDI	RSI
-----	-----

MXCSR

DR0	DR1
-----	-----

RSP	RBP
-----	-----

DR6	DR7
-----	-----

RIP	RFLAGS
-----	--------

■ регистры общего назначения
■ индексные регистры
■ стековые регистры

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

■ табличные регистры
■ управляющие регистры
■ отладочные регистры

Части регистров

Изначально регистры были 16-битными (AX) и состояли из 2 половин (AH, AL).

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).

Физически RAX, EAX, AX, AH и AL – это один регистр

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

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



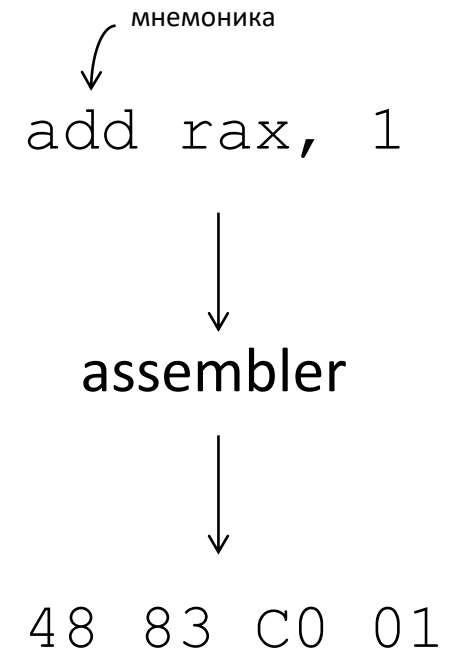
Язык ассемблера

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.



Язык ассемблера NASM

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

- очень простой язык – базовое описание можно уместить на пару страниц (без учета описания инструкций процессора).
- регистронезависимый язык(ADD = add = AdD), исключение – имена меток и секций (см. далее);
- используется синтаксис Intel;
- отсутствие типов данных (нет никаких проверок корректности ваших действий ☺);
- 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.

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

Размер	Подходящий тип C/C++	Размер
byte	char	1
word	short	2
dword	long/float	4
qword	long long/double	8
tword	long double	10

*список неполный

Синтаксис Intel

В нотации Intel инструкции языка ассемблера имеют форму

`<instruction> [приемник/операнд1], [операнд2], [операнд3]`

Примеры:

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

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

Д/З: синтаксис AT&T

Структура программы

Всякий исполняемый файл содержит в себе несколько областей (сегментов).

.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

Для выделения функций и переменных используются **метки**.

Синтаксис метки:

<имя метки>:

При ассемблировании метка заменяется на соответствующий адрес или эквивалентное ему смещение (дистанцию от инструкции до цели, на которую указывает метка).

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>

```
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
    array1: 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
```

Перемещение данных

mov <приемник>, <источник>

- Размер приемника и источника должен быть равен (`mov rax, eax` – нельзя).
- Если операнд – адрес в памяти, то размер перемещаемых данных равен 2 операнду.
- 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 [rbx], rax</code>	*RBX = 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 нельзя, но можно использовать отрицательные числа).

В адресном выражении могут быть указаны максимум 2 регистра.

Метки в адресных выражениях эквивалентны константам.

