Работа со стеком в x86

Сорокин Иван

Введение

Для чего требуется стек?

В отличии от глобальных переменных для локальных переменных нельзя назначить фиксированный адрес в памяти. Функции бывают рекурсивные и может потребоваться несколько копий локальных переменных.

Поскольку функции вызываются и выходят в порядке стека, можно использоваться стек для хранения данных у которых требуется своя копия для каждого вызова функции.

Введение

Архитектура x86 предоставляет специальные операции для работы со стеком.

Этот стек называется стеков вызова фунций (call stack).

Инструкции работы со стеком используют регистр общего назначения RSP (Stack Pointer).

64-битные регистры

RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSP	ESP	SP		SPL
RBP	EBP	BP		BPL
RSI	ESI	SI		SIL
RDI	EDI	DI		DIL

R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

64-битные регистры

RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSP	ESP	SP		SPL
RBP	EBP	BP		BPL
RSI	ESI	SI		SIL
RDI	EDI	DI		DIL

R8	R8D	R8W	R8B
R9	R9D	R9W	R9B
R10	R10D	R10W	R10B
R11	R11D	R11W	R11B
R12	R12D	R12W	R12B
R13	R13D	R13W	R13B
R14	R14D	R14W	R14B
R15	R15D	R15W	R15B

Инструкции PUSH/POP

Двумя самыми простыми инструкциями для работы со стеком являются инструкции PUSH и POP. PUSH вставляет 64-битное число в стек, а POP извлекает 64-битное число из стека.

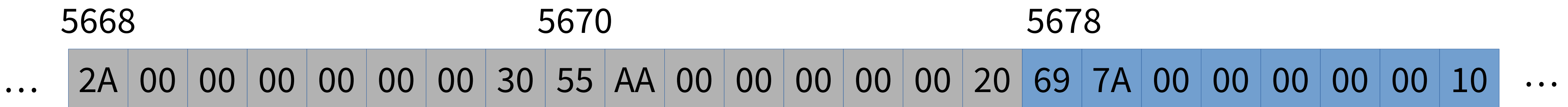
```
PUSH src            RSP = RSP – 8
                    MEM[RSP] = src

POP dst             tmp = MEM[RSP]
                    RSP = RSP + 8
                    dst = tmp
```

```
50          PUSH RAX
FF 33       PUSH QWORD [RBX]
6A 42       PUSH 42
58          POP RAX
8F 03       POP QWORD [RBX]
```

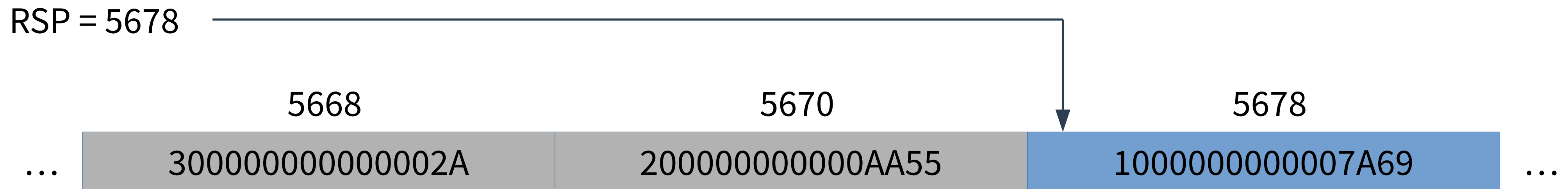
Стек

Стек

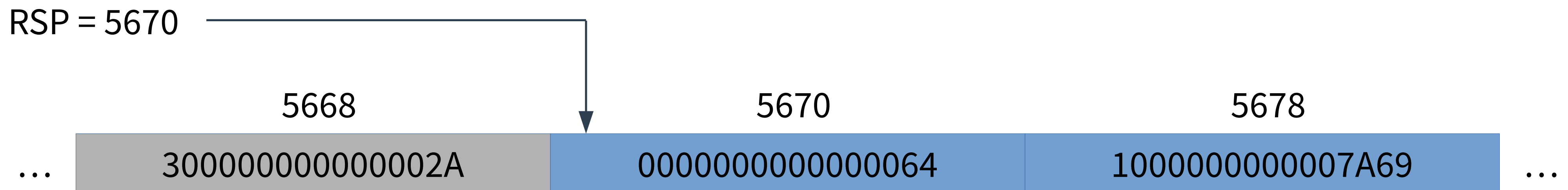
RSP = 5678



Стек



После исполнения инструкции `PUSH 100` содержимое памяти примет следующий вид:



Обратите внимание, что стек растёт в сторону младших адресов.

