第3章 程序的机器级表示
Machine-Level Programming III: Procedures

100076202： 计算机系统导论

III: 过程

III: Procedures

任课教师:

宿红毅 张艳 黎有琦 颜珂

原作者:

Randal E. Bryant and David R. O'Hallaron

Carnegie Mellon University

目标 Objectives

- push/pop和call/ret指令对的基本功能：Basic functionality of the pairs: push / pop and call / ret
- 学生应该能够识别栈的不同组件（返回地址、参数、保存的寄存器、局部变量）Students should be able to identify the different components of a stack (return address, arguments, saved registers, local variables)
- 解释被调用者和调用者保存寄存器的不同 Explain the difference between callee and caller save registers
- 解释栈如何允许函数被递归调用/重入 Explain how a stack permits functions to be called recursively / re-entrant

过程中的机制 Mechanisms in Procedures

- **传递控制 Passing control**
  - 进入过程代码的开始 To beginning of procedure code
  - 回到返回点 Back to return point
- **传递数据 Passing data**
  - 过程参数 Procedure arguments
  - 返回值 Return value
- **内存管理 Memory management**
  - 在过程执行期间分配内存 Allocate during procedure execution
  - 返回时释放内存 Deallocate upon return
- **所有机制由机器指令实现 Mechanisms all implemented with machine instructions**
- **x86-64的过程实现仅使用这些需要的机制 x86-64 implementation of a procedure uses only those mechanisms required**

```
P(…) {
  •
  •
  y = Q(x);
  print(y)
  •
}
```

```
int Q(int i)
{
  int t = 3*i;
  int v[10];
  •
  •
  return v[t];
}
```

Mechanisms in Procedures

P(…) {

机器指令实现该机制，但是具体选择由设计师确定。这些选择构成了**应用程序二进制接口（ABI）**。

Machine instructions implement the mechanisms, but the choices are determined by designers. These choices make up the **Application Binary Interface (ABI)**.

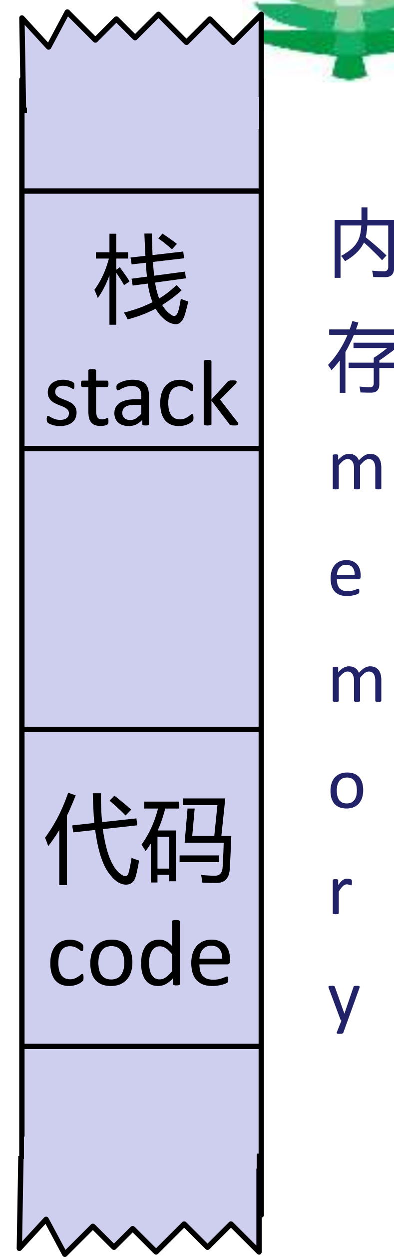
uses only those mechanisms required

议题

- 过程 Procedures
  - 栈结构 Stack Structure
  - 调用规则 Calling Conventions
    - 传递控制 Passing control
    - 传递数据 Passing data
    - 管理局部数据 Managing local data
  - 递归说明 Illustration of Recursion

