Машинный уровень 3: Процедуры

Основы информатики.
Компьютерные основы программирования

goo.gl/X7evF

На основе CMU 15-213/18-243:
Introduction to Computer Systems

goo.gl/Q7vgWw

Лекция 6, 12 марта, 2018

Лектор:

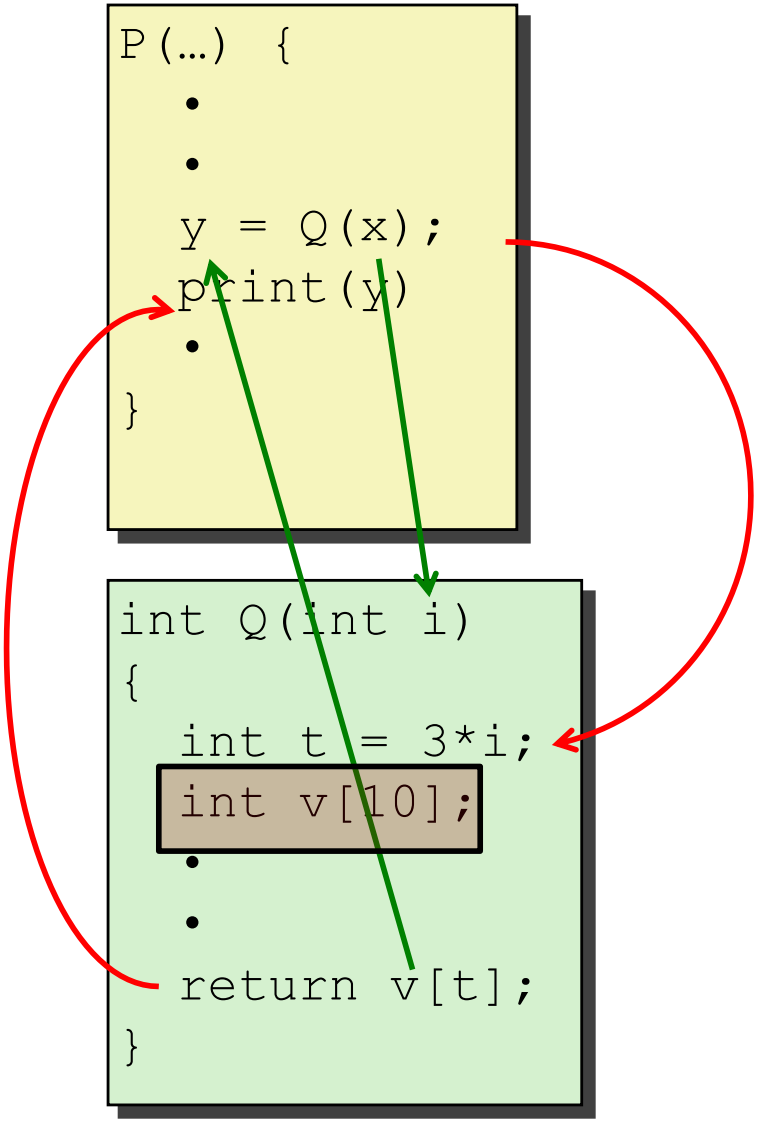
Дмитрий Северов, кафедра информатики 608 КПМ

dseverov@mail.mipt.ru

cs.mipt.ru/wp/?page_id=346

Механизмы процедур

- **Передача управления**
  - к началу кода процедуры
  - обратно в точку вызова
- **Передача данных**
  - Аргументы процедуры
  - Возвращаемое значение
- **Управление памятью**
  - Выделение на время выполнения
  - Высвобождение перед возвратом
- **Все механизмы реализованы машинными командами**
- **Реализация процедур в x86-64 задействует только необходимые механизмы**

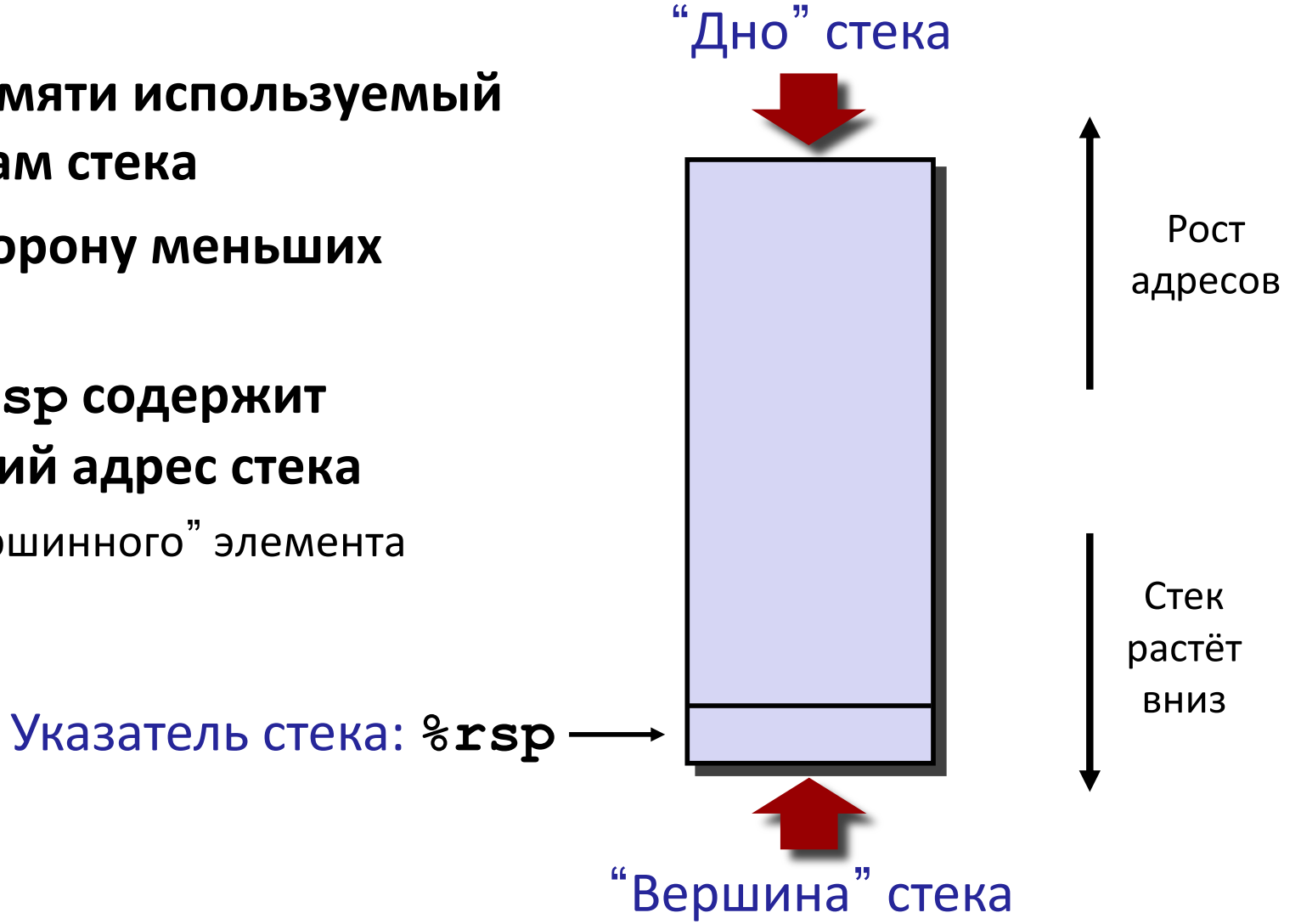


Машинный уровень 3: Процедуры

- Процедуры x86-64
  - Структура стека
  - Соглашения вызова процедур
    - Передача управления
    - Передача данных
    - Управление локальными данными
  - Иллюстрация рекурсии

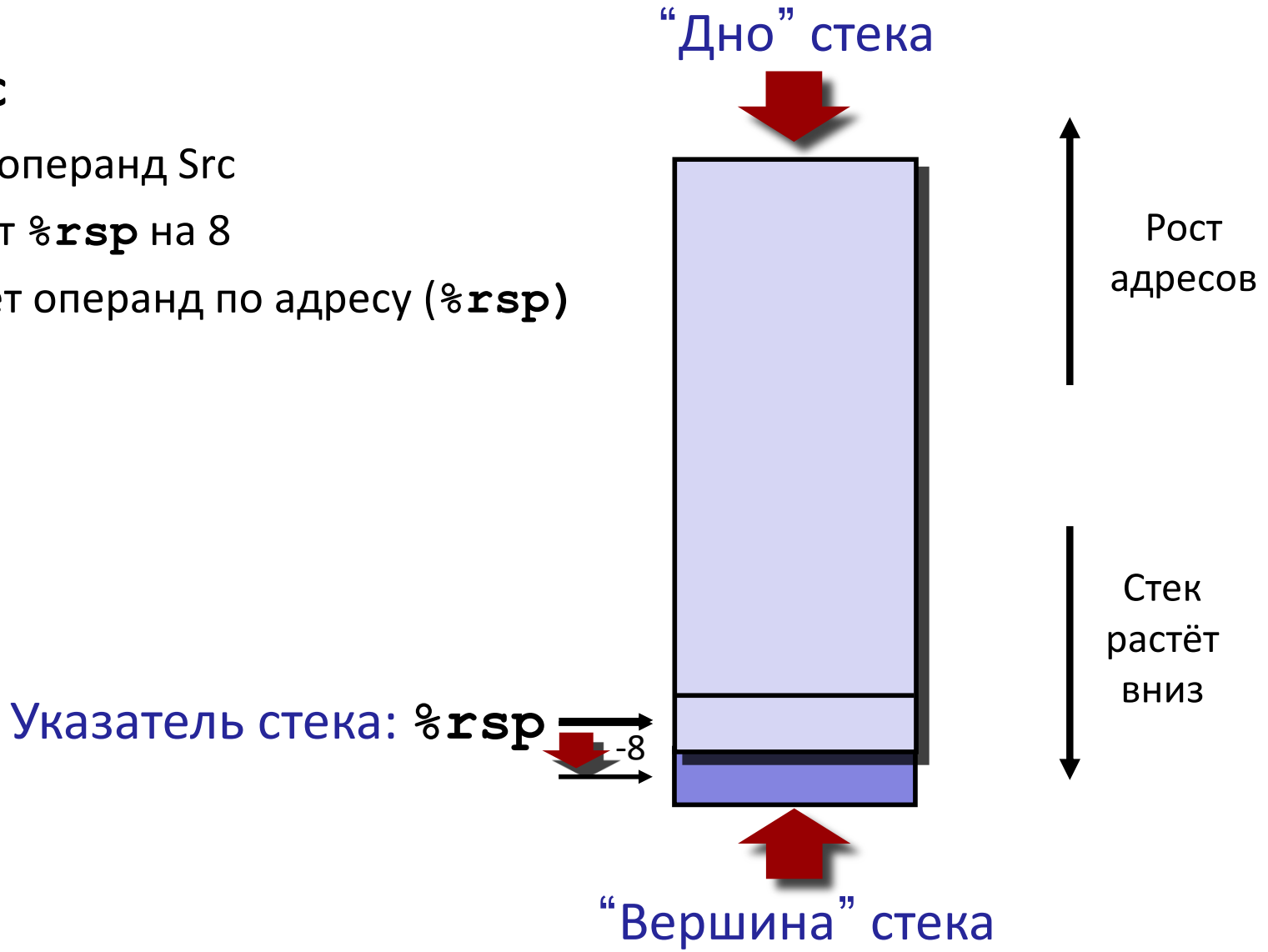
Стек x86-64

- **Участок памяти используемый по правилам стека**
- **Растёт в сторону меньших адресов**
- **Регистр `%rsp` содержит наименьший адрес стека**
  - адрес “вершинного” элемента



