

# Ассемблер (AArch64)

АКОС, МФТИ

10 октября, 2024



## Регистры AArch64 общего назначения:

**x0-x30** | 64-битные регистры общего назначения

**w0-w30** | Их нижние половинки

## Специальные регистры:

**sp** | Stack pointer

**xzr** | 64-битный нулевой регистр

**wzr** | 32-битный нулевой регистр

**pc** | Program counter

**nzcv** | Скрытый регистр флагов: переполнение, знак, обнуление.

Двух- и однобайтных регистров, как в x86, нет.

## x86 : инструкции могут иметь разную длину

`nop`

`mov rbx, rax`

`cmovg rax, r15`

`mov rax, 0xBADBADBEEF`

0x90

0x48 0x89 0xC3

0x49 0x0F 0x4F 0xC7

0x48 0xB8 0xEF 0xBE 0xAD 0xDB 0xBA 0x...

## AArch64 : все инструкции равнодлинные

`nop`

`mov x1, x0`

`cse1l x0, x15, x0, gt`

???

0x1F 0x20 0x03 0xD5

0xE0 0x03 0x01 0xAA

0xE0 0xC1 0x80 0x9A

???

x86\_64

```
mov rax, 0xBADBADBEEF
```

AArch64

```
mov    x8, 0xBEEF
movk   x8, 0xDBAD, lsl 16
movk   x8, 0xBA,    lsl 32
```

В AArch64 все инструкции 4-байтные, поэтому загрузка длинных значений в регистры происходит поэтапно.

# Работа с памятью

x86\_64

Многие инструкции принимают аргументы из памяти:

```
mov rax, [rbx]
add rax, [rbx + 8]
xor [rbx + 8], rax
```

Это называется “[Архитектура регистр-память](#)”.

AArch64

Все инструкции работают **только с регистрами**. Для работы с памятью используются специальные инструкции:

<code>ldr x0, [x1]</code>		<code>load</code> , из памяти в регистр
<code>str x0, [x1]</code>		<code>store</code> , из регистра в память

Это называется “**Архитектура регистр-регистр**” (или [load-store](#)).

# Зачем это всё?

```
add rax, [rbx + 8]
```

**Инструкций меньше**, но процессор всё равно разбивает их на простые операции.

```
ldr x0, [x1]  
add x0, x0, 8
```

Инструкций больше, но **процессор проще**

- **Сокращённый набор инструкций** и load-store архитектура упрощает ядро.
- **Инструкции фиксированного размера** проще декодировать.
- Проще ядро - **выше эффективность**.

# Больше упрощений!

~~push rax~~

~~pop rax~~

## Стек на AArch64 выровнен по 16 байт.

- Регистры - до 8 байт, **push** и **pop** реализовать не получится;
- Сдвигать **sp** и записывать стек нужно вручную;
- Адрес возврата тоже не запушить;
- Опять нужно переизобретать **call**?

# Как работал `call` на x86\_64

# Вызов функции

```
lea rax, [rip + X]
```

```
push rax
```

```
jmp _my_func
```

# Выход из функции

```
pop rax
```

```
jmp rax
```



# “Эквивалентный” код на AArch64

# Вызов функции

**adr** x30, 12

**str** x30, [sp, -16]!

**b** \_my\_func

# Выход из функции

**ldr** x30, [sp], 16

**br** x30

- Сдвиг **pc** всегда будет на 12 байт (размер инструкций фиксирован);
- Но наш аналог **push** x30 теряет 8 байт стека (из-за выравнивания по 16 байт).

