

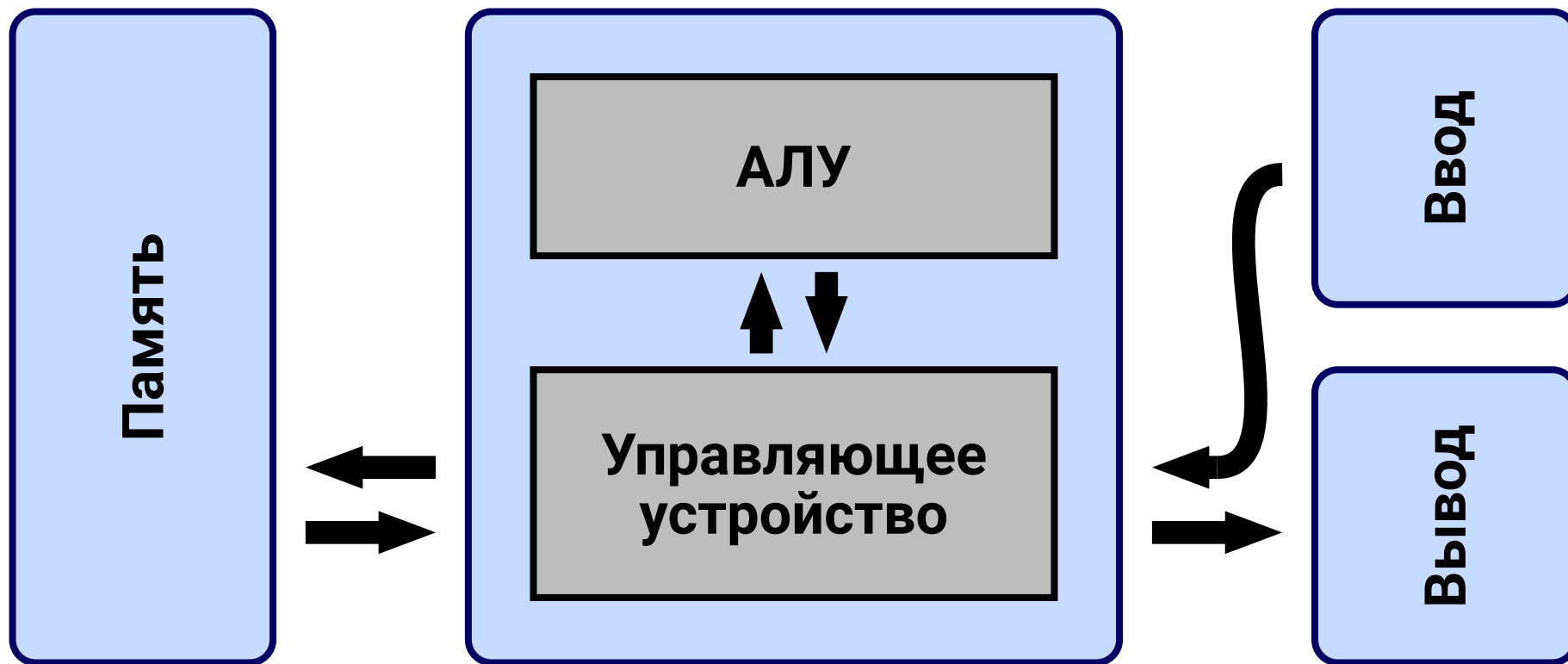
# Ассемблер (x86)

АКОС, МФТИ

03 октября, 2024



# Архитектура фон Неймана





**Игра про создание своего процессора из реле**

## С чем работает процессор?

- + С тривиальными инструкциями, закодированными в бинарном виде;
- + С 8-, 16-, 32-, 64-битными числами (иногда до 512 бит);
- + С адресами памяти (опять же, в виде чисел).
- + Со стеком (регистр для указателя на стек `sp`, инструкции `push` / `pop`).