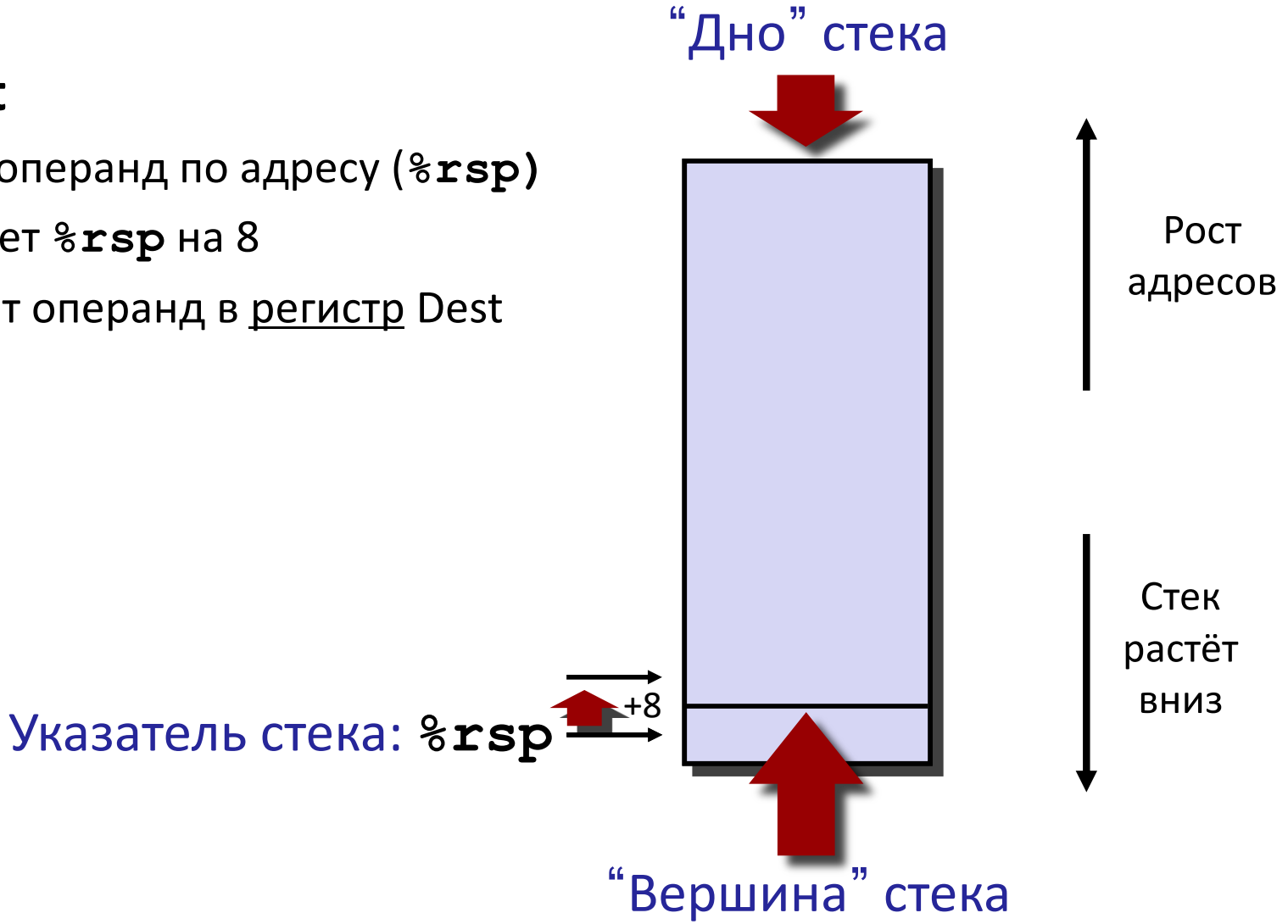
Стек x86-64: Вталкивание

- **`pushq Src`**
  - Выбирает операнд Src
  - Уменьшает **`%rsp`** на 8
  - Записывает операнд по адресу (**`%rsp`**)



Стек x86-64: Выталкивание

- **`popq` Dest**
  - Выбирает операнд по адресу (`%rsp`)
  - Увеличивает `%rsp` на 8
  - Записывает операнд в регистр Dest



Машинный уровень 3: Процедуры

- Процедуры x86-64
  - Структура стека
  - Соглашения вызова процедур
    - Передача управления
    - Передача данных
    - Управление локальными данными
  - Иллюстрация рекурсии

Пример кода

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx              # Сохранить %rbx
  400541: mov    %rdx,%rbx         # Сохранить dest
  400544: callq  400550 <mult2>    # mult2(x,y)
  400549: mov    %rax,(%rbx)       # Сохранить по *dest
  40054c: pop    %rbx              # Восстановить %rbx
  40054d: retq                     # Вернуть
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax        # a
  400553:  imul   %rsi,%rax        # a * b
  400557:  retq                    # вернуть
```

Поток управления процедуры

- **Стек используется при вызове процедур и возврате из них**
- **Вызов процедуры: `call label`**
  - Вталкивает адрес возврата в стек
  - Переходит к **`label`**
- **Адрес возврата:**
  - Адрес команды следующей за сразу за командой вызова
  - Дизассемблированный пример

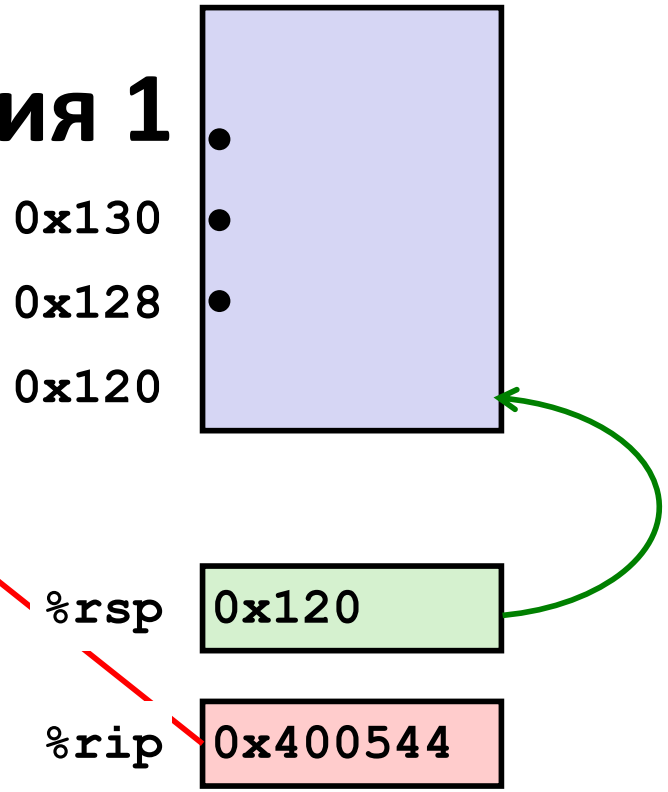
```
400544: callq  400550 <mult2>    # mult2(x,y)
400549: mov    %rax,(%rbx)       # Сохранить по *dest
```

  - Адрес возврата = **`0x400544`**
- **Возврат из процедуры: `ret`**
  - Выталкивает адрес возврата из стека
  - Переходит по адресу возврата

Пример потока управления 1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

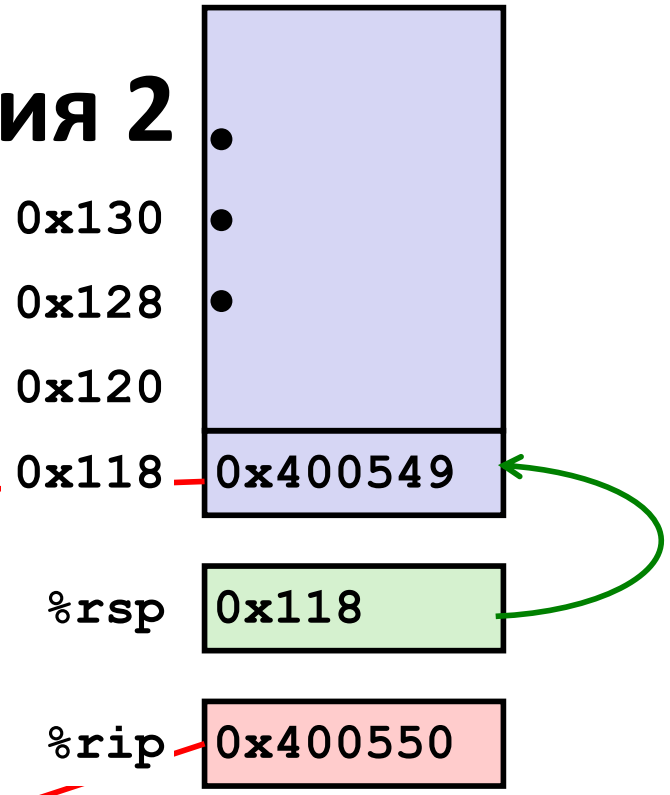
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Пример потока управления 2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

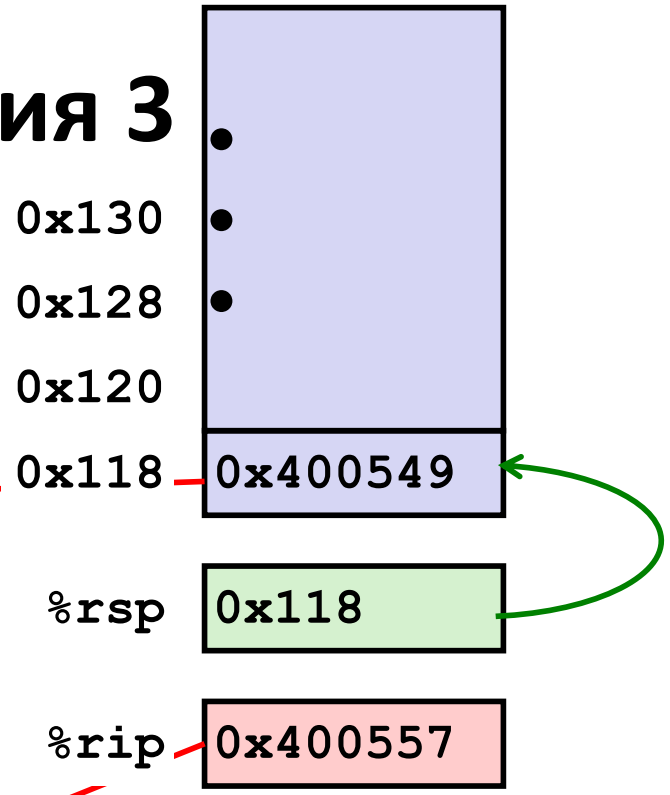
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Пример потока управления 3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

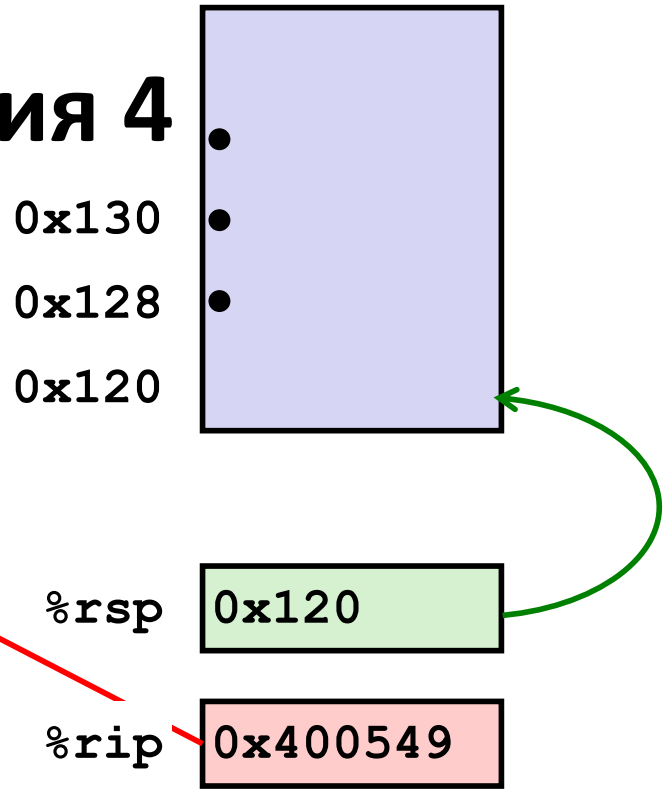
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Пример потока управления 4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



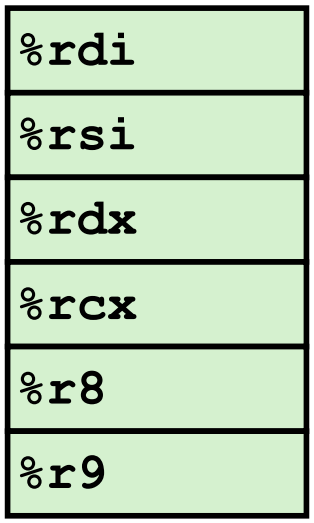
Машинный уровень 3: Процедуры

- Процедуры x86-64
  - Структура стека
  - Соглашения вызова процедур
    - Передача управления
    - Передача данных
    - Управление локальными данными
  - Иллюстрация рекурсии

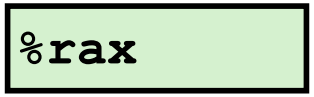
Передача данных в процедуру и обратно

Регистры

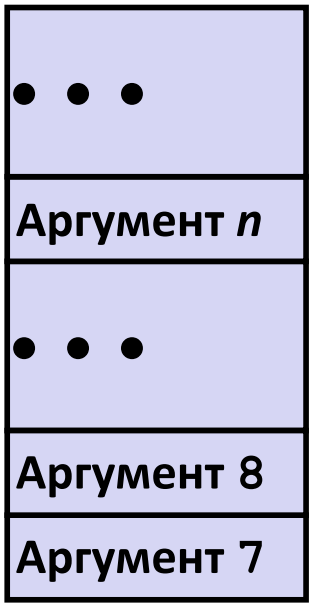
- Первые 6 аргументов



- Возвращаемый результат



Стек



- Стек занимается только по необходимости

Примеры передачи данных

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x в %rdi, y в %rsi, dest в %rdx
  •••
  400541: mov    %rdx,%rbx          # сохранить dest
  400544: callq  400550 <mult2>     # вызвать mult2(x,y)
  # t в %rax
  400549: mov    %rax,(%rbx)        # сохранить по *dest
  •••
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax         # a
  400553:  imul   %rsi,%rax         # a * b
  # s in %rax
  400557:  retq                     # вернуть
```