# Вспомним про стековый фрейм

x86\_64

1 `_my_func:`

2 `push rbp`

3 `mov rbp, rsp`

4 `# ...`

5 `mov rsp, rbp`

6 `pop rbp`

7 `ret`

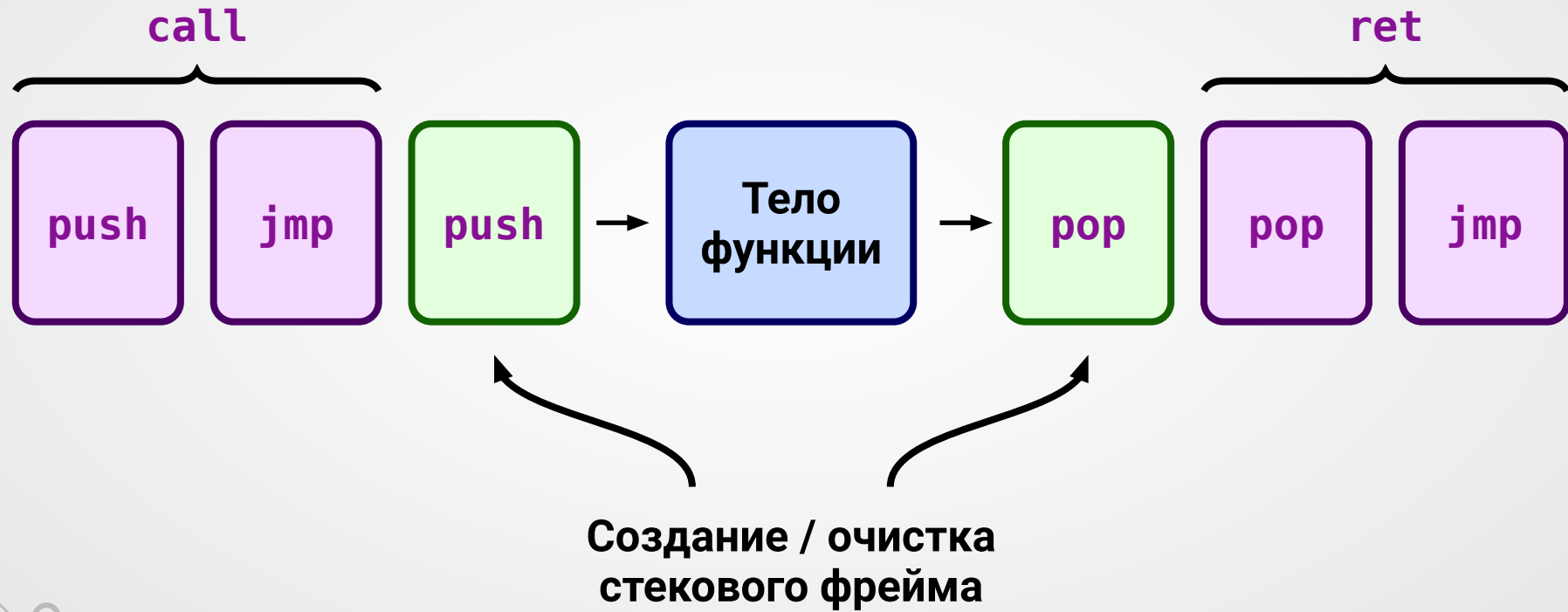
Создание фрейма

Очистка фрейма

- В начале вызванной процедуры происходит **ещё один push** ;
- Перед возвратом происходит **ещё один pop** ;
- Каждый такой **push** и **pop** в AArch64 потерял бы лишние 8 байт стека.

**Можно ли не терять стек?**

# Этапы вызова функции в x86\_64





**Проблема:** каждый **push** использует только половину 16-байтной ячейки стека.

**Что, если склеить вместе два **push** и два **pop**, и использовать все 16 байт?**

# Как объединить два **push**?



Или



**Ответ:** - как сверху, после **jmp** мы уже не восстановим адрес возврата.

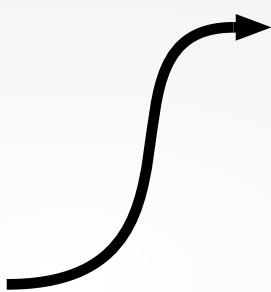
# Объединение **push** и **pop**

```
_main:  
    adr x30, 12  
    str x30, [sp, -16]!  
    b _my_func  
  
# ...
```

```
_my_func:  
    str x29, [sp, -16]!  
    mov x29, sp  
  
# Тело функции  
  
    ldr x29, [sp], 16  
    ldr x30, [sp], 16  
    br x30
```

# Объединение **push** и **pop** – шаг 1 / 2

```
_main:  
  adr x30, 8  
  str x30, [sp, -16]!  
  b _my_func  
  
# ...
```



```
_my_func:  
  str x30, [sp, -16]!  
  str x29, [sp, -16]!  
  mov x29, sp  
  
# Тело функции  
  
  ldr x29, [sp], 16  
  ldr x30, [sp], 16  
  br x30
```