## С чем не работает процессор?

- Структуры, классы, строки – всё это абстракции над адресами памяти;
- Массивы данных. Каждый элемент нужно обрабатывать отдельно.
- Функции, методы, исключения. Всё, что есть – это стек;

**Задача процессора** – уметь очень быстро выполнять простые инструкции, из которых уже можно составлять сложные программы.

# Что такое инструкция?

**mov** **rbx**, **rax**

The diagram shows the instruction 'mov rbx, rax' with three horizontal brackets above it. From each bracket, a vertical line descends to a label. The first bracket is under 'mov' and points to 'Мнемоника (название операции)'. The second bracket is under 'rbx' and points to 'Первый операнд'. The third bracket is under 'rax' and points to 'Второй операнд'.

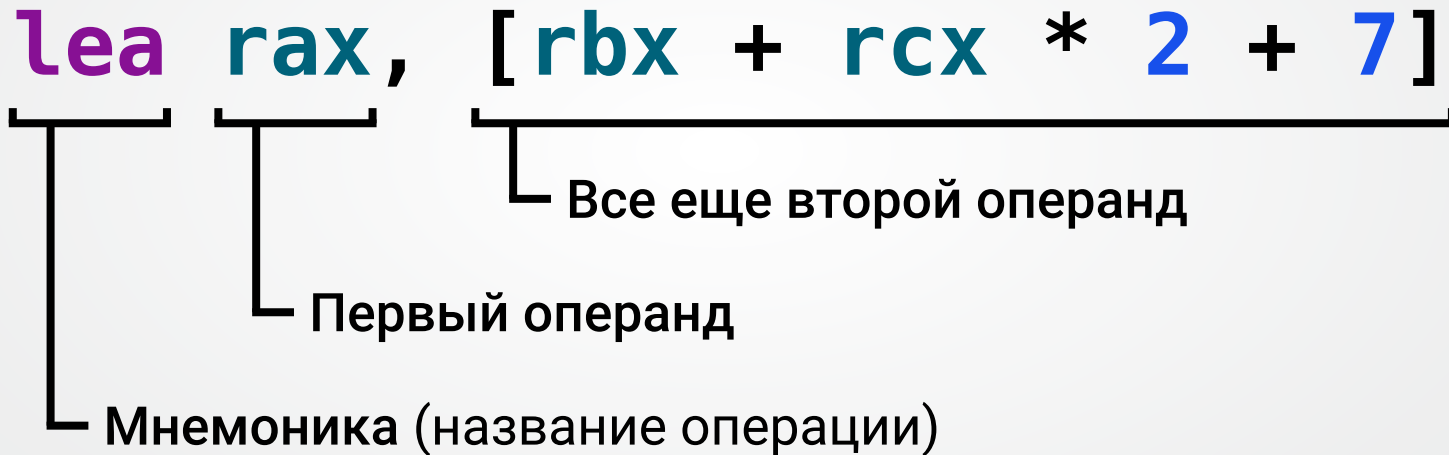
Мнемоника (название операции)

Первый операнд

Второй операнд

Для процессора она выглядит как **48 89 C3**

# Что такое инструкция?



Для процессора она выглядит как **48 8D 44 4B 07**

# Наборы инструкций

## x86 (x86\_64)

- Наиболее распространённый на **настольных компьютерах** / ноутбуках;
- Использует **CISC** - Complex Instruction Set.

## arm(v1 ... v8)

- Живёт на **большинстве телефонов**, роутеров, во встраиваемых системах;
- Сейчас захватывает и настольный рынок (Apple M1, Snapdragon X);
- Использует **RISC** - Reduced Instruction Set.

Еще есть MIPS и PowerPC , но их мы не затронем.