Машинный уровень 3: Процедуры

- Процедуры x86-64
  - Структура стека
  - Соглашения вызова процедур
    - Передача управления
    - Передача данных
    - Управление локальными данными
  - Иллюстрация рекурсии

Языки использующие стек

- **Языки поддерживающие рекурсию**
  - например, Си, Паскаль, Ява
  - код процедуры может быть “реентерабельным”
    - пригоден для следующего вызова, до завершения предыдущего
  - есть отдельное место для хранения состояния каждого вызова
    - Аргументы
    - Локальные переменные
    - Адрес возврата
- **Вызов процедур следует стековой дисциплине**
  - Состояние процедуры востребовано ограниченное время
    - от момента вызова до момента возврата
  - Возврат из вызванной всегда раньше возврата из вызвавшей
- **Стек наполняется кадрами (frames)**
  - Стековый кадр – состояние одного запуска процедуры

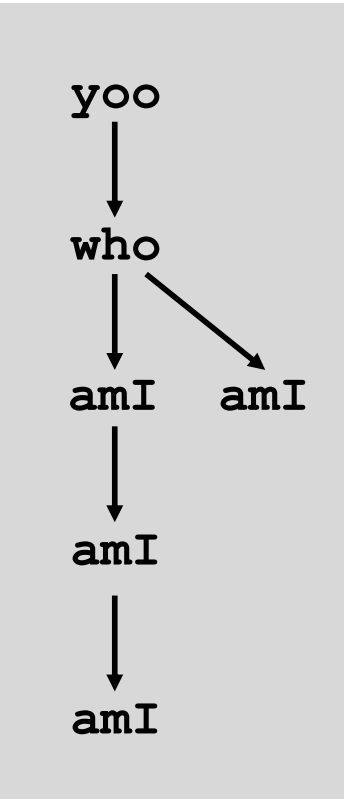
Пример нескольких вызовов

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

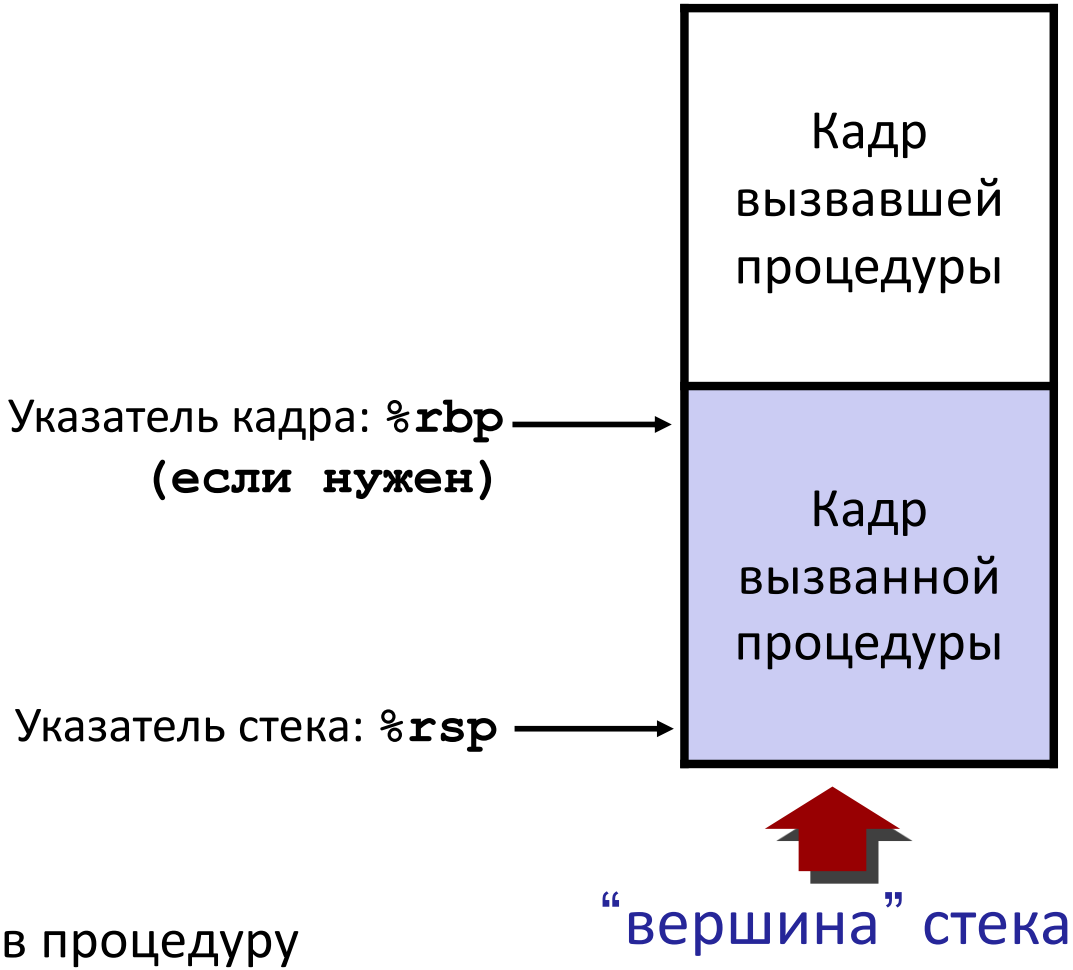
Примерная схема вызовов



amI() - рекурсивная процедура

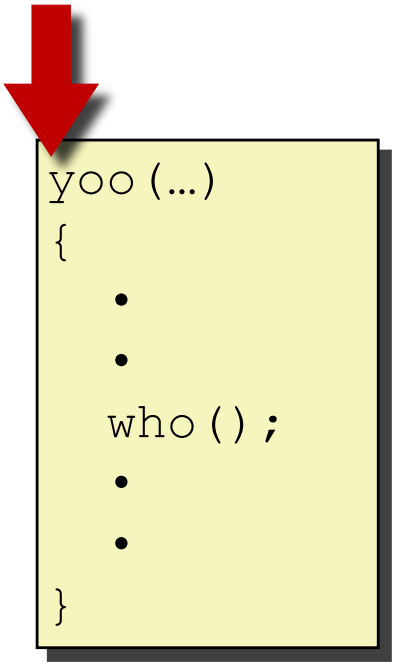
Стековые кадры

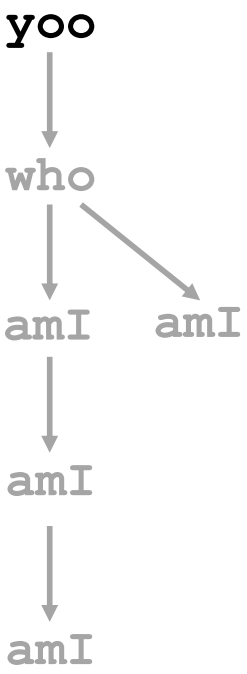
- **Содержимое**
  - Локальные переменные
  - Информация для возврата
  - Временное пространство

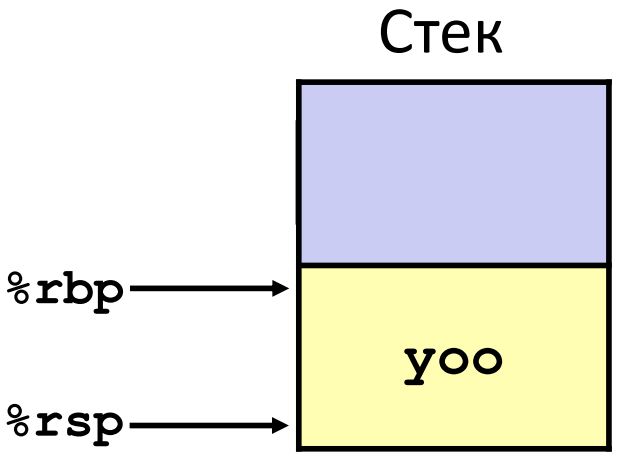


- **Управление**
  - Место занимается при входе в процедуру
    - Код “пролога”
  - Освобождается при выходе
    - Код “эпилога”