Инструкции PUSH/POP

```
PUSH src        RSP = RSP – 8
                MEM[RSP] = src

POP dst         tmp = MEM[RSP]
                RSP = RSP + 8
                dst = tmp
```

Этот псевдокод написан так, чтобы быть верным даже когда аргументом инструкции является сам RSP. В этом случае PUSH вставляет уже уменьшенное значение RSP, а POP просто читает память и записывает в RSP.

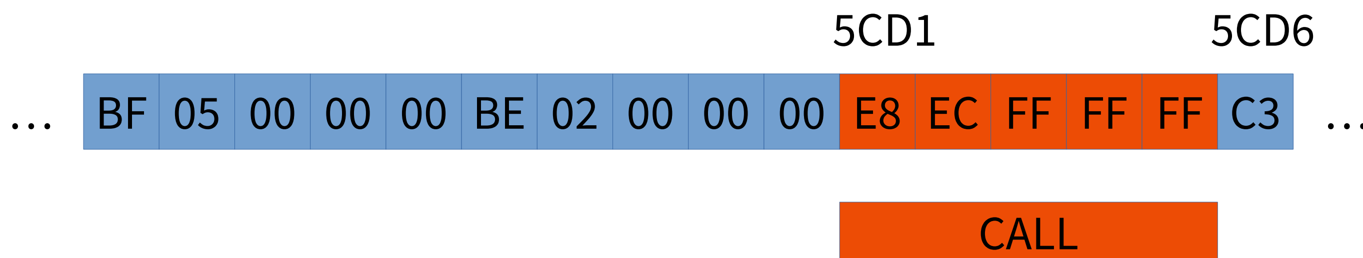
Инструкции CALL/RET

Ещё одной парой инструкция для работы со стеком являются инструкции CALL и RET. CALL используется для вызова функций, а RET для возврата из функций.

```
CALL func          PUSH <адрес следующей после CALL инструкции>
                   JMP func

RET                POP tmp
                   JMP tmp
```

Инструкции CALL/RET

Что означает «`адрес следующей после CALL инструкции`»? Рассмотрим на примере. Пусть инструкция CALL размещена по адресу 5CD1 и занимаете 5 байт, тогда адрес следующей инструкции — 5CD6. Его инструкция CALL и запишет в стек. А после RET исполнение продолжится с этого адреса.




```
CALL func            PUSH <адрес следующей после CALL инструкции>
                     JMP func

RET                  POP tmp
                     JMP tmp
```

Передача параметров в регистрах

```
func:
          mov   eax, edi
          sub   eax, esi
          ret



          ...
          mov   edi, 5
          mov   esi, 2
          call  func                  ; f(5, 2)
          ; eax = 3
          mov   edi, 10
          mov   esi, 6
          call  func                  ; f(10, 6)
          ; eax = 4
          ...
```

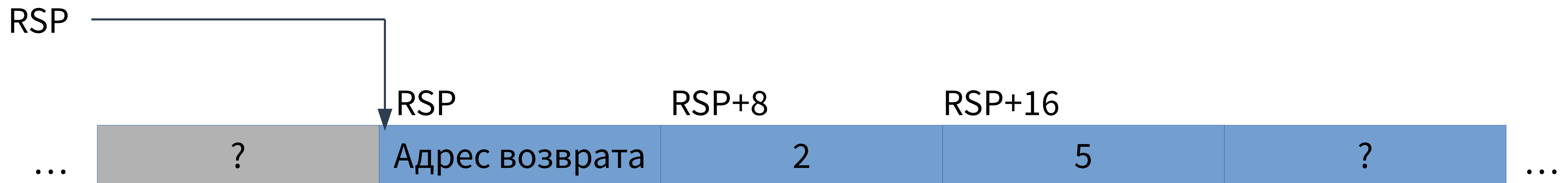
Передача параметров через стек

RSP

…	?	Адрес возврата	2	5	?	…
		RSP	RSP+8	RSP+16		

```
func:
        mov   rax, ?
        sub   rax, ?
        ret

        ...
        push  5
        push  2
        call  func
        ...
```