**CISC** : инструкции могут иметь разную длину

`nop`

`0x90`

`mov rbx, rax`

`0x48 0x89 0xC3`

`cmovg rax, r15`

`0x49 0x0F 0x4F 0xC7`

**RISC** : все инструкции равнодлинные

`nop`

`0x1F 0x20 0x03 0xD5`

`mov x1, x0`

`0xE0 0x03 0x01 0xAA`

`cse1 x0, x15, x0, gt`

`0xE0 0xC1 0x80 0x9A`



**x86**

# 16-битные регистры общего назначения

<b>ax</b>	Accumulator
<b>bx</b>	Base index (для работы с массивами)
<b>cx</b>	Counter
<b>dx</b>	Accumulator extension
<b>r8w - r15w</b>	У этих даже названия нет, просто регистры
<b>sp</b>	Stack pointer
<b>bp</b>	Base pointer
<b>si</b>	Source index
<b>di</b>	Destination index

## Специальные 16-битные регистры:

<b>ip</b>	Instruction pointer
<b>flags</b>	Скрытый регистр флагов: переполнение, чётность, знак, ...

# Вложенность регистров – **ax**

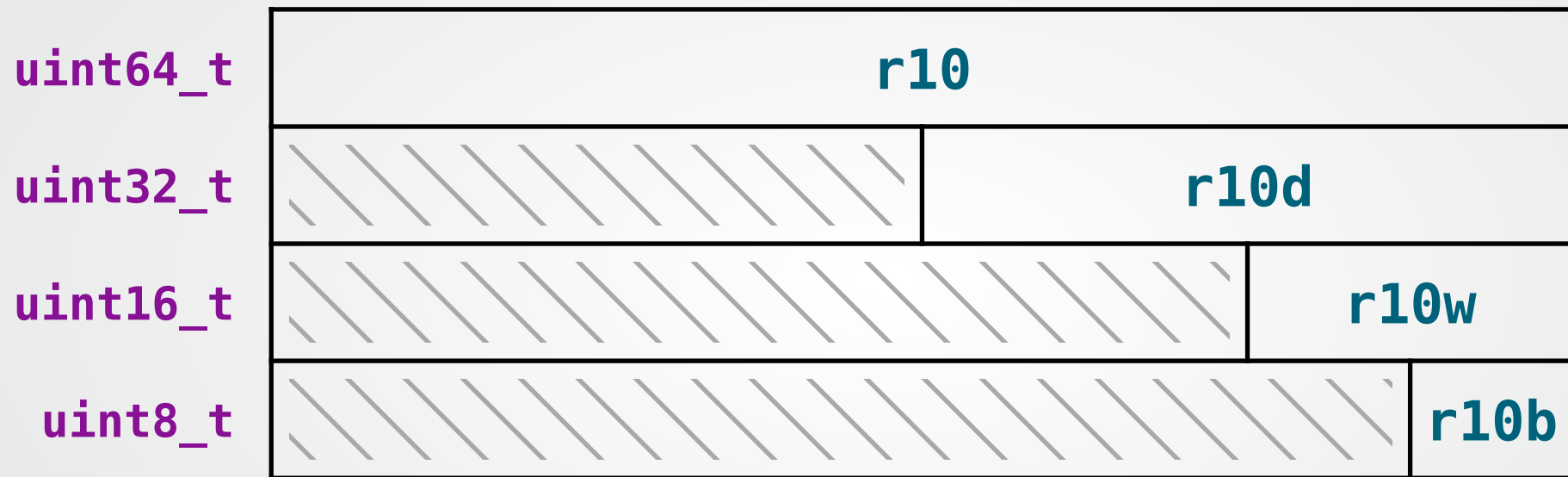
<b>uint64_t</b>	<b>rax</b>		
<b>uint32_t</b>			<b>eax</b>
<b>uint16_t</b>			<b>ax</b>
<b>uint8_t</b>			<b>ah</b>
			<b>al</b>

Аналогично работают **bx**, **cx**, **dx**

**sp**, **bp**, **si**, и **di**, не имеют вариантов с **h** в конце.

**ip** в чистом виде не встречается, только **eip** и **rip**.

# Вложенность регистров – r10



Аналогично работают r8 ... r15

# Intel

```
mov rax, 5
lea eax, [ecx + ebx * 2 + 7]
and DWORD PTR [eax], 7
```

- Порядок операндов: куда ← откуда
- Адреса: [base + index \* scale + offset]
- Битность инструкции обычно угадывается;
- Если не угадывается, то есть директивы:
  - BYTE PTR - 1 байт
  - WORD PTR - 2 байта
  - DWORD PTR - 4 байт
  - QWORD PTR - 8 байт