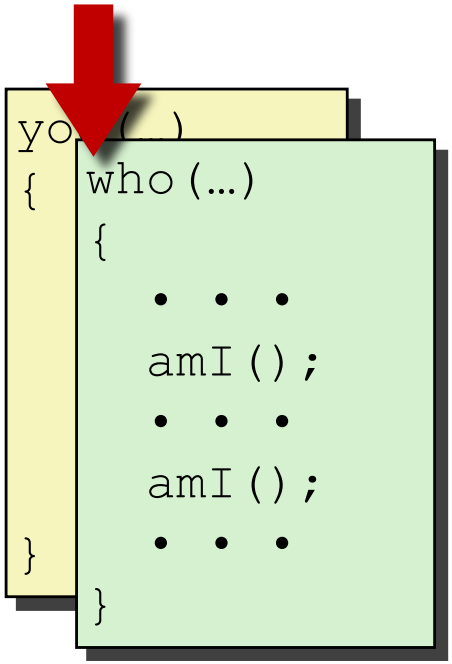
Пример

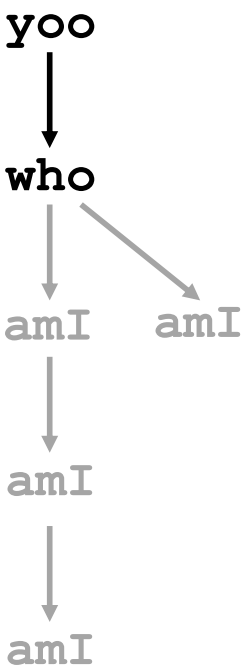


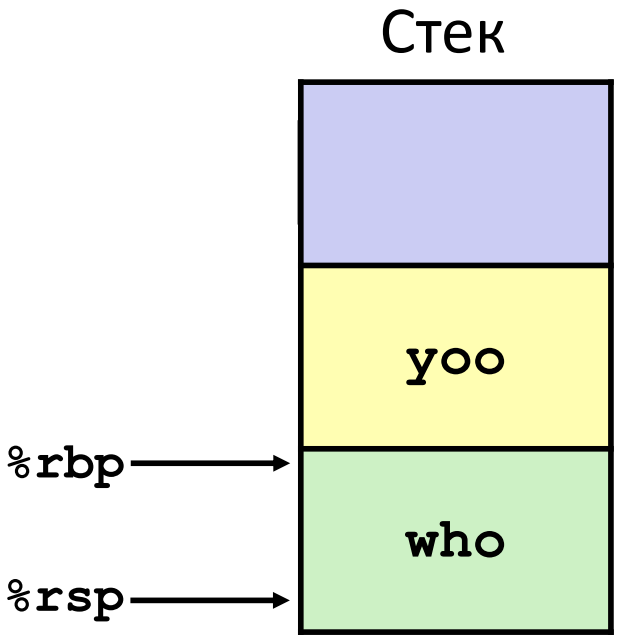




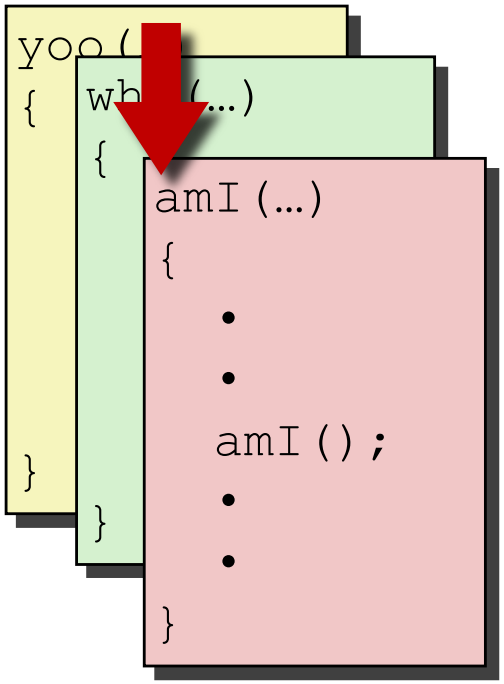
Пример

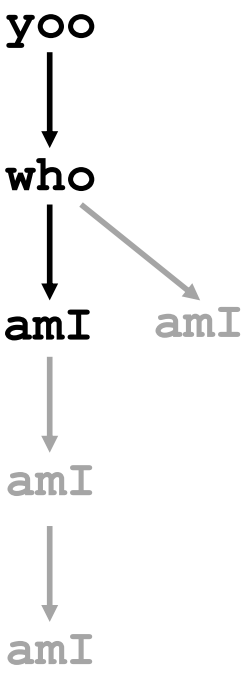


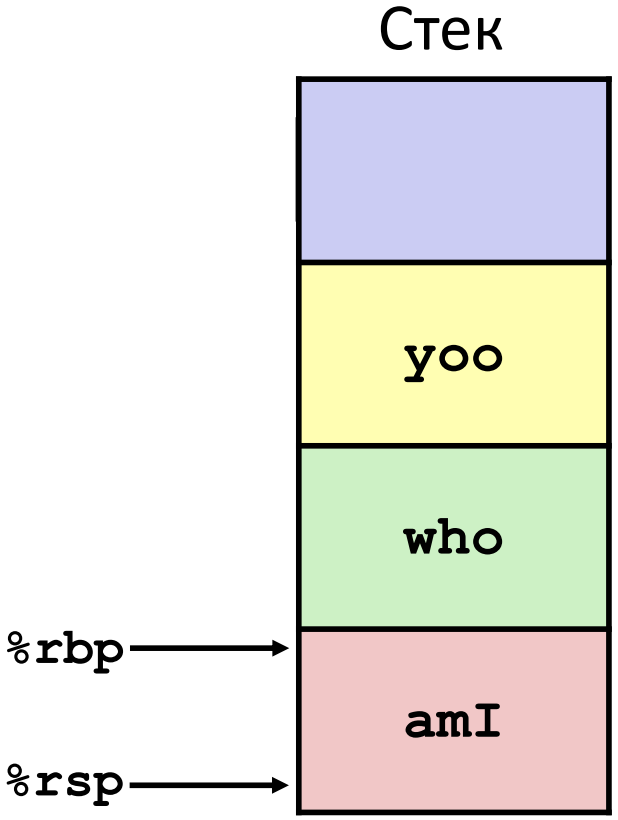




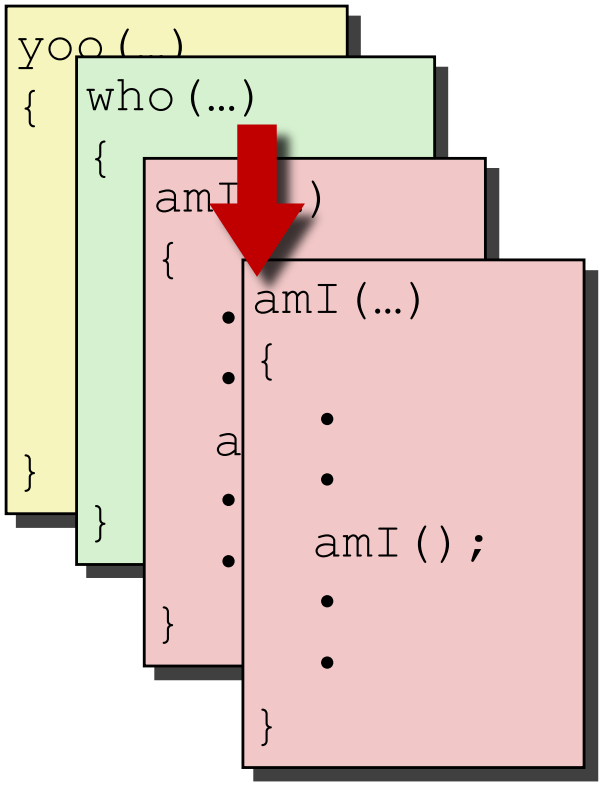
Пример

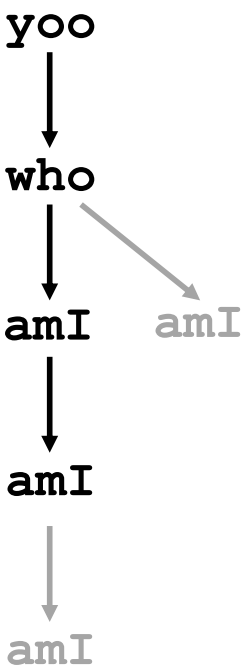


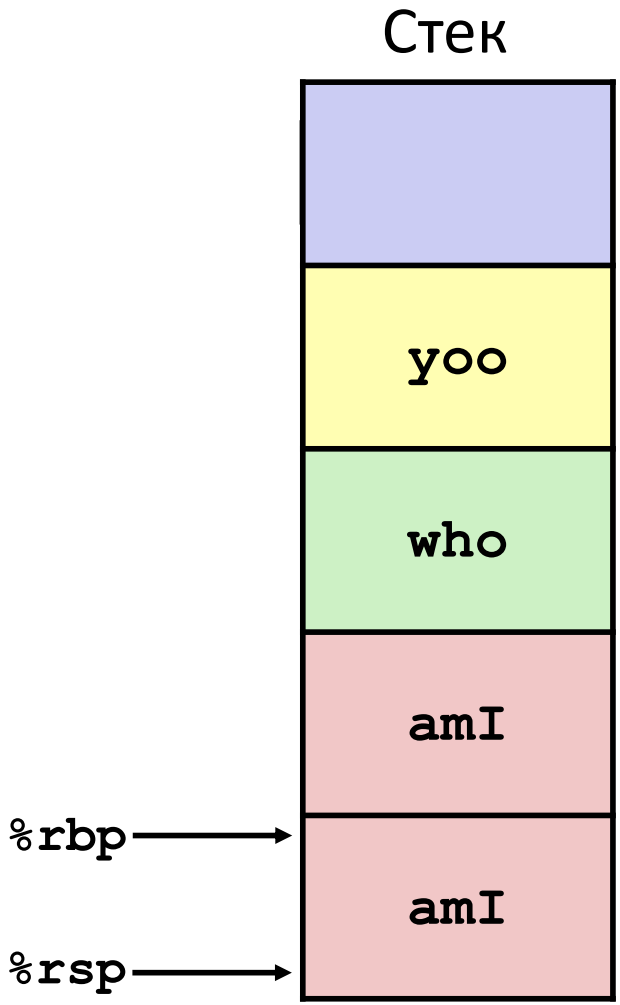




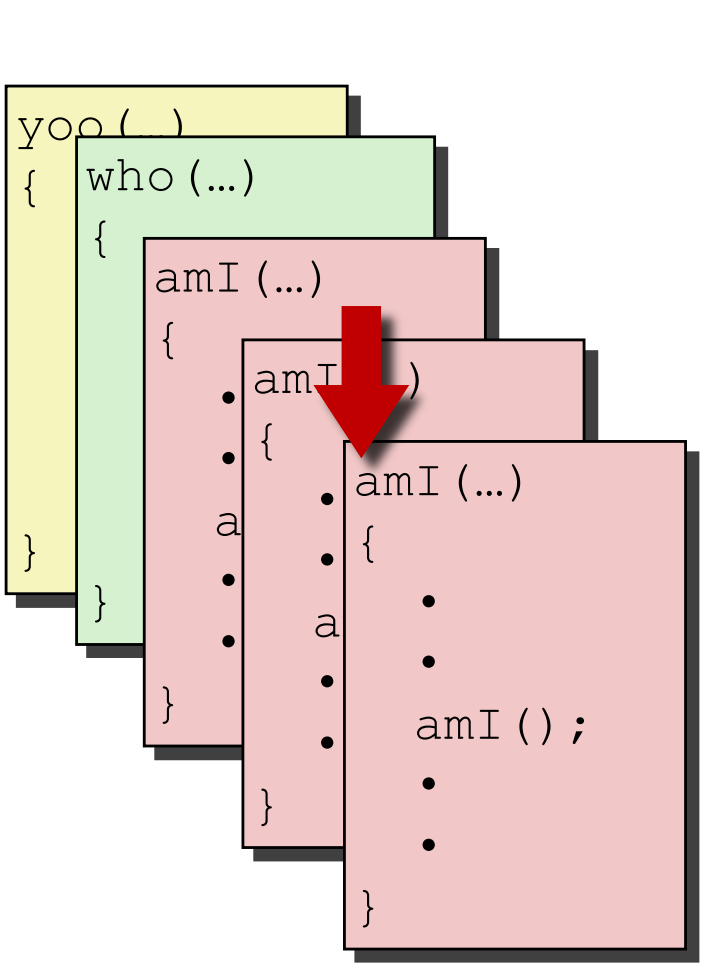
Пример

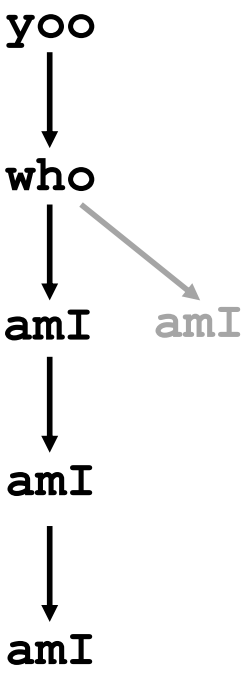

yoo( )
{
who(…)
{
amI( )
{
amI(…)
{
amI();
}

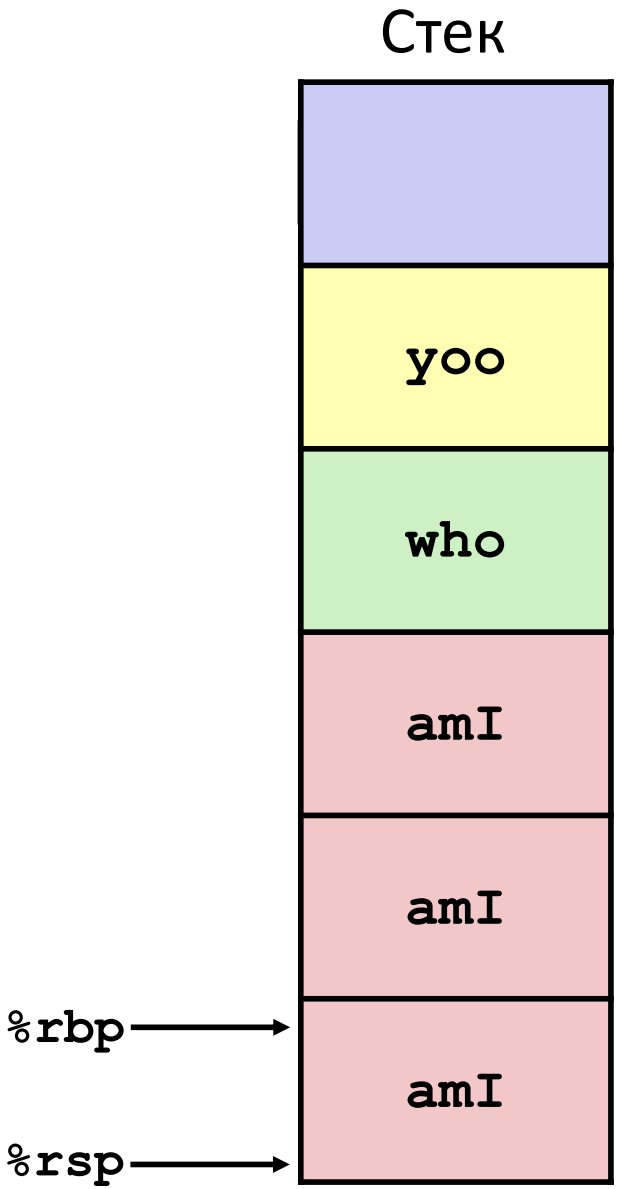

yoo
who
amI
amI
amI
amI


Стек
yoo
who
amI
amI
%rbp
%rsp

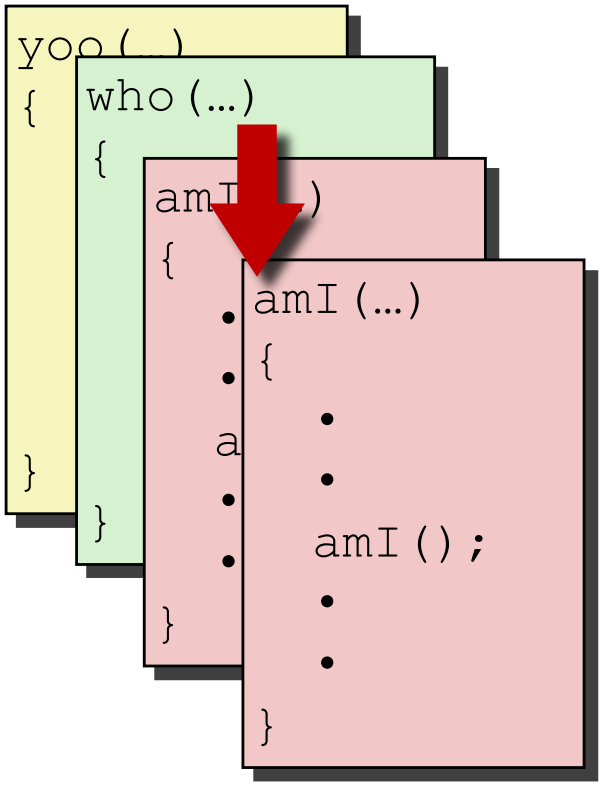
Пример