Передача параметров через стек

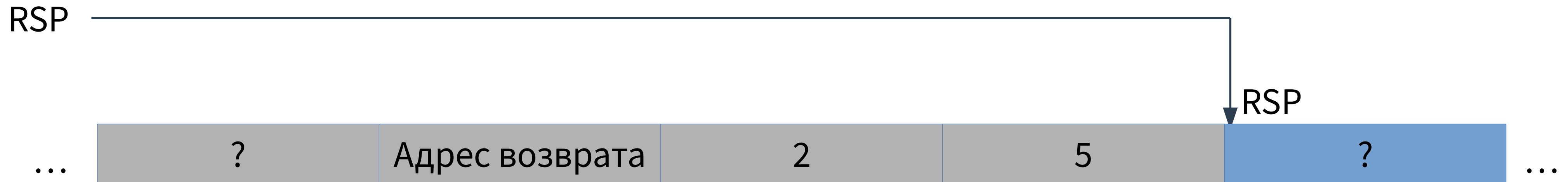



```
func:
        mov   rax, [rsp+16]
        sub   rax, [rsp+8]
        ret

        ...
        push  5
        push  2
        call  func
        ...
```

Передача параметров через стек

```
func:
        mov   rax, [rsp+16]
        sub   rax, [rsp+8]
        ret

        ...
        push  5
        push  2
        call  func
        ...
```

Передача параметров через стек




```
func:
        mov   rax, [rsp+16]
        sub   rax, [rsp+8]
        ret

        ...
        push  5
        push  2
        call  func
        add   rsp, 16
        ...
```

Передача параметров через стек

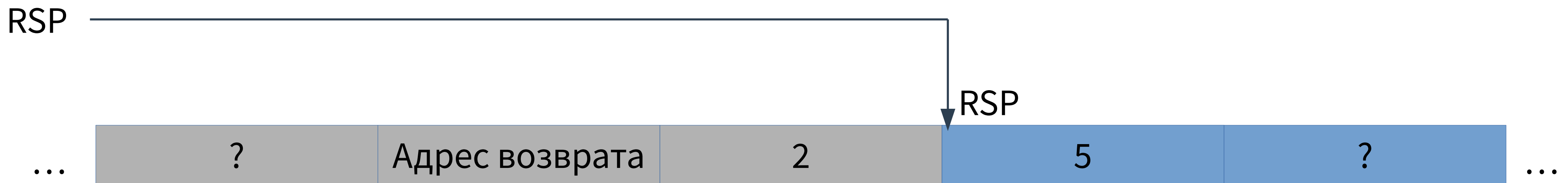
Можно ли внести **`add rsp, 16`** внутрь функции?

```
func:
        mov   rax, [rsp+16]
        sub   rax, [rsp+8]
        add   rsp, 16
        ret

        ...
        push  5
        push  2
        call  func
        ...
```

Передача параметров через стек

```
func:
        mov   rax, [rsp+16]
        sub   rax, [rsp+8]
        add   rsp, 16
        ret

        ...
        push  5
        push  2
        call  func
        ...
```