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

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

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

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

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

```
mov rdx, [array + 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 rsi, [array+4*rcx+4]
    lea rsi, [rsi-4]

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

Стек вызовов

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

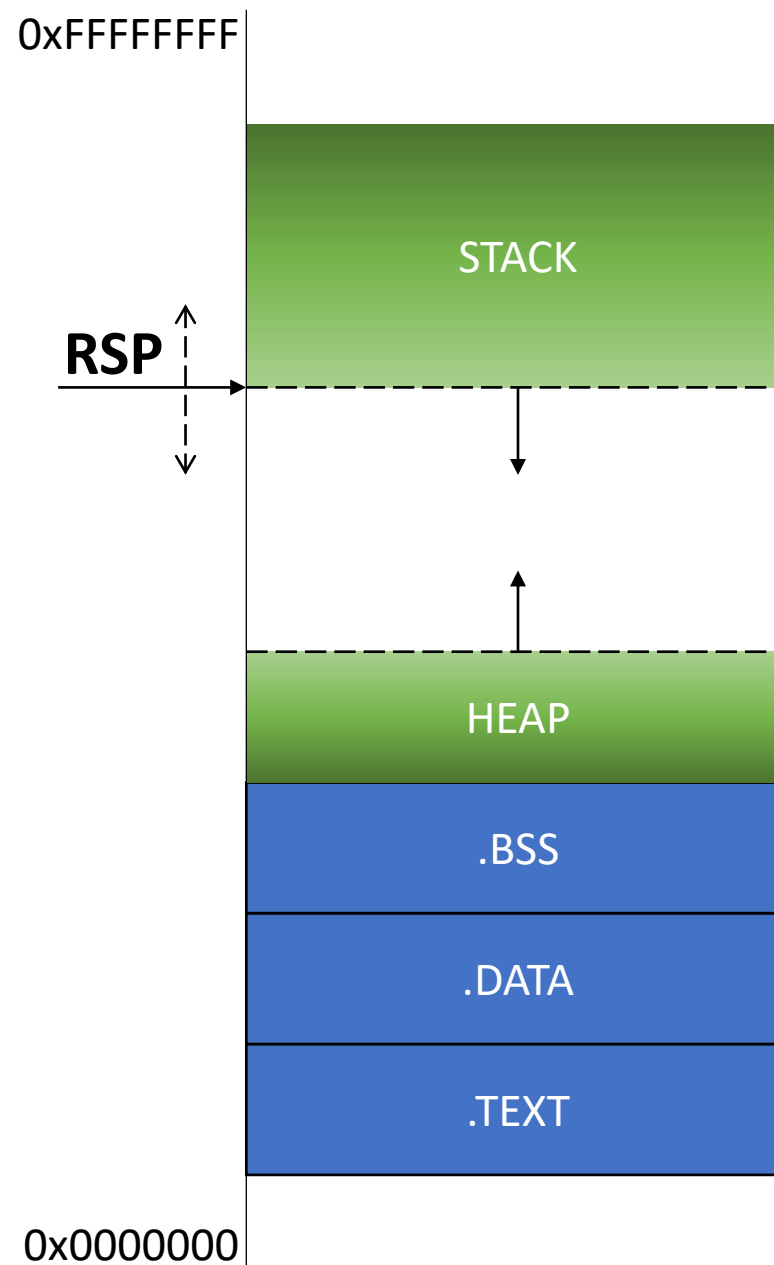
Указатель на вершину стека хранится в регистре **RSP**.

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 положил `int i` по адресу `[RSP-4]` 😊



Доступ к стеку

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], al – 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

Дополнительный код

Отрицательные числа представляются в дополнительном коде
(=> старшие биты отрицательного числа = 1).

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

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

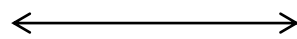
Дополнительный код позволяет выполнять знаковое и беззнаковое сложение/вычитание одинаковым образом.

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

Представление чисел в памяти

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

```
number: dd 0x77AABBCC
```



```
number: db 0xCC, 0xBB, 0xAA, 0x77
```

```
mov ax, [number]; AX=??
```

Д/З: BigEndian и LittleEndian

(см. «Архитектура компьютера», с.97)

Простые инструкции (пример)

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

Битовые операции

И and	Или or	Исключающее или xor	Не not
and ax, 10 and ebx, ebx and dx, word[rsi] and byte[rdi], cl	or ax, 10 or ebx, ebx or dx, word[rsi] or byte[rdi], cl	xor ax, 10 xor ebx, ebx xor dx, word[rsi] xor byte[rdi], cl	not rax

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, cl	ror rax, 5 rol cx, 5 ror edx, cl
AL = -2 = 0xFE=11111110 shr al, 3 AL = 31 = 0x1F=00011111	AL = -2 = 0xFE=11111110 sar al, 3 AL = -1 = 0x1F=11111111	AL = -2 = 0xFE=11111110 ror al, 3 AL = -33 = 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)*.

```
mul    rbx    {RDX:RAX} = RAX*RBX          div    bx    AX = {DX:AX} / BX
                                           DX = {DX:AX} % BX
```