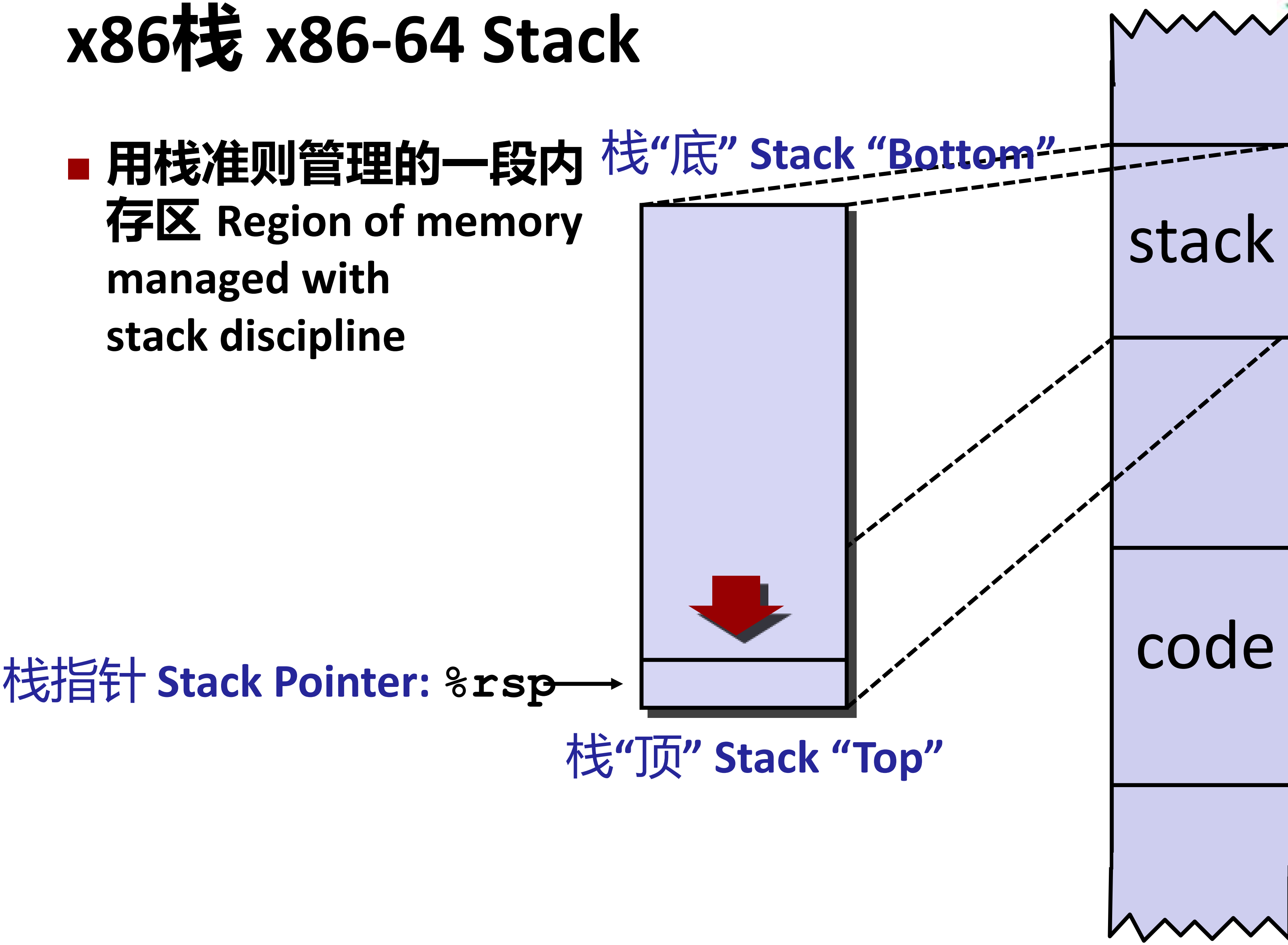
x86栈 x86-64 Stack

- **用栈准则管理的一段内存区 Region of memory managed with stack discipline**
  - 内存看成字节数组 Memory viewed as array of bytes.
  - 不同区域有不同用途 Different regions have different purposes.
  - （类似ABI，策略决策事情）(Like ABI, a policy decision)