Передача параметров через стек



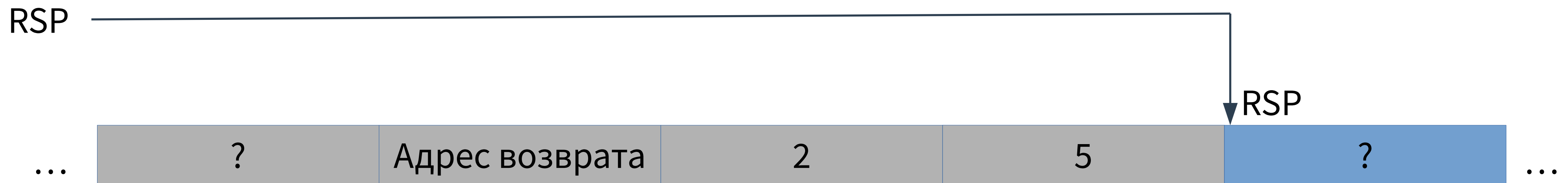
Нельзя вносить **add rsp, 16** внутрь функции.

```
func:
        mov   rax, [rsp+16]
        sub   rax, [rsp+8]
        add   rsp, 16
        ret

        ...
        push  5
        push  2
        call  func
        ...
```

Инструкция ret n

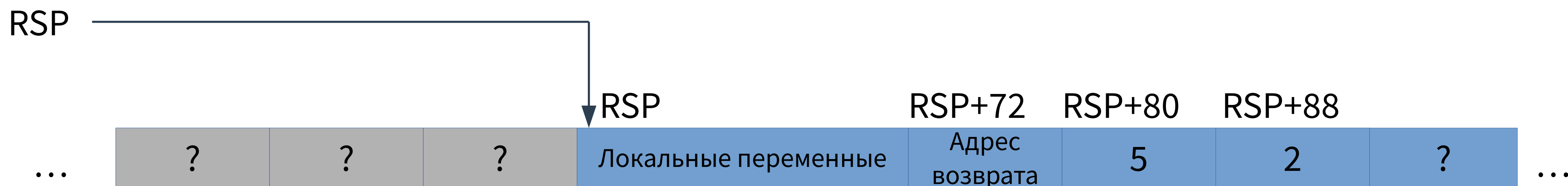
```
RET            POP tmp
               JMP tmp

RET n          POP tmp
               RSP = RSP + n
               JMP tmp
```

Передача параметров через стек




```
func:
        mov   rax, [rsp+16]
        sub   rax, [rsp+8]
        ret   16

        ...
        push  5
        push  2
        call  func
        ...
```

Локальные переменные




```
func:
        sub   rsp, 72
        mov   rax, [rsp+80]
        sub   rax, [rsp+88]
        add   rsp, 72
        ret

        ...
        push  2
        push  5
        call  func
        add   rsp, 16
        ...
```

Соглашения вызова функций

Набор соглашений между вызывающей и вызываемой функциями о том:

- Как передавать аргументы?
  - В регистрах или в стеке? Если в регистрах, то в каких? Если в стеке, то в каком порядке?
  - Если в стеке, то вызывающая или вызываемая функция будет чистить стек?
- В каких регистрах возвращать результат?
- Значения каких регистров, должны остаться неизменными после вызова функции?
- Другие особенности
  - Наличие red zone'ы (Itanium ABI)
  - Резервируются ли слоты в стеке, для аргументов передаваемых в регистрах (аля MSVC 64-bit)
  - Как передаются структуры

Соглашения вызова функций

	Регистры для параметров	Порядок параметров в стеке	Чистка стека
Linux 32-bit	—	←	Вызывающая
Linux 64-bit	rdi, rsi, rdx, rcx, r8, r9	←	Вызывающая
Windows 32-bit __cdecl	—	←	Вызывающая
Windows 32-bit __stdcall	—	←	Вызываемая
Windows 32-bit __thiscall	ecx	←	Вызываемая
Windows 32-bit __fastcall	ecx, edx	←	Вызываемая
Windows 64-bit	rcx, rdx, r8, r9	←	Вызывающая

Variadic функции

```
printf("hello %d %d %d", 1, 2, 3);
```

```
push 3
push 2
push 1
push str
call printf
add rsp, 32
```

