Ассемблер x86

Intel x86

- Одна из самых распространённых архитектур процессоров
- Мы будем изучать x86-64 (2003 г.)
- AMD64 – другая архитектура (2001 г.), но почти одинаковая с x86-64
- x64, x86_64, AMD64, Intel 64 – синонимы

Машинный код

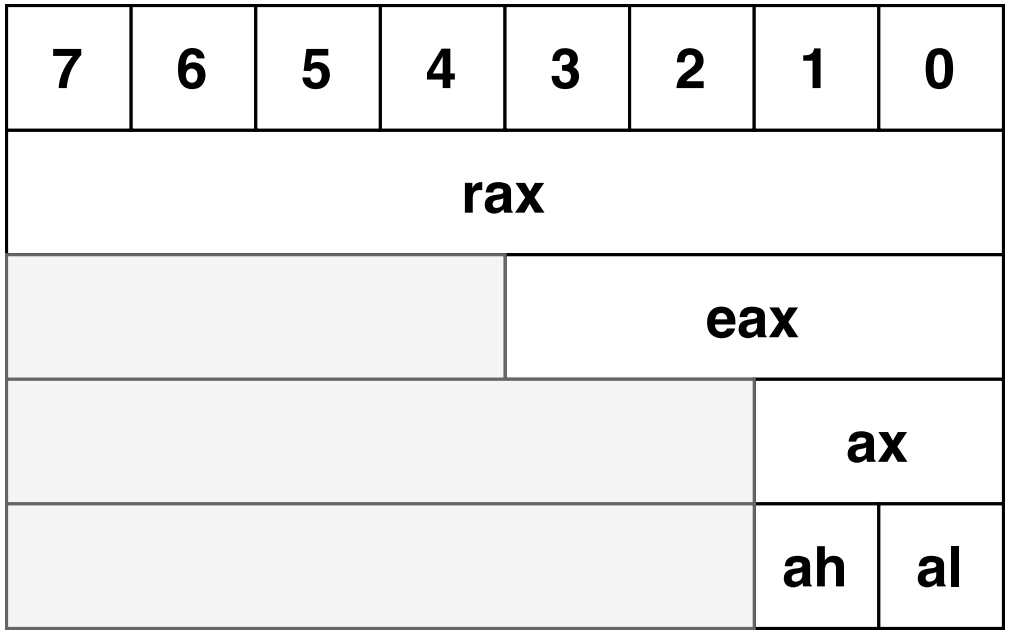
- Инструкции выполняются процессором
- Опкоды (opcodes) – кодировка инструкций
- Мнемоники – человекопонятная расшифровка опкодов
- Платформозависимый!

Регистры

- Очень быстрые ячейки памяти на самом процессоре
- По 64 бита на x86-64
- Числа в two-complement little endian
- Регистры общего назначения (general purpose registers): `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8–r15`
- GPR со специальным значением: `rbp`, `rsp`
- «Виртуальные» регистры: `rip`, `rflags`, …

Вложенность регистров

- Некоторые регистры вложены друг в друга: например, `eax` – нижние 32 бита `rax`
- Запись в `eax` приведёт к обнулению верхних бит `rax`
- Запись в `ax` не перезапишет верхние биты `eax`
- Запись в `ah` / `al` не перезапишет верхние биты `ax`



Какие ещё бывают регистры?

- Управляющие: `cr0`, …, `cr3`
- Отладочные `dr0`, …, `dr4`
- Сегментные: `cs`, `ds`, `ss`, `es`, `fs`, `gs`
- Model-specific: не имеют отдельных обозначений, `wrmsr` / `rdmsr`
- FPU x87: `st(0)`, …, `st(7)`
- SSE: `xmm0`, …, `xmm7`

Как читать документацию x86 ассемблера?