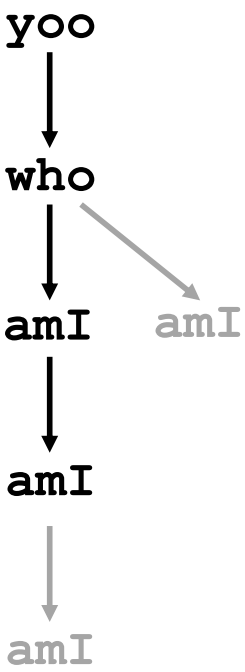

who(…)
{
amI(…)
{
amI(…)
{
amI(…)
{
amI();
}

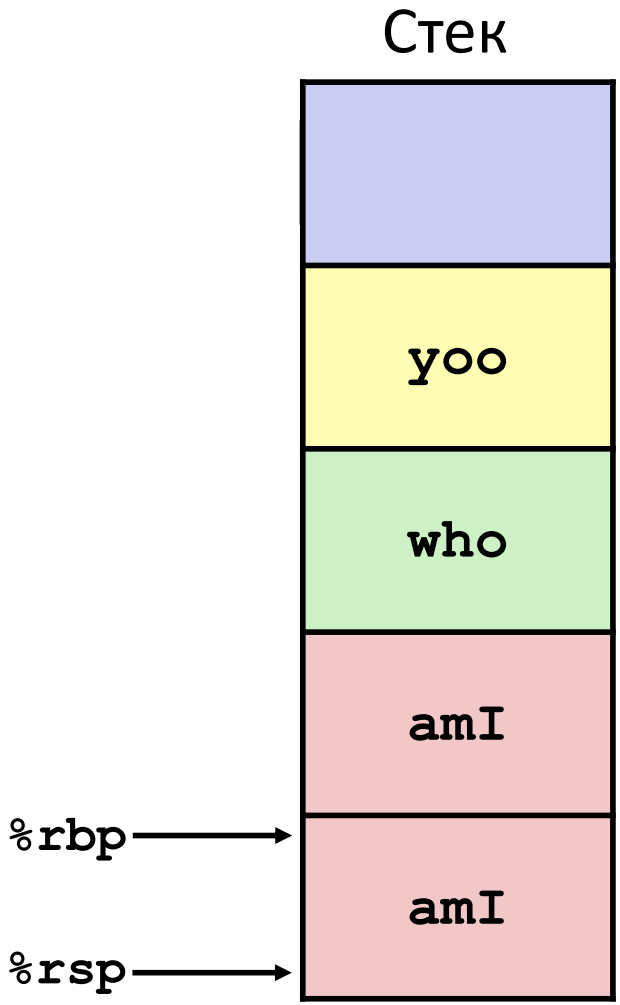

yoo
who
amI
amI
amI
amI


Стек
yoo
who
amI
amI
amI
%rbp
%rsp

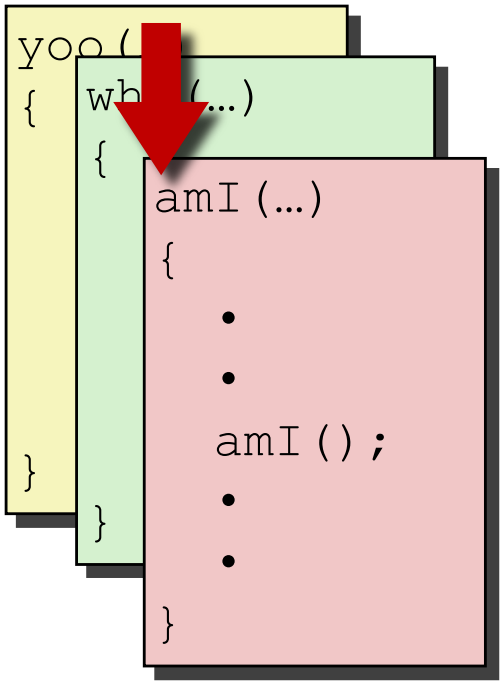
Пример

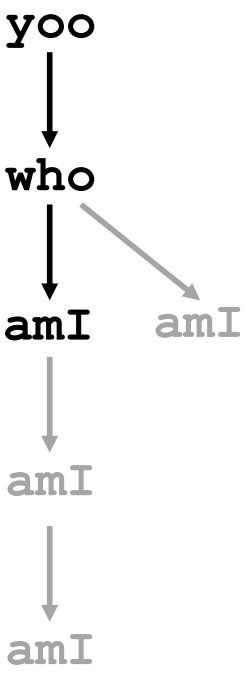

yoo( )
{
who(…)
{
amI( )
{
amI(…)
{
amI();
}

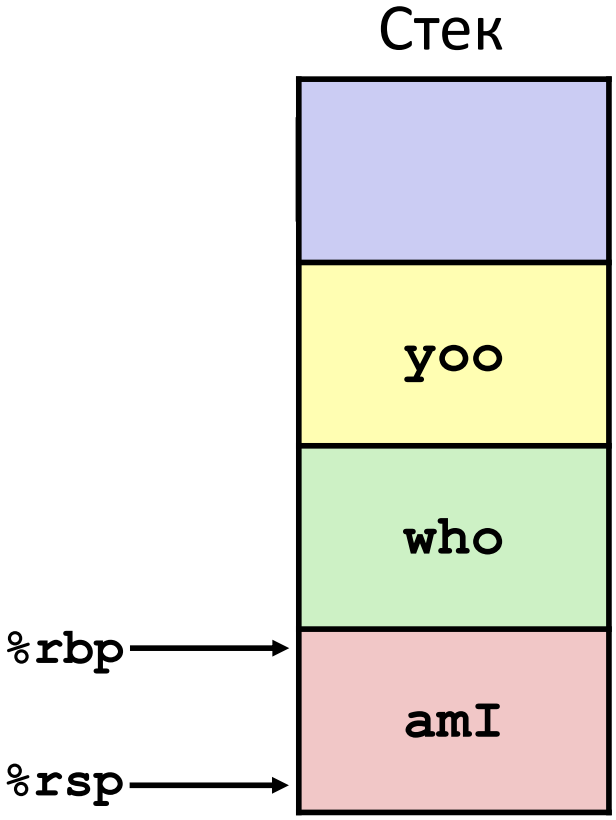

yoo
who
amI
amI
amI
amI


Стек
yoo
who
amI
amI
%rbp
%rsp

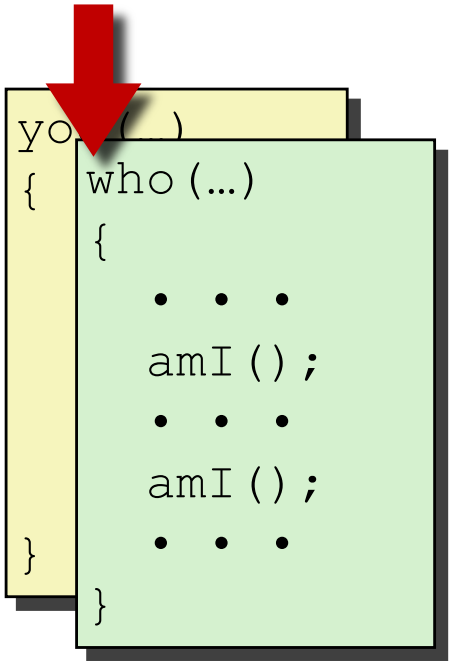
Пример

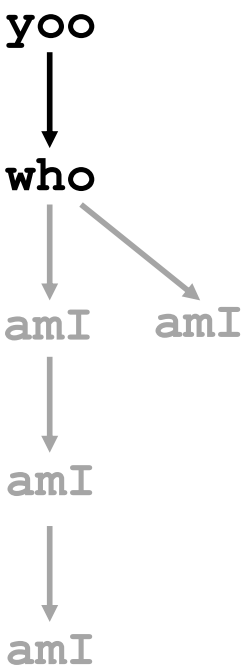


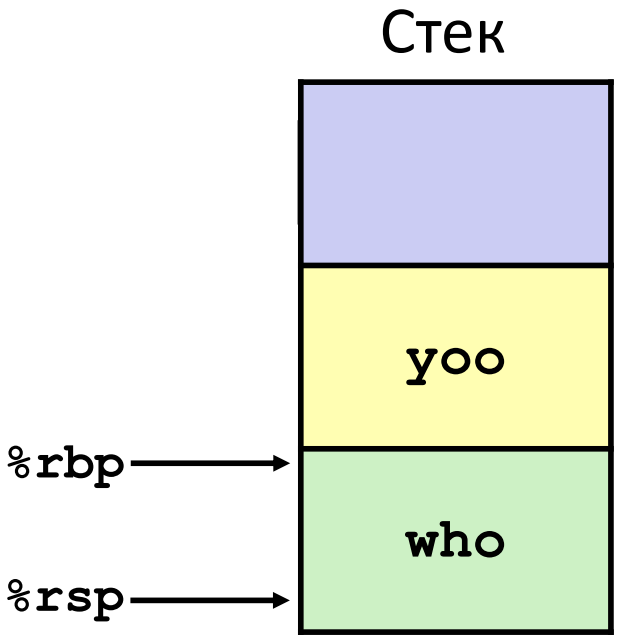




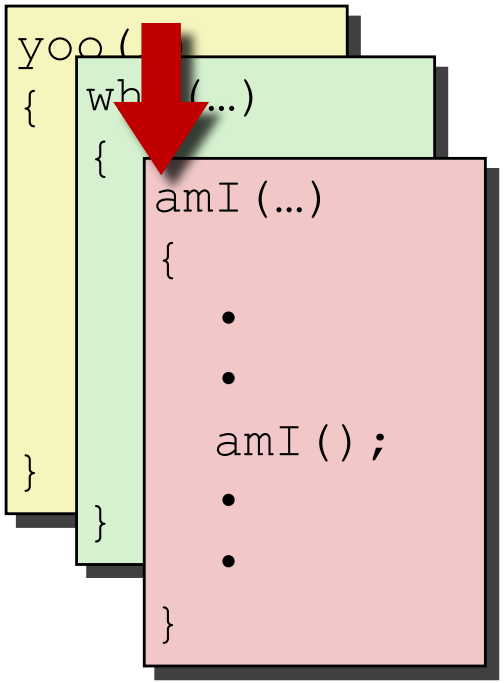
Пример

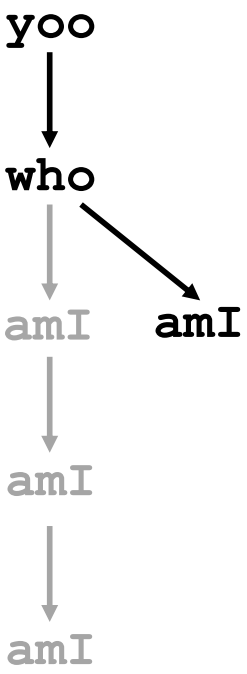


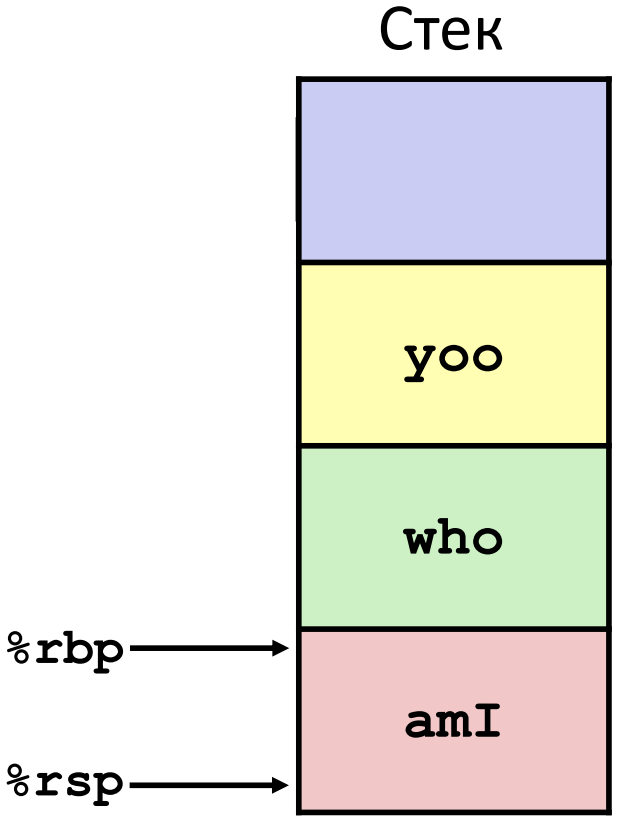