Стековый фрейм

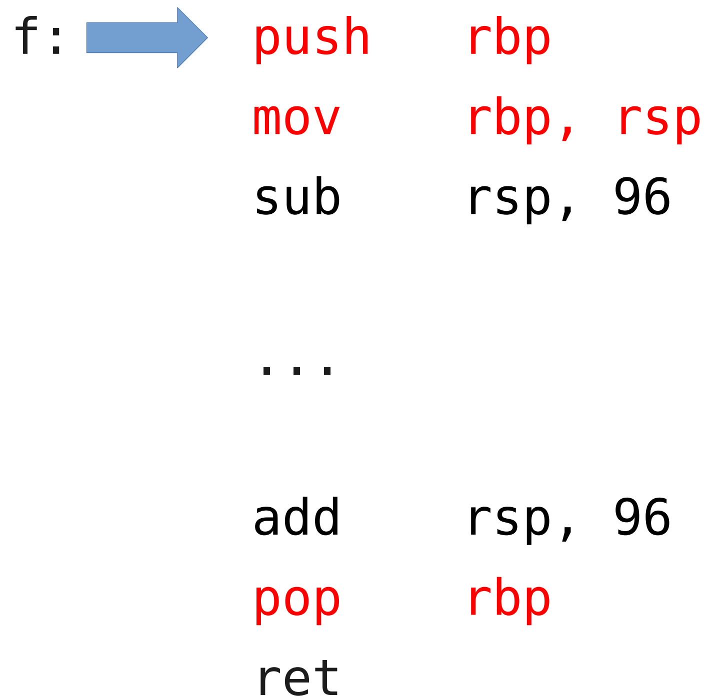
```
f:      push    rbp
        mov     rbp, rsp

        ...

        pop     rbp
        ret
```

Стековый фрейм

```
f:      push    rbp
        mov     rbp, rsp
        sub     rsp, 96

        ...

        add     rsp, 96
        pop     rbp
        ret
```

Стековый фрейм

Адрес возврата	Параметры f	...



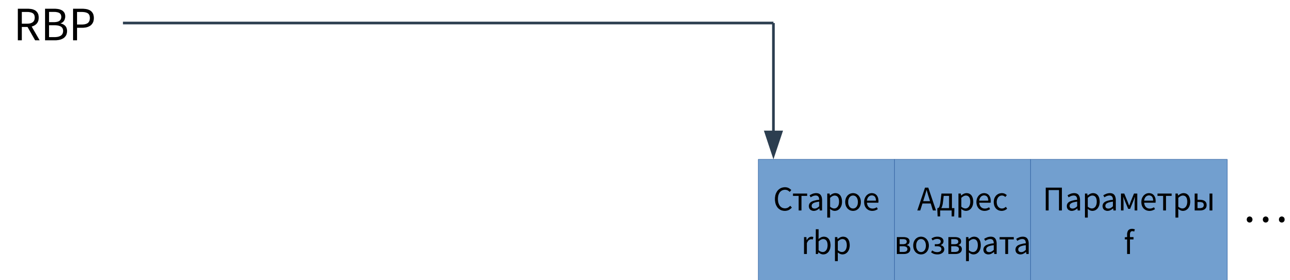
```
f:  push    rbp
    mov     rbp, rsp
    sub     rsp, 96

    ...

    add     rsp, 96
    pop     rbp
    ret
```

Стековый фрейм

Старое rbp	Адрес возврата	Параметры f	...

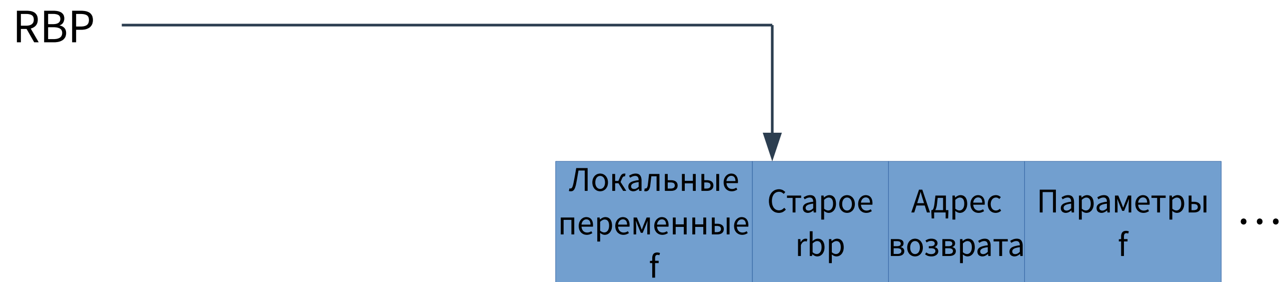
```
f:      push    rbp
        mov     rbp, rsp
        sub     rsp, 96

        ...

        add     rsp, 96
        pop     rbp
        ret
```