Типы операндов команд:

- `imm8/16/32/64` — константы (располагаются непосредственно в опкоде)
- `r8/16/32/64` — регистры соотв. битности
- `m8/16/32/64` — операнд в памяти

```
mov r/m8,  r8     ; Move r8 to r/m8.
mov r/m16, r16    ; Move r16 to r/m16.
...
mov r/m32, imm32 ; Move imm32 to r/m32.
```

Intel vs AT&T

Intel

```
mov rax, qword ptr [rax, 2 * rcx + 0x10]
```

AT&T

```
movq 0x10(%rax, %rcx, 2), %rax
```

Адресация памяти

```
mov ...,  [base, index * 1/2/4/8 + offset]
```

- `base` и `index` – всегда регистры
- `offset` – всегда константное число

```
mov r8,  [rbp]                  ; r8 = *rbp
mov rcx, [rbx, rdi]             ; rcx = *(rbx + rdi)
mov rax, [rax, rcx * 2 + 0x10] ; rax = *(rax + rcx * 2 + 0x10)
```

Адресация памяти

```
mov [rax], 0x1
```

- `rax` ссылается на 64-битное число?
- Сколько байт запишет эта инструкция?

Адресация памяти

```
mov dword ptr [rax], 0x1
```

- Можно напрямую указать размера операнда памяти
- `byte` / `word` / `dword` / `qword`

Какие бывают инструкции?

- «Вычислительные»
- Управляющие

«Вычислительные» инструкции

```
add rax, 1   ; rax += 1
sub rax, rbx ; rax -= rbx
mul 3        ; rdx:rax = rax * 3
div r8       ; rax = rax / r8, rdx = rax % r8
xor rcx, rcx ; rcx ^= rcx
inc r9       ; r9++
```

https://www.felixcloutier.com/x86/

Флаги

- Регистр `EFLAGS` / `RFLAGS` содержит специальные биты (флаги) результата операции
- Основные флаги:
  - `ZF` : в результате операции получился `0`
  - `SF` : результат отрицательный
  - `OF` : знаковое переполнение
  - `CF` : беззнаковое переполнение

Флаги

- `0x0001 – 0x0001 = 0x0000` (выставится `ZF`)
- `0x0000 – 0x0001 = 0x1111` (выставится `SF`)
- `0x1111 + 0x0001 = 0x0000` (выставится `CF`)
- `0x0111 (7) + 0x0001 (1) = 0x1000 (–8)` (выставится `OF`)

Вычисление флагов

- Есть инструкции для вычисления флагов без изменения регистров общего назначения
- `cmp` = `sub`, который только вычисляет флаги
- `test` = `and`, который только вычисляет флаги

Прыжки

- Прыжки (обычно) выполняются по *метками* в коде
- `j** label_name` перейти на метку
- `jmp` безусловный переход
- `jz` перейти, если выставлен `ZF`
- Для знакомых чисел `jl`, `jle` и т. д.
- Для беззнаковых чисел `jb`, `jbe` и т. д.

Пример: бесконечный цикл

```
loop:
    jmp loop
```

```
while (true) {
}
```

Пример: аналог for

```
    mov rax, 0
    mov rcx, 0
loop:
    add rax, rcx
    add rcx, 1
    cmp rcx, 100
    jnz loop
```

```
int rax = 0;
for (int rcx = 0; rcx < 100; rcx++) {
    rax += rcx;
}
```

Пример: if

```
    test rax, rax
    jz   next
    mov  rcx, 0xdeadbeef
next:
    ...
```

```
if (rax != 0) {
    rcx = 0xdeadbeef
}
```

```
    test rax, rax
    jnz  body
next:
    ...

body:
    mov  rcx, 0xdeadbeef
    jmp  next
```

Пример: if с &&

```
    test rax, rax
    jz   else
    cmp  dword ptr [rax], 1337
    jg   else
    mov  rcx, 0xdeadbeef
    jmp  next
else:
    xor  rcx, rcx
next:
    ...
```

```
if (rax != 0 && *(uint32_t)rax <= 1337) {
    rcx = 0xdeadbeef
} else {
    rcx = 0
}
```