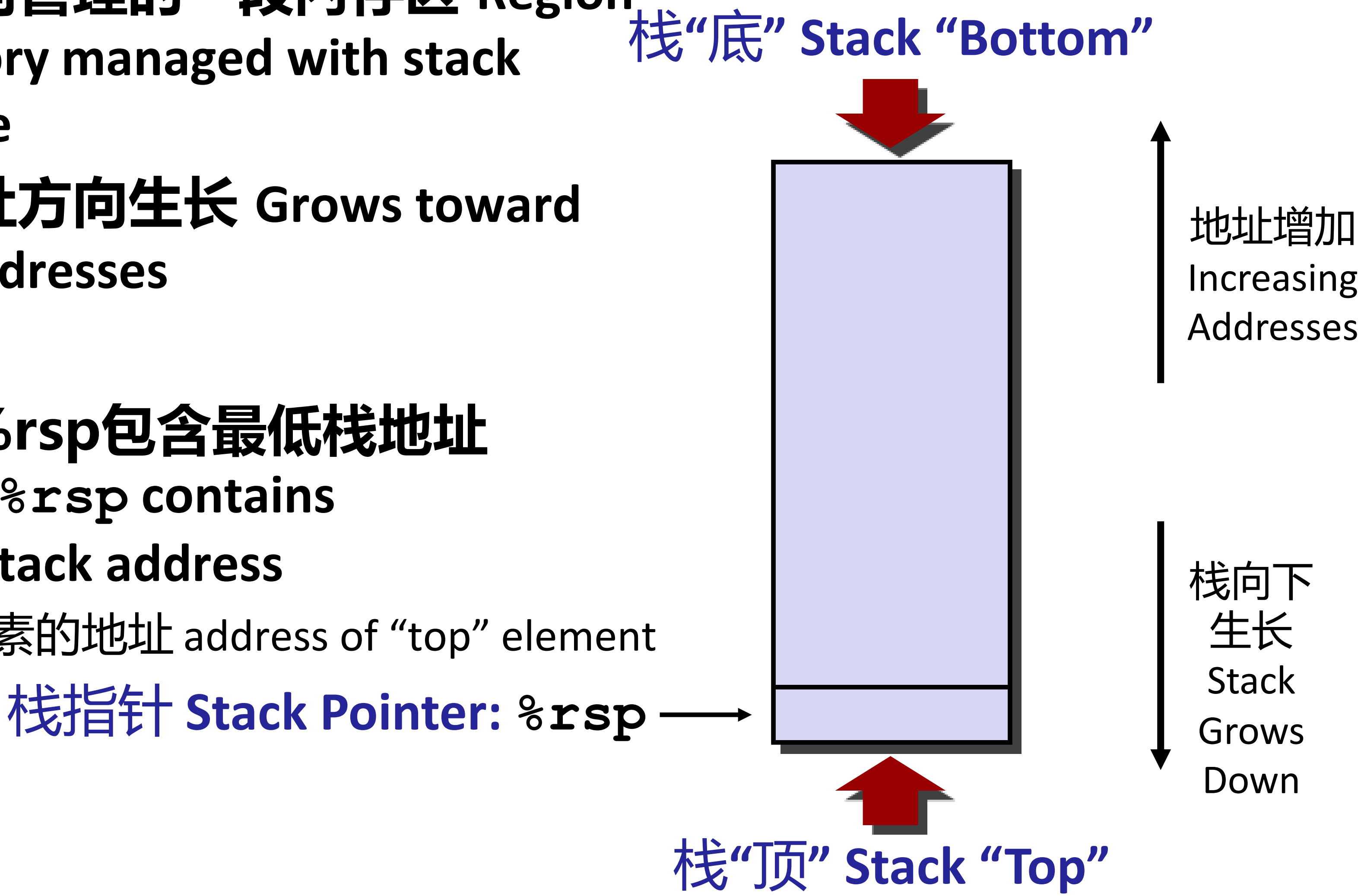
x86栈 x86-64 Stack

- 用栈准则管理的一段内存区 Region of memory managed with stack discipline



x86-64栈 x86-64 Stack

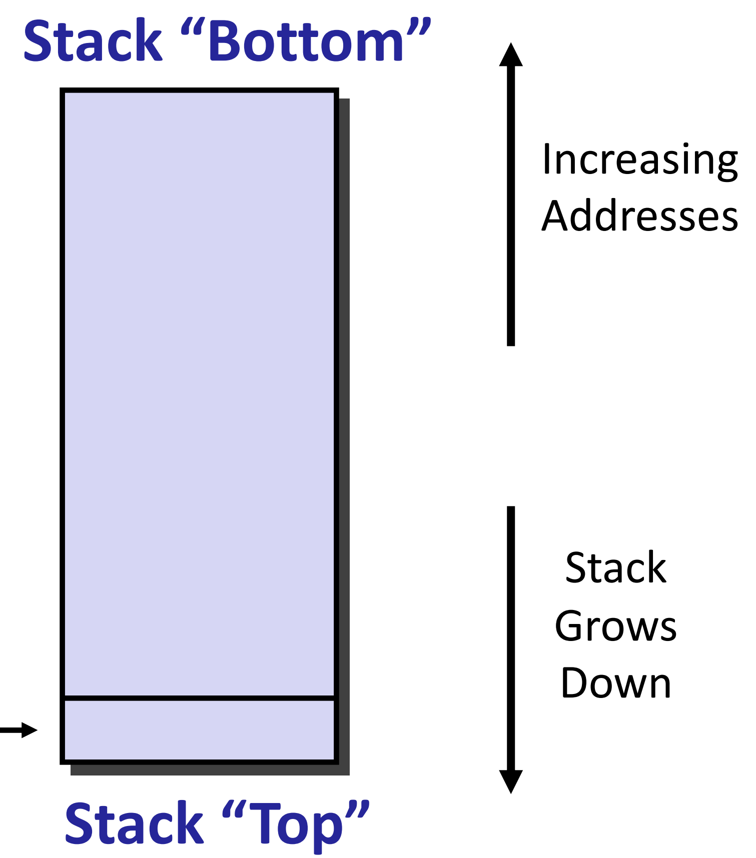
- **用栈准则管理的一段内存区 Region of memory managed with stack discipline**
- **向低地址方向生长 Grows toward lower addresses**