# AT&T

```
movq $5, %rbx
leal 7(%ecx, %ebx, 2), %eax
andl $7, (%eax)
```

- Порядок операндов: откуда → куда
- Адреса: offset(base, index, scale)
- Битность явно указывается суффиксом:
  - b - 1 байт
  - w - 2 байта
  - l - 4 байт
  - q - 8 байт
- Перед каждым регистром %, перед каждым числом \$

# Базовые инструкции

<code>mov</code>	Перенести значение между регистрами / памятью.
<code>lea</code>	Загрузить адрес второго операнда в первый.
<code>add</code> <code>sub</code> <code>and</code> ...	Арифметика / логика над значением в регистре / памяти. Работают как <code>+=</code> , <code>-=</code> , <code>&amp;=</code> , <code> =</code> , <code>^=</code> , <code>~=</code> , ... в Си.
<code>neg</code>	Поменять знак значения в регистре / памяти
<code>imul</code> <code>mul</code>	Знаковое / беззнаковое умножение.
<code>idiv</code> <code>div</code>	Знаковое / беззнаковое деление.
<code>cmp</code>	Математическое сравнение величин (результат в <code>flags</code> ).
<code>test</code>	Побитовое сравнение величин (результат в <code>flags</code> ).
<code>push</code> <code>pop</code>	Добавить / считать значение со стека.
<code>jmp</code>	Переход в другое место программы.
<code>syscall</code>	Системный вызов.

# Минимальная корректная программа на ассемблере:

```
1  _main:  
2      mov rax, SYS_EXIT  
3      mov rdi, (код возврата)  
4      syscall
```

- Выполняет **системный вызов** `exit(int retval)`
- **Номер** системного вызова передаётся в `rax`
- **Аргумент** системного вызова передаётся в `rdi`
- **Системный вызов** выполняется инструкцией `syscall`.

**Как выразить вызов функции?**



```
1  uint64_t global_var;
2
3  void my_func() {
4      // Делаем что-то полезное здесь
5      global_var = 42;
6  }
7
8  void main() {
9      my_func();
10     __exit(global_var);
11 }
```

```
1  # global_var = r10
2
3  _my_func:
4      mov r10, 42
5      # return ...?
6
7  _main:
8      # _my_func() ...?
9
10     mov rax, SYS_EXIT
11     mov rdi, r10
12     syscall
```

# Вариант 1

А так нельзя!

# Вызов функции

**mov** r10, rip

**add** r10, 11

**jmp** \_my\_func

# Выход из функции

**jmp** r10

\* Сдвиг не всегда будет 11 байт, размер инструкции **jmp** может отличаться

## Вариант 2 – `lea`

# Вызов функции


```
lea r10, [rip + 5]
```

```
jmp _my_func
```

# Выход из функции

```
jmp r10
```

Так можно,  
но закончатся регистры



\* И здесь, сдвиг не всегда будет 5 байт

# Вариант 3 – `lea + push`

```
# Вызов функции  
lea rax, [rip + 6]  
push rax  
jmp _my_func
```

```
# Выход из функции  
pop rax  
jmp rax
```