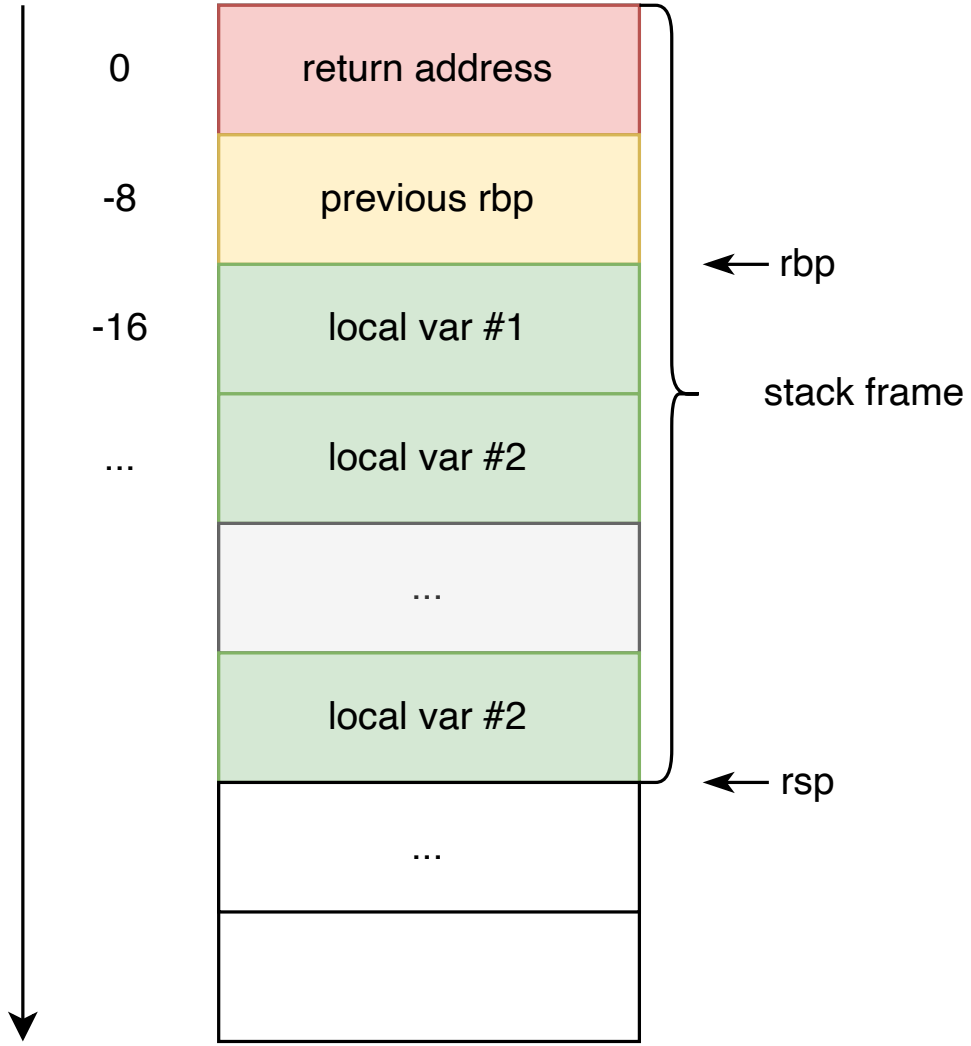
Пример: цикл с условием внутри

```
    mov rax, 0
    mov rdx, 0
    mov rcx, 0
loop:
    test rcx, 1
    jnz odd
    add rax, rcx
    jmp next
odd:
    add rdx, rcx
next:
    add rcx, 1
    cmp rcx, 100
    jnz loop
```

```
int rax = 0;
int rdx = 0;
for (int rcx = 0; rcx < 100; rcx++) {
    if (rcx % 2 == 0) {
        rax += rcx;
    } else {
        rdx += rcx;
    }
}
```

Стек

- Стек – просто область памяти
- Стек растёт вниз!
- На текущую вершину (первый свободный байт) указывает `rsp` (stack pointer)



Стек: `push` и `pop`

```
push 1337

; или

sub rsp, 8
mov [rsp], 1337
```

```
pop rax

; или

mov rax, qword ptr [rsp]
add rsp, 8
```