- **寄存器%rsp包含最低栈地址 Register `%rsp` contains lowest stack address**
  - 最顶元素的地址 address of “top” element



X86-64栈：压栈 x86-64 Stack: Push

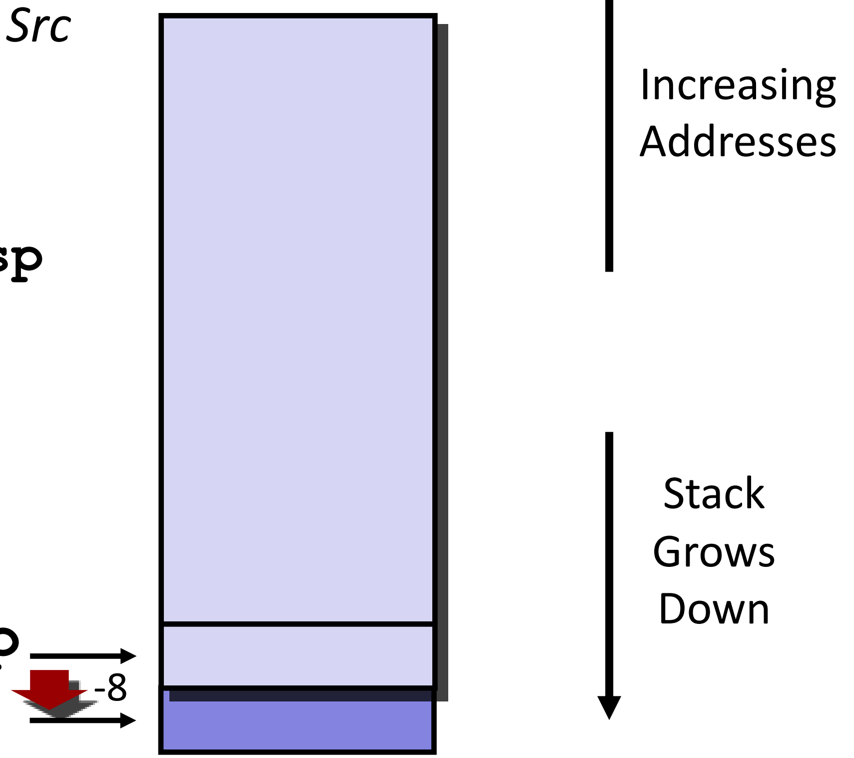
- **`pushq`** ***Src***
  - 获取Src处的操作数 Fetc val at *Src*
  - %rsp减8 Decrement **`%rsp`** by 8
  - 把操作数写入%rsp指定的地址
  - Write operand at address given by **`%rsp`**