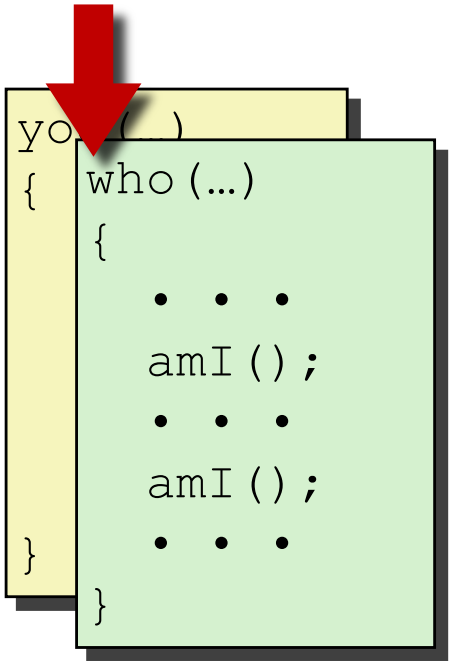
Пример

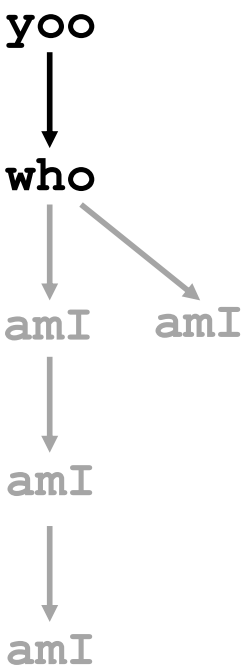


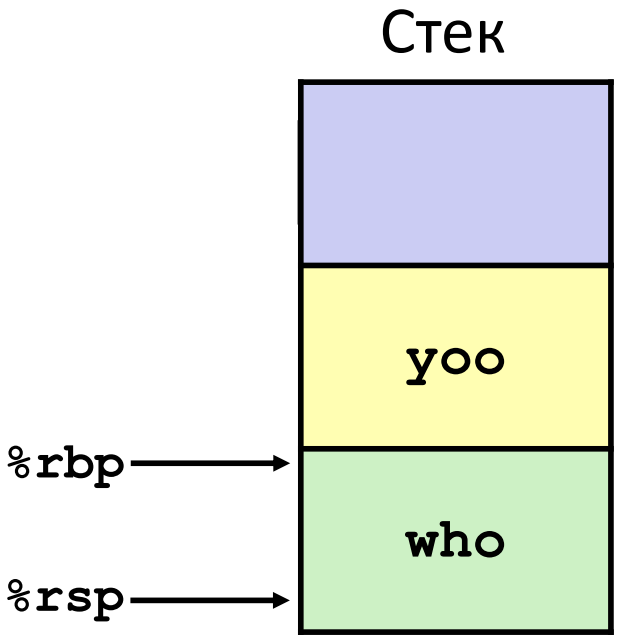




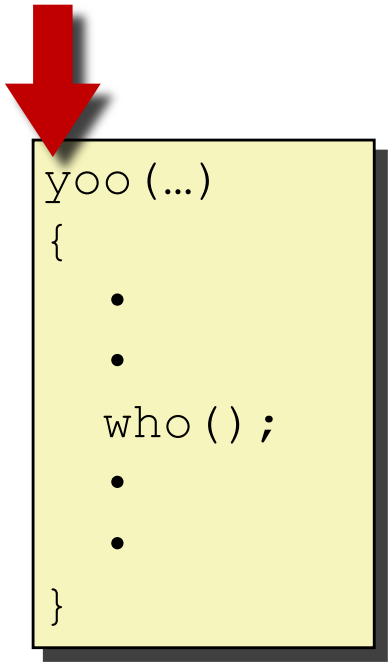
Пример

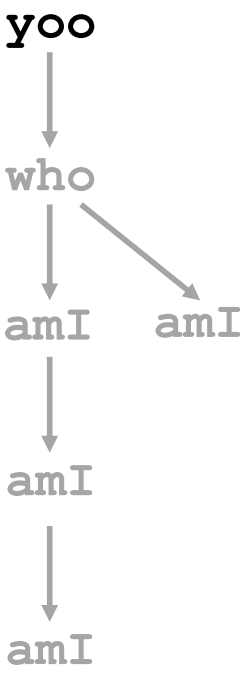


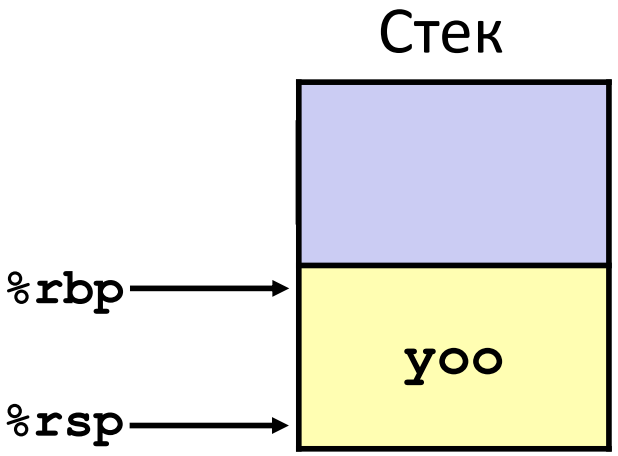




Пример

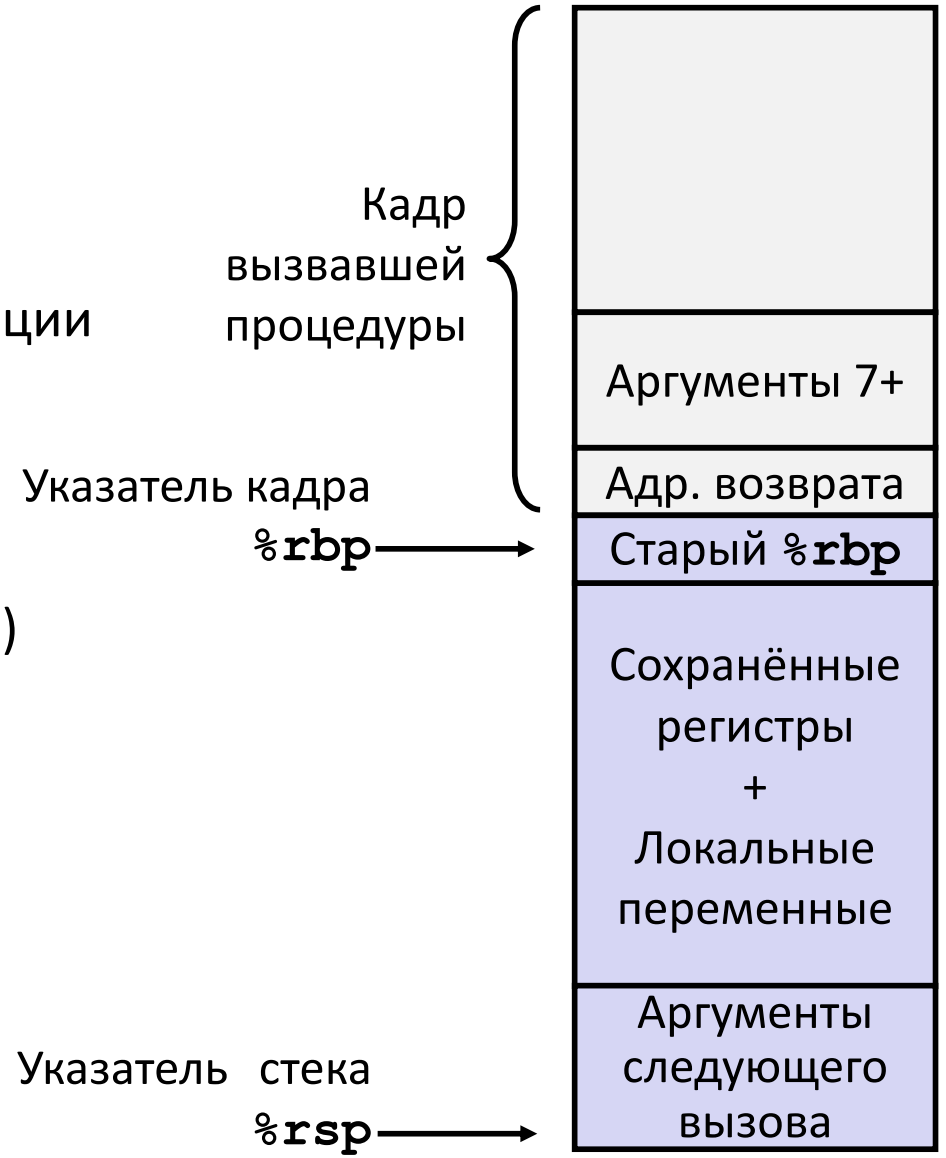






Стековый кадр x86-64/Linux

- **Текущий кадр (от вершины ко дну стека)**
  - Аргументы 7+ следующего вызова: параметры вызываемой позже функции
  - Локальные переменные (если не хватает регистров)
  - Сохранённые значения регистров
  - Старое значение `%rbp` (если нужен)
- **Кадр вызвавшей процедуры**
  - Адрес возврата
    - Втолкнут в стек командой `call`
  - Аргументы 7+ вызова текущей