Функции: `call` и `ret`

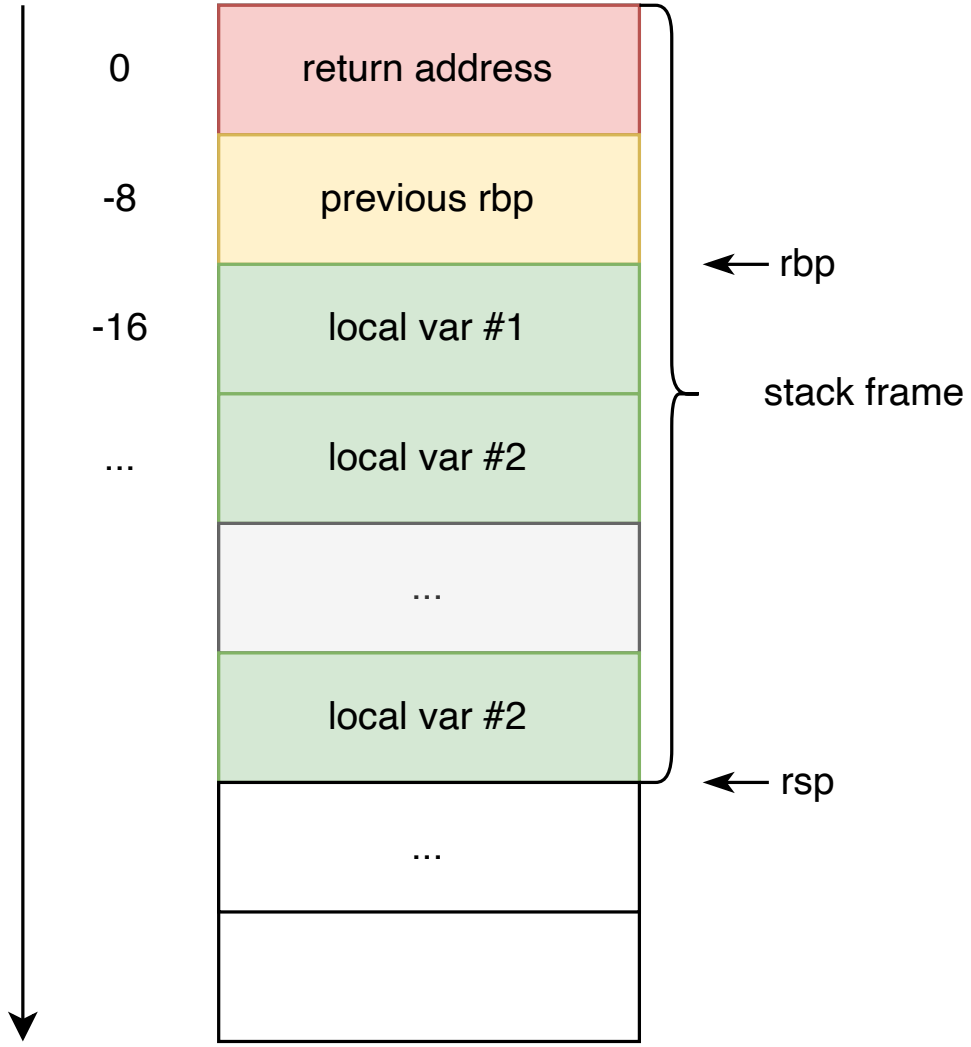
- Стек позволяет реализовывать функции
- `call` вызывает функцию – кладёт адрес следующей инструкции на стек и прыгает по метке
- `ret` возвращается из функции – достаёт адрес возврата со стека

```
call func
test rax, rax
    ; ...

func:
    ; ...
```

Функции: фрейм стека, пролог и эпилог

- Фрейм – область стека
- Фрейм – виртуальное понятие, т.е. он не сущствует для процессора
- Каждый вызов функции создаёт новый фрейм, в котором лежит адрес возврата
- Функция сохраняет текущий `rbp` на стеке и устанавливает новый в текущую вершину стека
- Такая последовательность действий называется прологом, а обратная – эпилогом