Stack Pointer: **`%rsp`**



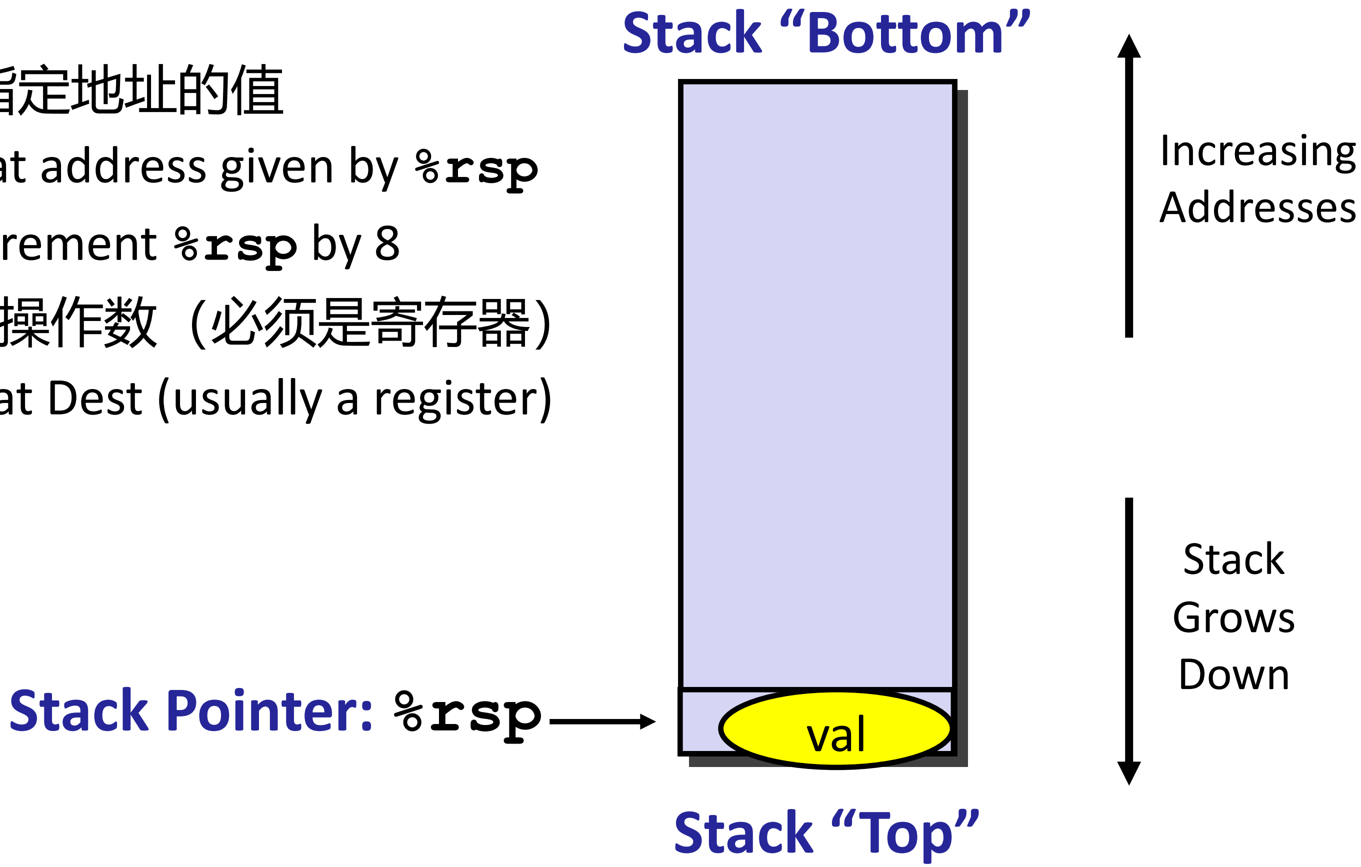
X86-64栈：压栈 x86-64 Stack: Push

- **`pushq`** ***Src***
  - 获取Src处的操作数 Fetc val at *Src*
  - %rsp减8 Decrement **`%rsp`** by 8
  - 把操作数写入%rsp指定的地址
  - Write operand at address given by **`%rsp`**