\* Здесь тоже сдвиг может быть другой

# Вариант 4 – а что, так можно было?

# Вызов функции  
`call _my_func`

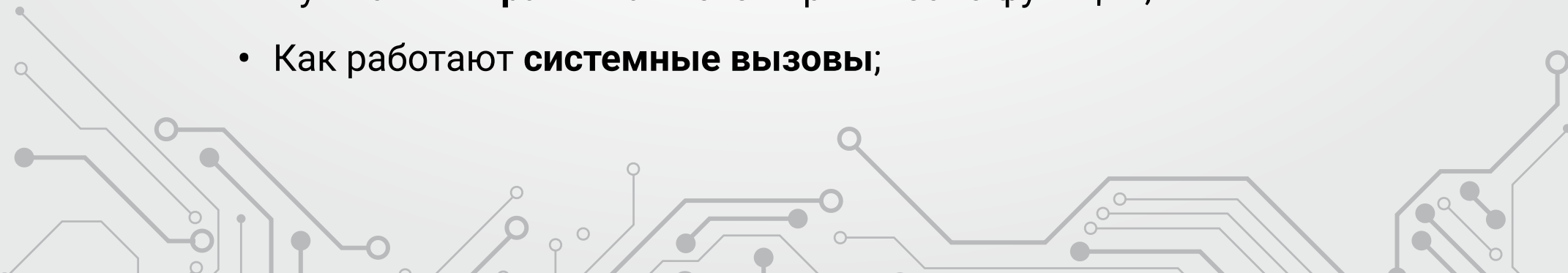
# Выход из функции  
`ret`

# Соглашение о вызове

(Calling convention)

# Что включает в себя соглашение о вызове?

- Где хранится **адрес возврата**;
- Как передаются **аргументы** и возвращаемое значение;
- Какие регистры можно **перезаписывать**, а какие надо **восстанавливать** перед выходом из функции;
- Нужно ли **выравнивать стек** при вызове функции;
- Как работают **системные вызовы**;





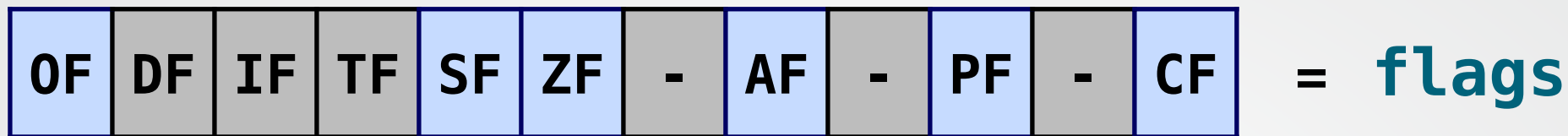
# System V

## Calling Convention

- Аргументы передаются через **rdi, rsi, rdx, rcx, r8, r9** ;
- **float** и **double** - передаются через **xmm0 - xmm7** ;
- То, что не влезло, передаётся **через стек** – справа налево;
- Возвращаемое значение передается в **rax** (и в **rdx**, если не влезает);
- **float** и **double** - возвращаются через **xmm0 - xmm1** ;
- **Стек выравнивается** до двух байт перед вызовом;
- **rbx, rsi, rbp, r12, r13, r14, r15** нужно восстанавливать перед возвратом;
- Номер системного вызова передается через **rax**.

# Сеанс магии

# Регистр флагов



Carry Flag – беззнаковое переполнение  
Parity Flag – последний результат был чётным  
Auxiliary Carry Flag – переполнение нижнего слова  
Zero Flag – последний результат равен нулю  
Sign Flag – последний результат был чётным  
Trap Flag – системный флаг  
Interrupt Enable Flag – системный флаг  
Direction Flag – управляющий флаг  
Overflow Flag – знаковое переполнение

Но как им пользоваться, если он скрыт?

## Условные переносы - **cmov\*\***

<b>cmovs</b>	Перенести, если установлен <b>Sign Flag</b>
<b>cmovz</b> <b>cmove</b>	Перенести, если установлен <b>Zero Flag</b>
<b>cmovp</b> <b>cmovpe</b>	Перенести, если установлен <b>Parity Flag</b>
<b>cmovns</b> <b>cmovnz</b> <b>cmovne</b> <b>cmovnp</b> <b>cmovpo</b>	То же самое, но наоборот (если флаг <b>не</b> установлен)

## Условные переходы - **j\*\***

<b>js</b>	Перейти, если установлен <b>Sign Flag</b>
<b>jz, je</b>	Перейти, если установлен <b>Zero Flag</b>
<b>jp, jpo</b>	Перейти, если установлен <b>Parity Flag</b>

Ещё есть **cset\*\*** - установить один байт в **0** или **1** в зависимости от флагов.