Стековый фрейм



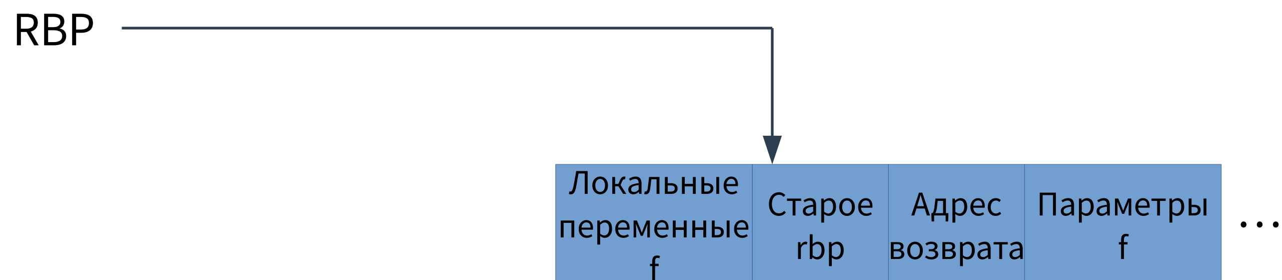

```
f:      push    rbp
        mov     rbp, rsp
        sub     rsp, 96

        ...

        add     rsp, 96
        pop     rbp
        ret
```

Стековый фрейм



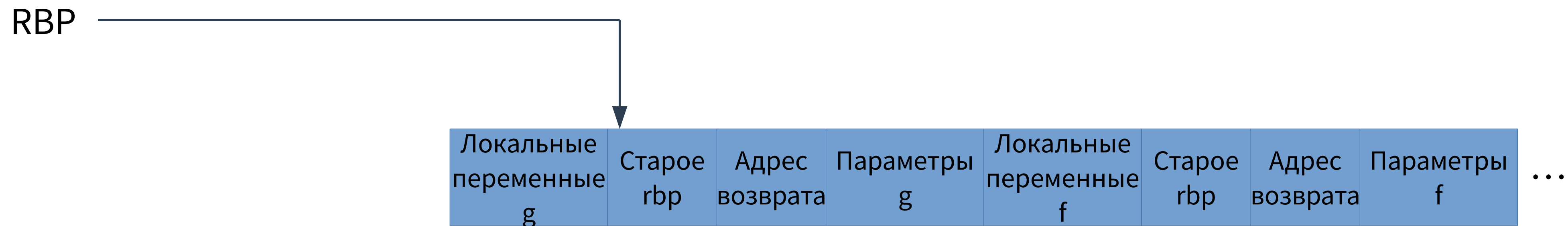

```
f:      push    rbp
        mov     rbp, rsp
        sub     rsp, 96

        ...

        add     rsp, 96
        pop     rbp
        ret
```