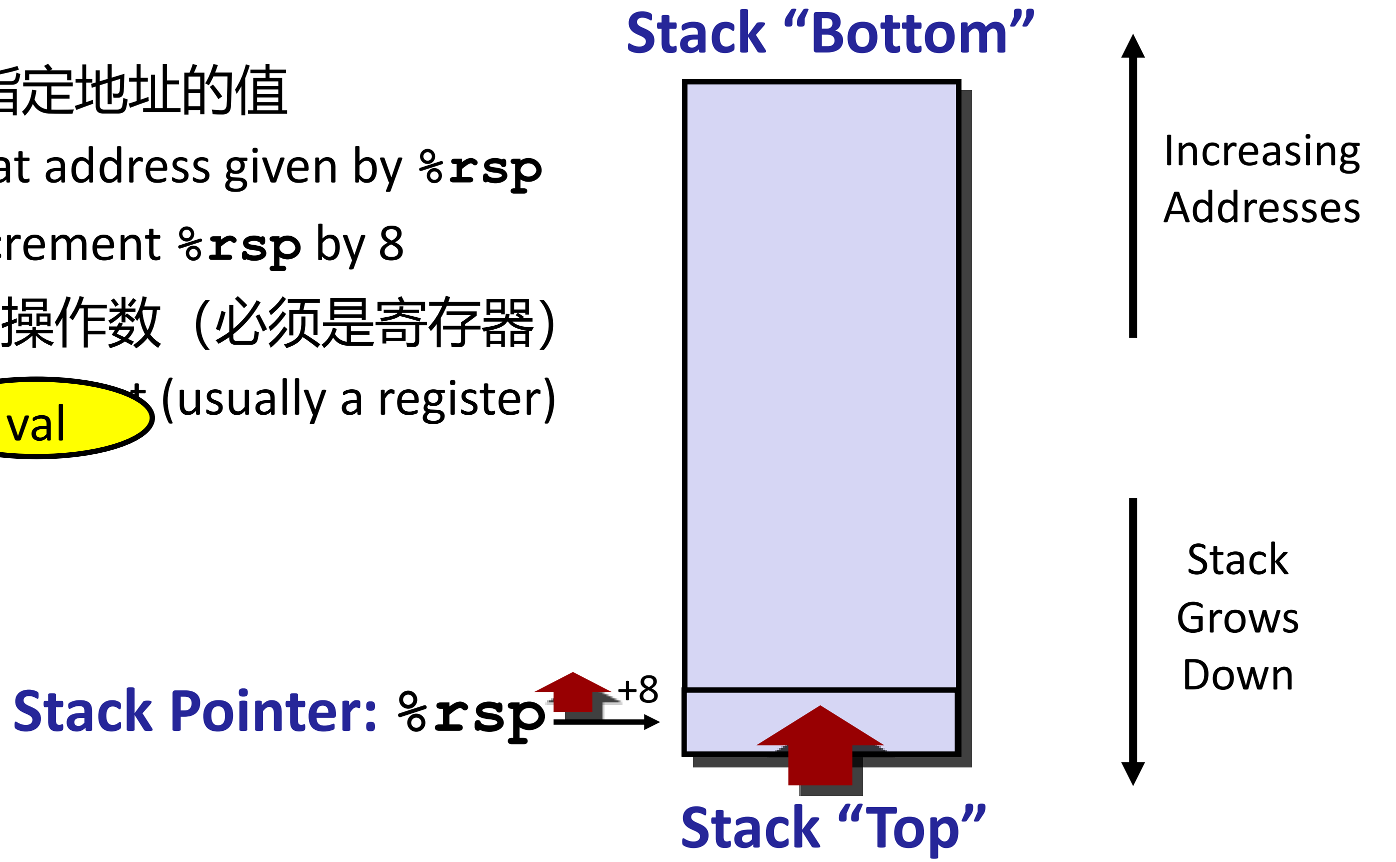
x86-64栈：弹出栈 x86-64 Stack: Pop

- `popq` ***Dest***
  - 读取由%rsp指定地址的值
    - Read value at address given by `%rsp`
  - %rsp增加8 Increment `%rsp` by 8
  - 存储值到目的操作数（必须是寄存器）
    - Store value at Dest (usually a register)