Пример: `incr`

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

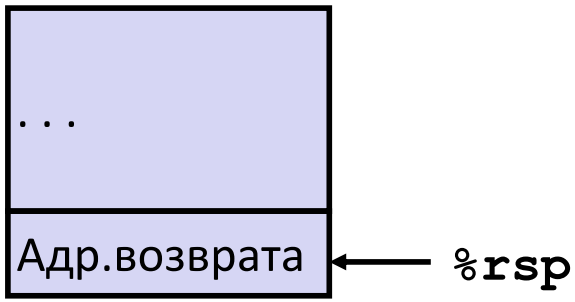
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Регистр	Использование
`%rdi`	Аргумент **`p`**
`%rsi`	Аргумент **`val`**, **`y`**
`%rax`	**`x`**, возвращаемое

Пример: вызов `incr 1`

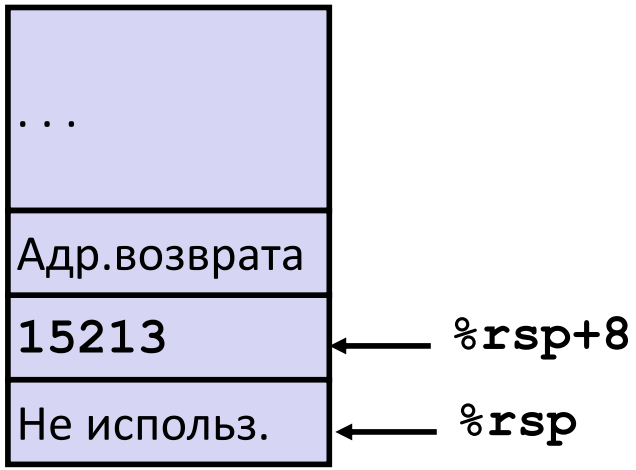
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Исходная структура стека




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Новая структура стека

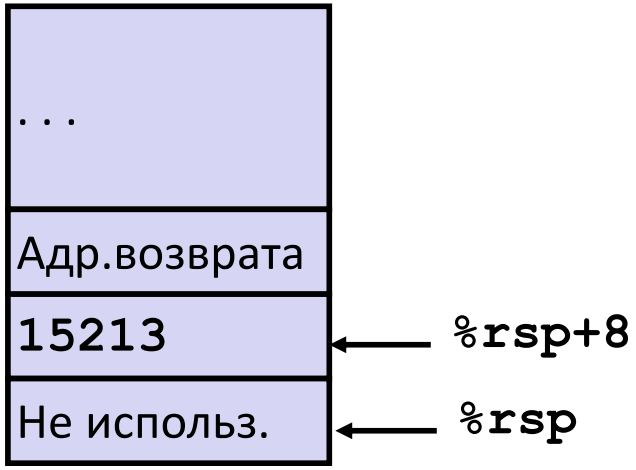


Пример: вызов `incr 2`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Структура стека



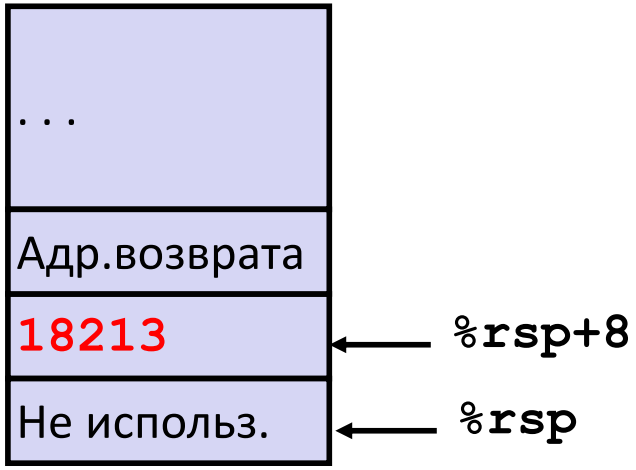
Регистр	Использование
%rdi	&v1
%rsi	3000

Пример: вызов `incr 3`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Структура стека

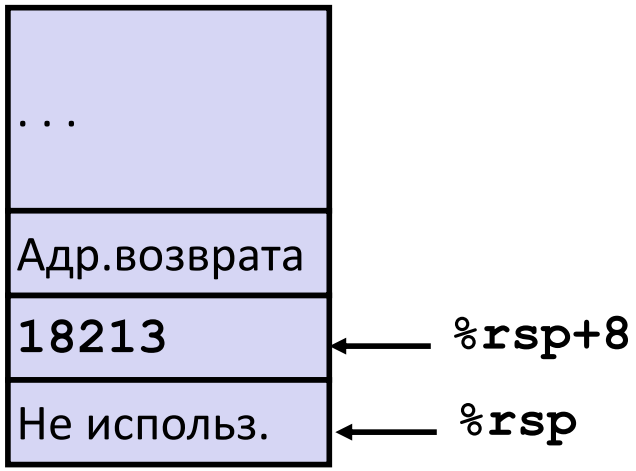


Регистр	Использование
%rdi	&v1
%rsi	3000

Пример: вызов `incr 4`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

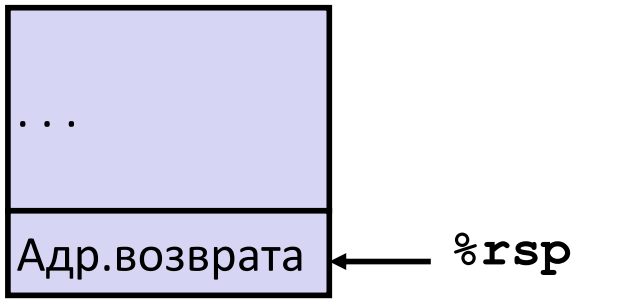
Структура стека




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Регистр	Использование
%rax	Возвращаемое

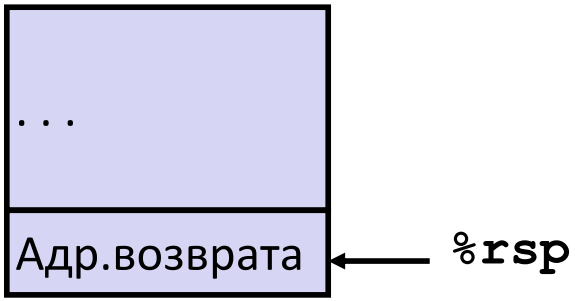
Структура изменённого стека



Пример: вызов `incr 5`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

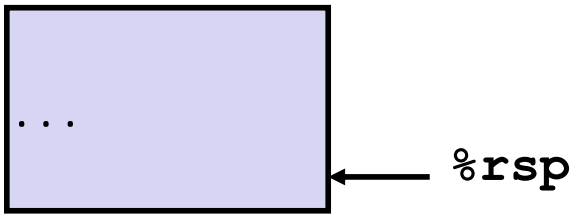
Структура изменённого стека




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Регистр	Использование
%rax	Возвращаемое

Финальная структура стека



Соглашения о сохранении регистров

- **При вызове процедурой `yoo` процедуры `who`:**
  - **`yoo`** - вызывающая
  - **`who`** - вызываемая
- **Можно ли в регистрах временно хранить данные?**

```
yoo:
    • • •
    movl $15213, %rdx
    call who
    addl %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

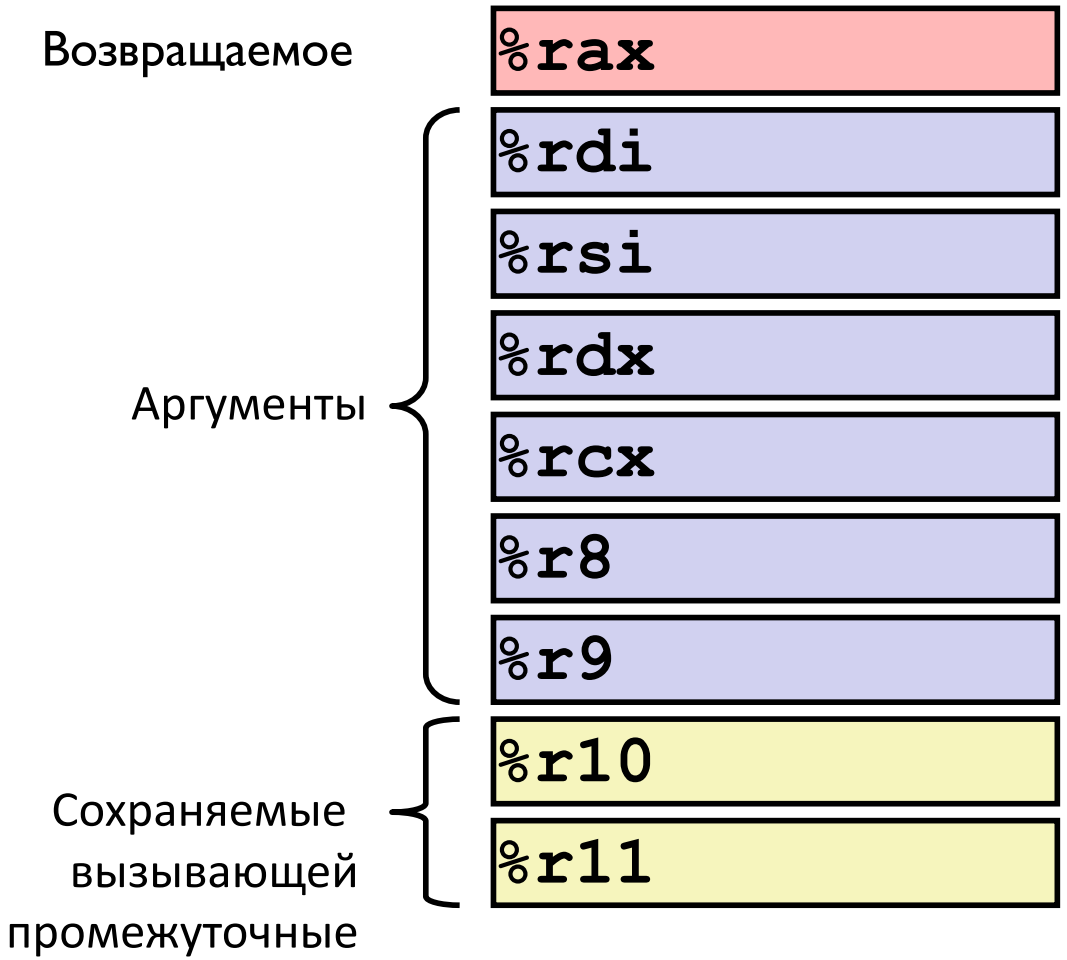
  - Данные **`yoo`** для регистра **`%rdx`** затираются данными **`who`**
  - Требуется согласованное использование

Соглашения о сохранении регистров

- **При вызове процедурой `yoo` процедуры `who`:**
  - `yoo` - вызывающая
  - `who` - вызываемая
- **Можно ли в регистрах временно хранить данные?**
- **Варианты соглашений**
  - “Сохраняет вызывающая”
    - Вызывающая сохраняет временные значения в своём кадре перед вызовом
  - “Сохраняет вызываемая”
    - Вызываемая сохраняет временные значения в своём кадре перед началом использования
    - Вызываемая восстанавливает временные значения из своего кадра перед возвратом

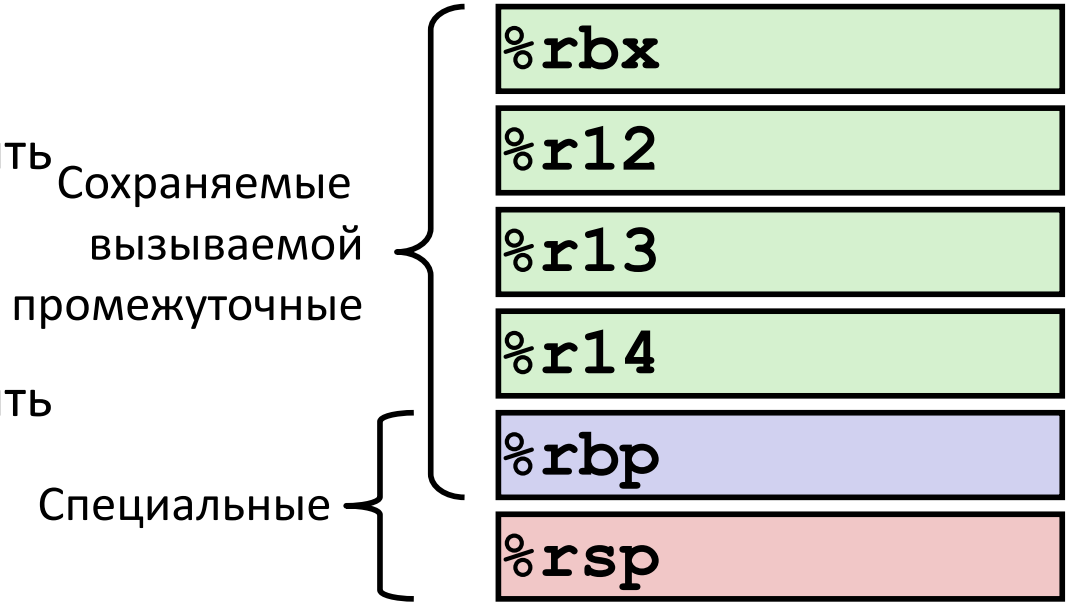
Использование регистров в x86-64 Linux 1

- `%rax`
  - Возвращаемое значение
  - Сохраняет вызывающая
  - Вызываемая может менять
- `%rdi`, …, `%r9`
  - Arguments
  - Сохраняет вызывающая
  - Вызываемая может менять
- `%r10`, `%r11`
  - Сохраняет вызывающая
  - Вызываемая может менять



Использование регистров в x86-64 Linux 2

- **`%rbx`, `%r12`, `%r13`, `%r14`**
  - Вызываемая должна сохранить и восстановить
- **`%rbp`**
  - Вызываемая должна сохранить и восстановить
  - Может использоваться как указатель кадра
- **`%rsp`**
  - Вызываемая должна сохранить и восстановить специальным образом перед возвратом

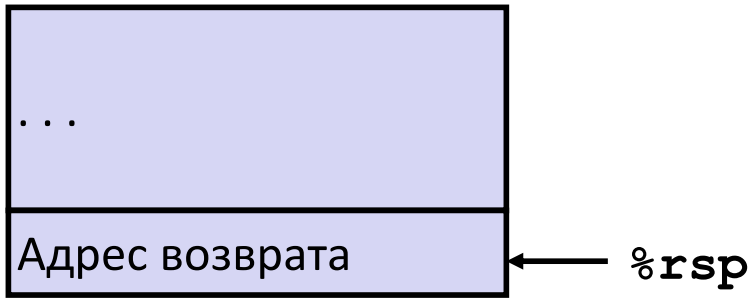


Пример сохранения вызываемой 1

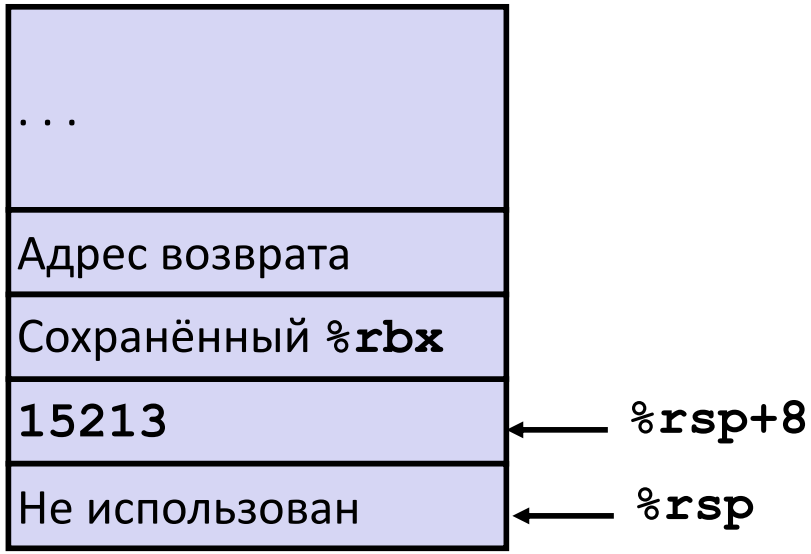
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Исходная структура стека



Новая структура стека

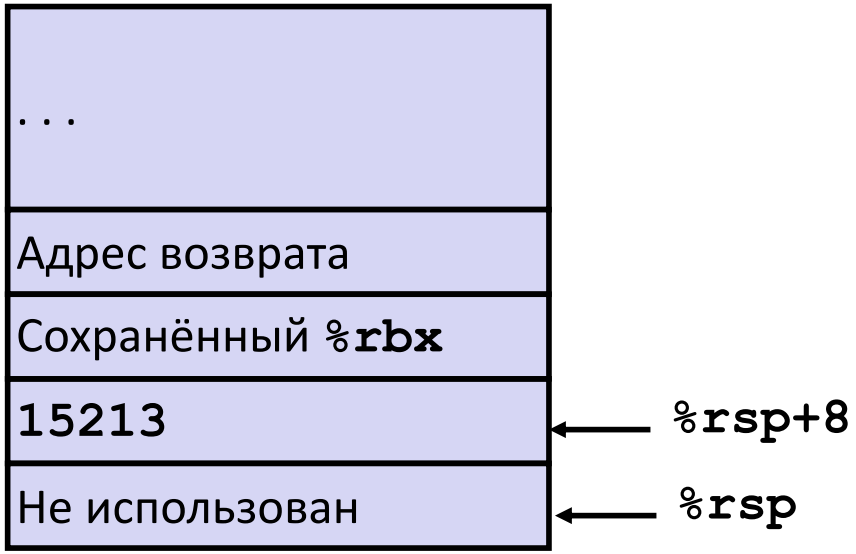


Пример сохранения вызываемой 2

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Новая структура стека



Структура стека перед возвратом

Машинный уровень 3: Процедуры

- Процедуры x86-64
  - Структура стека
  - Соглашения вызова процедур
    - Передача управления
    - Передача данных
    - Управление локальными данными
  - Иллюстрация рекурсии

Рекурсивная функция