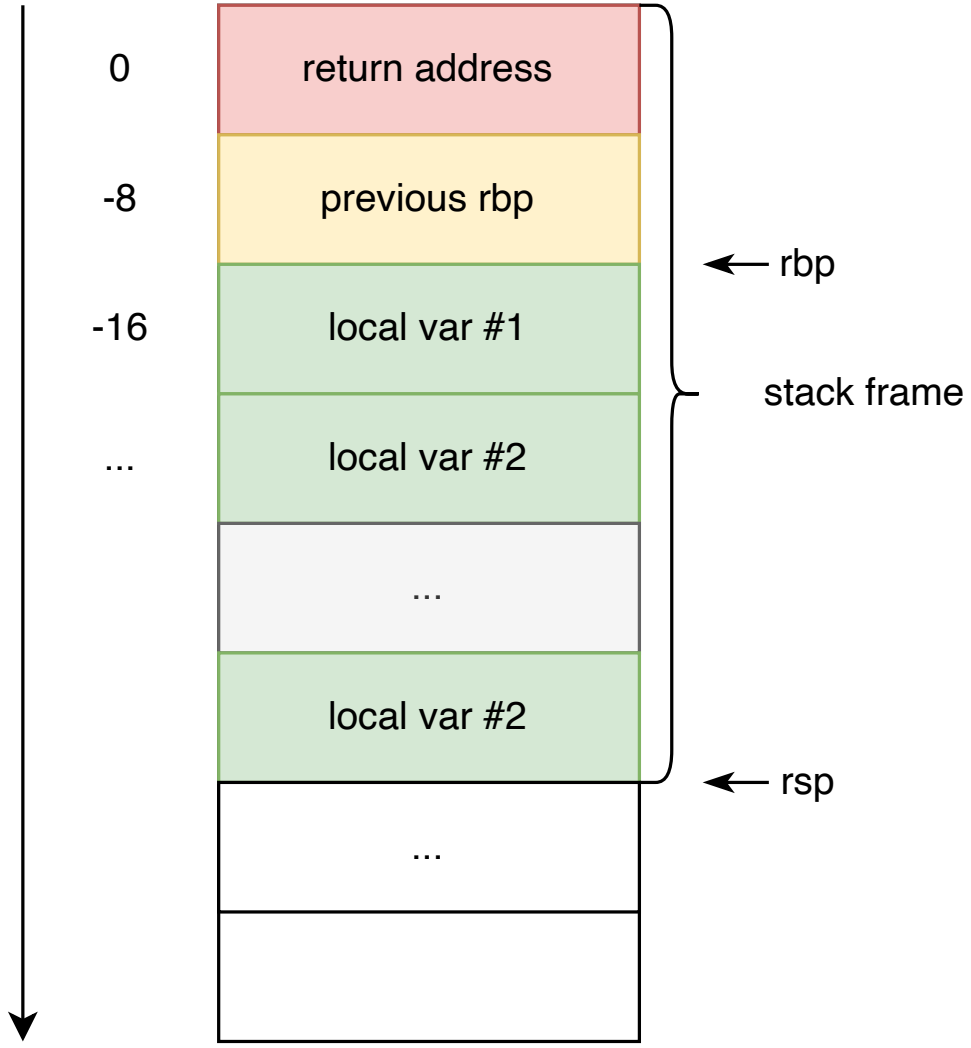


Функции: пролог и эпилог

```
; Prologue
push rbp
mov  rbp, rsp
sub  rsp, 100

...

; Epilogue
mov rsp, rbp
pop rbp
ret
```



Calling conventions

- Соглашения о вызовах – правила о том, как разные программы вызывают друг друга
- System V AMD64 ABI (application binary interface)
- Описывают, например, как вызывать функции библиотеки

Calling conventions

- При вызове функций вершин стека (`rsp`) должна быть выровнена по 16 байт
- Нужно сохранить значения `rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11`, если они вам нужны (scratch registers)
- `rbx, rsp, rbp, r12, r13, r14, r15` нужно вернуть в неизменном виде после выхода из функции (preserved registers)

Merci!