# Объединение **push** и **pop** – шаг 2 / 2

```
_main:  
  adr x30, 8  
  b _my_func  
  
# ...
```

```
_my_func:  
  stp x29, x30, [sp, -16]!  
  mov x29, sp  
  
# Тело функции  
  
  ldp x29, x30, [sp], 16  
  br x30
```

# Объединение **push** и **pop** – шаг 3 / 2

**\_main:**

**adr** x30, 12

**b** \_my\_func

**bl** \_my\_func

# ...

**\_my\_func:**

**stp** x29, x30, [sp, -16]!

**mov** x29, sp

# Тело функции

**ldp** x29, x30, [sp], 16

**br** x30

**ret**

**bl** и **ret** устроены значительно проще, чем **call** и **ret** в x86.

Они работают только с регистрами.

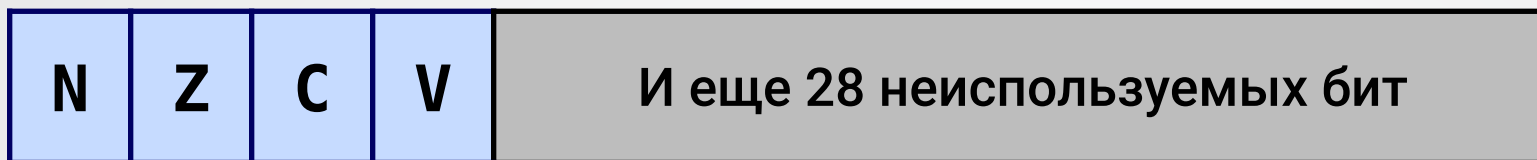
# Соглашение о вызове

<b>x0 - x7</b>	Передача <b>аргументов</b> и <b>возвращаемого значения</b>
<b>x8</b>	Указатель на возвращаемую структуру
<b>x0 - x15</b>	<b>Caller-saved</b> - регистры
<b>x16 - x18</b>	Intra-procedure-call corruptible registers. Проще говоря - <b>Caller-saved</b> .
<b>x19 - x28, sp</b>	<b>Callee-saved</b> - регистры
<b>x29</b>	Frame pointer
<b>x30</b>	Link register, или <b>адрес возврата</b>

**Аргументы** передаются через первые 8 регистров. Лишнее передаётся через стек.

**Флаги**

# Регистр флагов AArch64



= **NZCV**

Overflow Condition Flag – результат знаково переполнился  
Carry Condition Flag – результат беззнаково переполнился  
Zero Condition Flag – результат нулевой  
Negative Condition Flag – результат меньше нуля

# Установка флагов

- По умолчанию, инструкции **не меняют регистр флагов**;
- Это делают только их **версии с суффиксом S**.
- Исключения: **CMP** , **CMN** , **CCMP** , **CCMN** , **TST** – всегда обновляют флаги.

# Условные инструкции

К некоторым инструкциям можно добавить [условный суффикс](#).

<b>EQ</b>	<b>Z</b>	Равно, ноль	Обратный – <b>NE</b>
<b>CS, HS</b>	<b>C</b>	Беззнаково >=	Обратный – <b>CC, LO</b>
<b>MI</b>	<b>N</b>	Меньше нуля	Обратный – <b>PL</b>
<b>VS</b>	<b>V</b>	Переполнение	Обратный – <b>VC</b>
<b>HI</b>	<b>C &amp;&amp; !Z</b>	Беззнаково >	Обратный – <b>LS</b>
<b>GE</b>	<b>N == V</b>	Знаково >=	Обратный – <b>LT</b>
<b>GT</b>	<b>!Z &amp;&amp; N == V</b>	Знаково >	Обратный – <b>LE</b>

# Пример: подсчёт $3^n$

```
1  _pow3:
2      mov x8, 1
3      tst x0, x0
4      b.eq exit
5      loop:
6      add x8, x8, x8, lsl 1
7      subs x0, x0, 1
8      b.ne loop
9      exit:
10     mov x0, x8
11     ret
```

**Интерактив**



**Спасибо за внимание!**

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