```
imul   ebx    {EDX:EAX} = EAX*EBX          idiv   bl    AL = AX / BL
                                           AH = AX % BL
```

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

* исключение – 1-байтовое умножение и деление, которые используют 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)EAX</code>
<code>movsx qword[rbx], eax</code>	<code>*RBX = (long)EAX</code>

Преобразование чисел

<p>a:dd -2</p> <p>mov EAX,[a]</p>	<p>Было: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0]</p> <p>Стало: RAX = 0x00000000FFFFFFFF [RAX>0, EAX<0]</p>
<p>a:dw -2</p> <p>mov AX,[a]</p>	<p>Было: RAX = 0x7777777777777777 [RAX>0, EAX>0, AX>0]</p> <p>Стало: RAX = 0x7777777777777FFE [RAX>0, EAX>0, AX<0]</p>
<p>a:db 10</p> <p>mov AL,[A]</p>	<p>Было: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0, AX<0, AL<0]</p> <p>Стало: RAX = 0xFFFFFFFFFFFFF0A [RAX<0, EAX<0, AX<0, AL>0]</p>
<p>a: dd -2</p> <p>movsx RAX,dword[a]</p>	<p>Б: RAX = 0x0000000000000000 [RAX=0, EAX=0, AX=0, AL=0]</p> <p>С: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0, AX<0, AL<0]</p>
<p>a:dw -2</p> <p>movsx RAX,word[a]</p>	<p>Б: RAX = 0x7777777777777777 [RAX>0, EAX>0, AX>0, AL>0]</p> <p>С: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0, AX<0, AL<0]</p>
<p>a:db 0x10</p> <p>movzx RAX,byte[a]</p>	<p>Б: RAX = 0xFFFFFFFFFFFFFFFF [RAX<0, EAX<0, AX<0, AL<0]</p> <p>С: RAX = 0x00000000000000FA [RAX>0, EAX>0, AX>0, AL>0]</p>

Выполнение программы. Регистр RIP

Инструкции считываются из памяти и выполняются последовательно.

Специальный регистр RIP указывает на *следующую* инструкцию, которая будет выполнена.

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:
ror eax, cl
```



*допускается также указывать относительное смещение вместо абсолютного адреса

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.

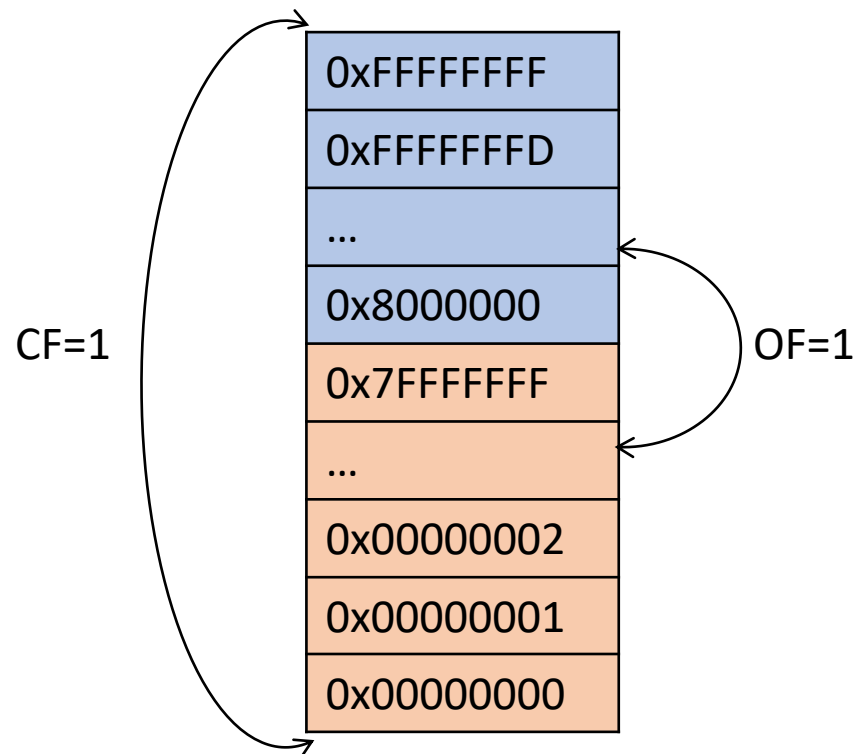
ZF – флаг нуля.

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 имеют форму j^* , где $*$ - символы, задающие условие.

Отрицание условия задается суффиксом n :

$je \rightarrow jne$, $jg \rightarrow jng$.

Допускается комбинировать условия:

$ja+je = jae$, $j1+je = j1e$

Инструкция	Значения флагов
$je(equal)$ $jz(zero)$	$ZF=1$
$jg(greater)$	$SF=0, ZF=OF$
$jl(less)$	$SF=1, ZF \neq OF$
$ja(above, unsigned\ greater)$	$CF=0, ZF=0$
$jb(below, unsigned\ less)$ $jc(carry)$	$CF=1$
$js(sign, less\ than\ zero)$	$SF=1$
$jo(overflow)$	$OF=1$


Циклы (пример)

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

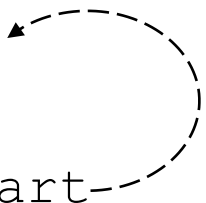
Инструкция `loop` принимает адрес (метку) начала цикла, как аргумент. При исполнении сначала выполняется декремент **ECX**, а потом проверяется его значение. Если `ECX != 0` - происходит прыжок на указанную метку.

```
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
```



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;
2. адрес возврата не должен быть перезаписан;

Помните, что `main()` – тоже функция.

Подробнее – см. лекцию 3.