```
/* Рекурсивный счётчик бит */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Нерекурсивный возврат

```
/* Рекурсивный счётчик бит */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl    $0, %eax
   testq   %rdi, %rdi
   je      .L6
   pushq   %rbx
   movq    %rdi, %rbx
   andl    $1, %ebx
   shrq    %rdi
   call    pcount_r
   addq    %rbx, %rax
   popq    %rbx
.L6:
   rep; ret
```

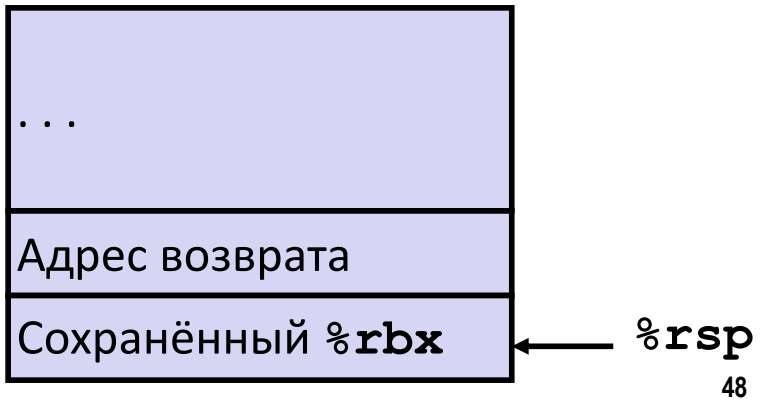
Регистр	Использование	Категория
%rdi	x	Аргумент
%rax	Возвращаемое	Возвращаемое

Сохранение регистров

```
/* Рекурсивный счётчик бит */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

Регистр	Использование	Категория
%rdi	x	Аргумент



Подготовка вызова

```
/* Рекурсивный счётчик бит */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Регистр	Использование	Категория
%rdi	x >> 1	Рекурсивный арг.
%rbx	x & 1	Сохр.вызываемая

Рекурсивный вызов

```
/* Рекурсивный счётчик бит */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Регистр	Использование	Категория
%rbx	x & 1	Сохр.вызываемая
%rax	Рекурсивное возвращаемое	

Рекурсивный учёт результата

```
/* Рекурсивный счётчик бит */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

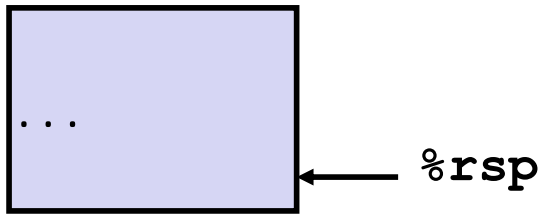
Регистр	Использование	Категория
%rbx	x & 1	Сохр.вызываемая
%rax	Возвращаемое	

Рекурсивный возврат

```
/* Рекурсивный счётчик бит */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

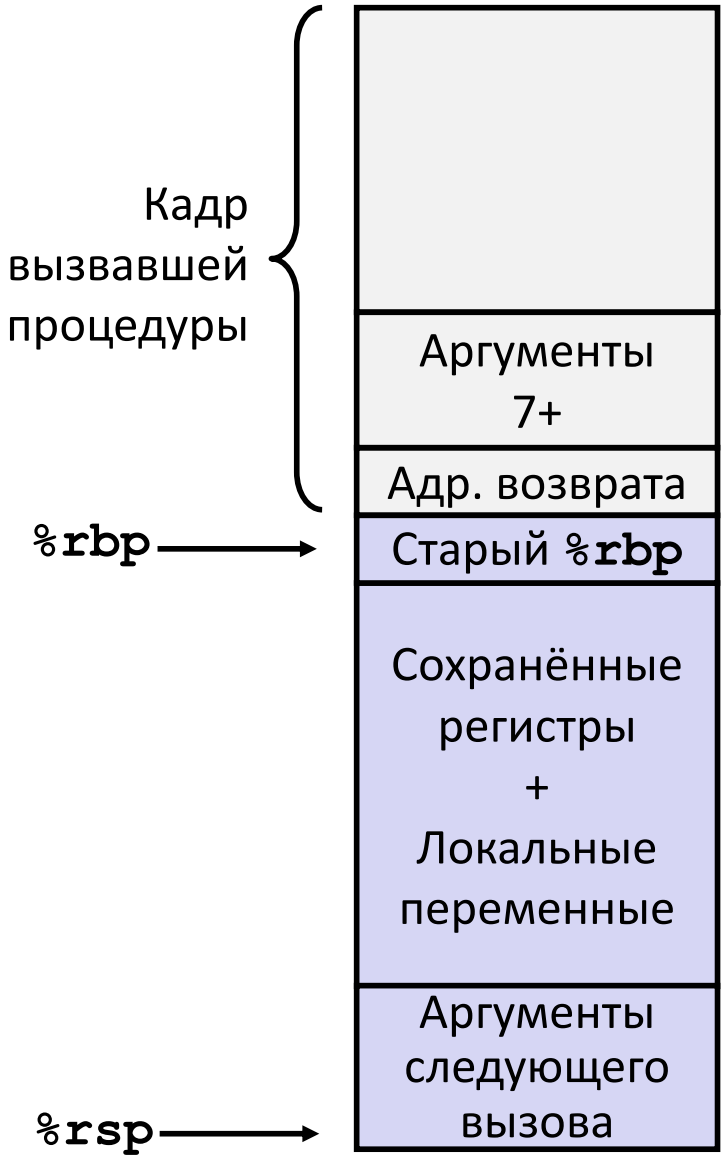
```
pcount_r:
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

Регистр	Использование	Категория
%rax	Возвращаемое	Возвращаемое



Важное о рекурсии

- **Организуется без специальных мер**
  - Стековый кадр – каждому вызову функции отдельное хранилище
    - Сохранённые регистры и локальные переменные
    - Сохранённый адрес возврата
  - Соглашение о сохранении регистров предотвращает повреждение данных одного процедурного вызова другим
    - Пока Си-код не сделает это явно
  - Схема вызовов/возвратов следует стековой дисциплине
    - Если P вызывает Q, то Q возвращает управление раньше, чем P
    - Последним вошёл, первым ушёл (LIFO)
- **Также работает для взаимной рекурсии**
  - P вызывает Q, а Q вызывает P

Сводка: процедуры x86-64

- **Важно!**
  - Стек – походящая структура данных для вызова процедур и возврата из них
    - если P вызывает Q, то возврат из Q раньше чем из P
- **(Взаимная) рекурсия реализуется обычными правилами вызова**
  - Значения безопасно хранятся в своём кадре и регистрах сохраняемых вызванной
  - Аргументы функции – на вершине стека
  - Результат возвращается в **`%rax`**
- **`Указатели – адреса значений`**
  - В стеке или среди глобальных переменных