Стековый фрейм

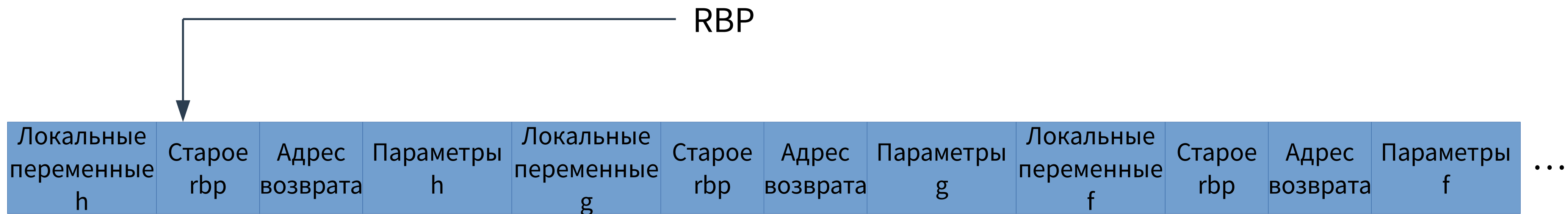



```
f:      push    rbp
        mov     rbp, rsp
        sub     rsp, 96

        ; [rbp – X] – локальные переменные
        ; [rbp + X] – параметры

        add     rsp, 96
        pop     rbp
        ret
```

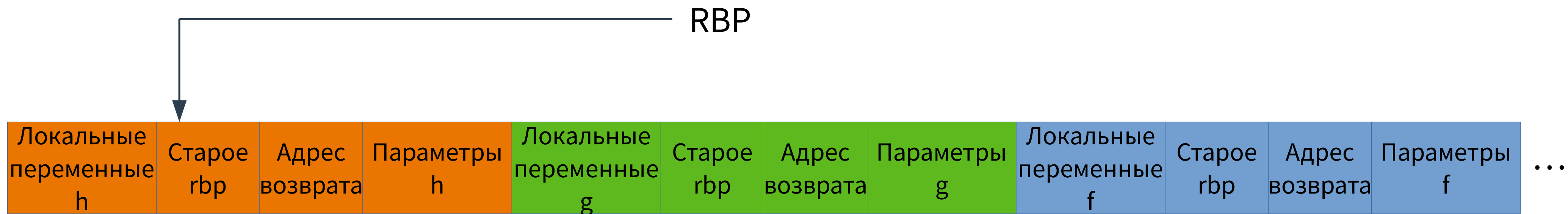
Стековый фрейм



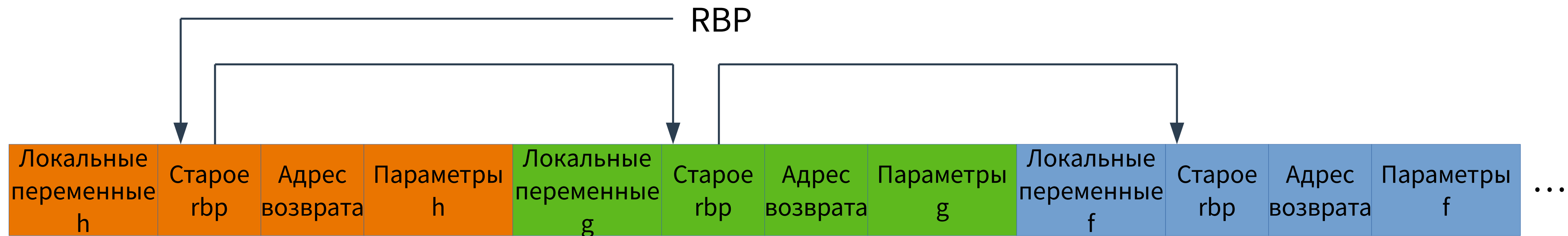
Стековый фрейм



Стековый фрейм



Стековый фрейм



Стековый фрейм

- Могут использоваться отладчиками/профиляторами как средство раскрутки стека.
- В GCC генерация стековых фреймов управляется опцией
  - `-f[no-]omit-frame-pointer`
- Создание стековых фреймов отключено по умолчанию в GCC в 64-битном режиме.
  - Экономит один регистр
  - Стек всё равно можно раскрутить имея отладочную информацию

Стековый фрейм

- Некоторые инструменты (`perf`) требуют специального ключика, который говорит, как раскручивать стек.
  - `--call-graph dwarf`
  - `--call-graph fp`
- Наши программы на ассемблере, не будут создавать стековые фреймы и не будут иметь соответствующей отладочной информации — обычные отладчики для них не смогут показать стек вызовов.