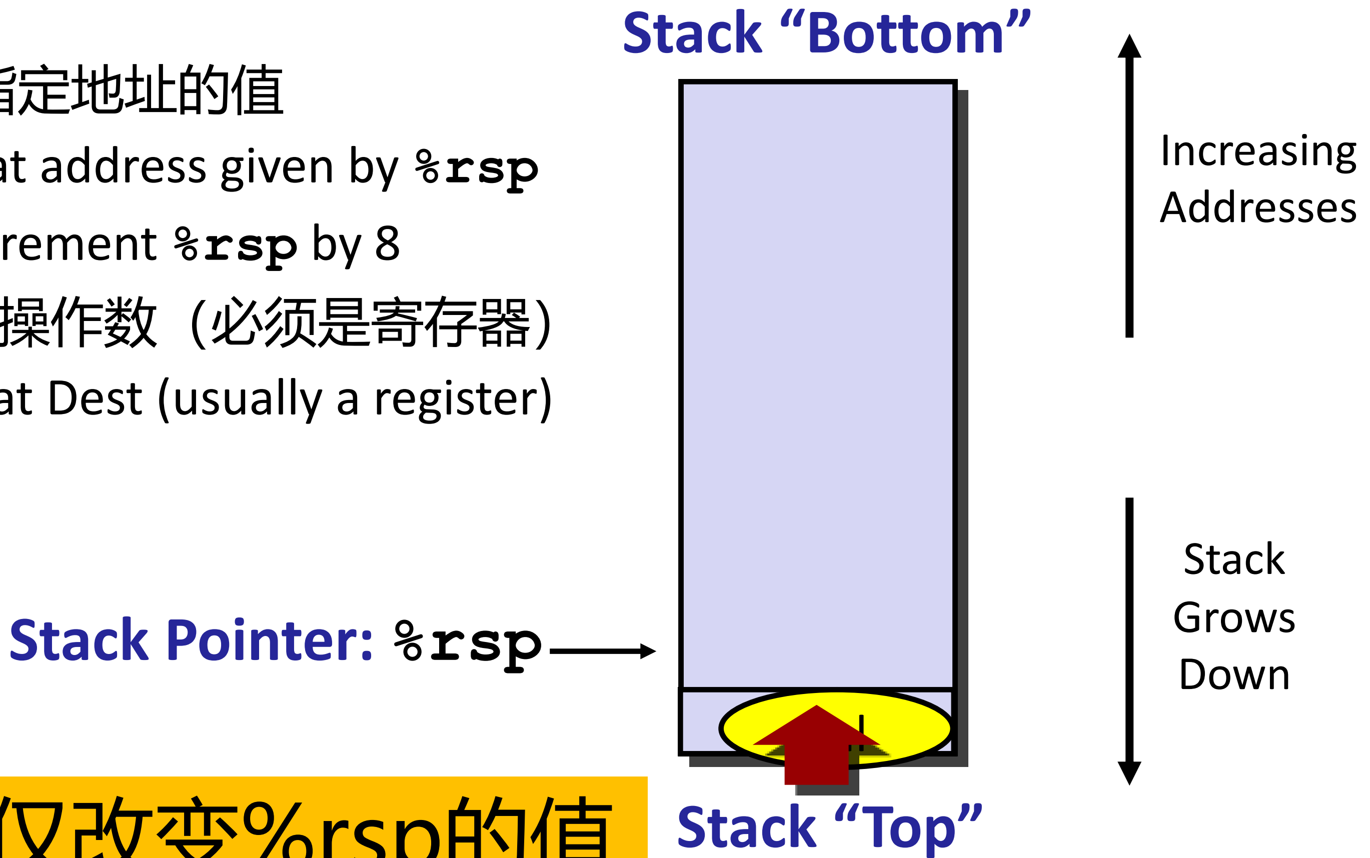
x86-64栈：弹出栈 x86-64 Stack: Pop

- `popq` ***Dest***
  - 读取由%rsp指定地址的值
    - Read value at address given by `%rsp`
  - %rsp增加8 Increment `%rsp` by 8
  - 存储值到目的操作数（必须是寄存器）
    - Store val val t (usually a register)



x86-64栈：弹出栈 x86-64 Stack: Pop

- `popq` ***Dest***
  - 读取由%rsp指定地址的值
    - Read value at address given by `%rsp`
  - %rsp增加8 Increment `%rsp` by 8
  - 存储值到目的操作数（必须是寄存器）
    - Store value at Dest (usually a register)



(内存没变，仅改变%rsp的值
The memory doesn't change,
only the value of `%rsp`)

议题

- 过程 Procedures
  - 栈结构 Stack Structure
  - 调用规则 Calling Conventions
    - 传递控制 Passing control
    - 传递数据 Passing data
    - 管理局部数据 Managing local data
  - 递归说明 Illustration of Recursion