# Больше условных переходов!

Представим, что мы вычли два числа: `sub rax, rbx`. Тогда:

- `rax`  $\geq$  `rbx` , если `CF` = `0`
- `rax`  $\leq$  `rbx` , если `CF` = `1`
- `rax` == `rbx` , если `ZF` = `0`

А если числа были **знаковые**, то:

- `rax`  $\geq$  `rbx` , если `SF` = `0F`
- `rax`  $\leq$  `rbx` , если `SF` != `0F`

Для этого придумали свои суффиксы:

<code>jge</code>	Перейти, если <b>знаково больше или равно</b> ( <code>SF</code> = <code>0F</code> )
<code>ja</code>	Перейти, если <b>беззнаково больше</b> ( <code>CF</code> = <code>1</code> и <code>ZF</code> = <code>0</code> )
<code>jl</code>	Перейти, если <b>знаково меньше</b> ( <code>SF</code> != <code>0F</code> и <code>ZF</code> = <code>0</code> )

Но `sub` поменяет значение `rax`. `cmp` тоже вычитает, но оставляет все как есть.

**Интерактив**

# Стековый фрейм

```
1  void merge_sort(int* arr, int n) {  
2      // Где хранить локальные переменные?  
3      int mid = n / 2;  
4      int left_size = mid, right_size = n - mid;  
5  
6      // ...  
7      merge_sort(left, left_size);  
8      merge_sort(right, right_size);  
9      // ...  
10 }
```

# Стековый фрейм

1 `_merge_sort:`

2 `push rbp`

3 `mov rbp, rsp`

4 `sub rsp, 16` # 16 байт для фрейма

5 `# ...`

6 `mov [rbp - 4], rax` # int mid = ...

7 `mov [rbp - 8], rax` # int left\_size = ...

8 `mov [rbp - 12], rax` # int right\_size = ...

9 `# ...`

10 `mov rsp, rbp`

11 `pop rbp`

12 `ret`

Создание фрейма

Очистка фрейма






# Стековый фрейм

- Если начинать каждую функцию с `push rbp` и `mov rbp, rsp`, то фреймы образуют односвязный список.
- Итерируя по нему, дебаггеры могут показать бектрейс:

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step into
  * frame #0: 0x0000000100003f00 a.out`fib
    frame #1: 0x0000000100003f2d a.out`fib_body + 19
    frame #2: 0x0000000100003f23 a.out`fib_body + 9
    frame #3: 0x0000000100003f23 a.out`fib_body + 9
    frame #4: 0x0000000100003f23 a.out`fib_body + 9
    frame #5: 0x0000000100003f23 a.out`fib_body + 9
    frame #6: 0x0000000100003f23 a.out`fib_body + 9
    frame #7: 0x0000000100003f23 a.out`fib_body + 9
    frame #8: 0x0000000100003f23 a.out`fib_body + 9
    frame #9: 0x0000000100003f23 a.out`fib_body + 9
    frame #10: 0x0000000100003f42 a.out`asm_func + 12
    frame #11: 0x0000000100003f8b a.out`main + 11
    frame #12: 0x0000000200012310 dyld`start + 2432
```

# Полезные ссылки

-  [godbolt.org](https://godbolt.org) – Онлайн-компилятор C / C++ в ассемблер;
-  [online-x86-assembler](https://online-x86-assembler.com) – Онлайн-компилятор ассемблера в машинный код x86;
-  [nandgame.com](https://nandgame.com) – Игра про создание процессора из реле. Советую!



**Спасибо за внимание!**

 [github.com/JakMobius/courses/tree/main/mipt-os-basic-2024](https://github.com/JakMobius/courses/tree/main/mipt-os-basic-2024)