代码示例
Code Examples

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

```
0000000000400540 <multstore>:
  400540: push   %rbx               # Save %rbx
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  400549: mov    %rax,(%rbx)        # Save at dest
  40054c: pop    %rbx               # Restore %rbx
  40054d: retq                      # Return
```

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

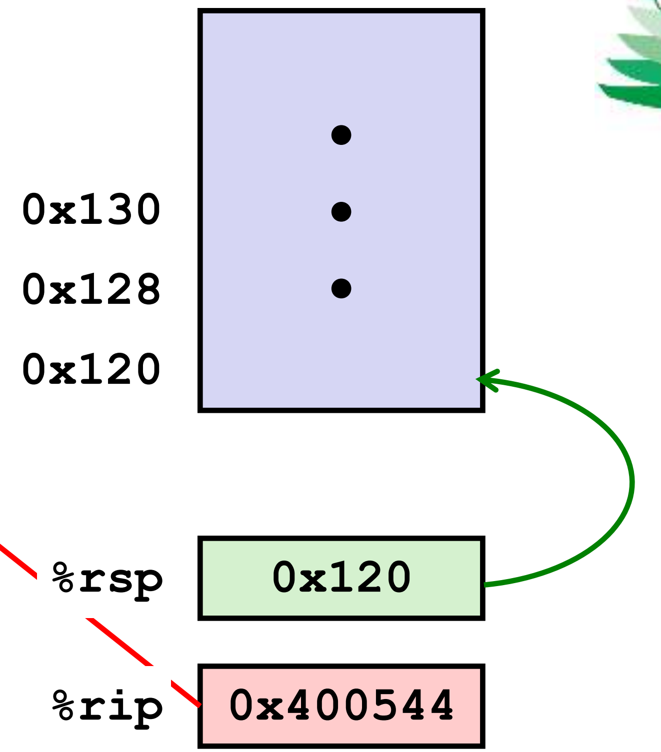
```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax       # a
  400553:  imul   %rsi,%rax       # a * b
  400557:  retq                   # Return
```

过程控制流 Procedure Control Flow

- **使用栈支持过程调用和返回 Use stack to support procedure call and return**
- **过程调用 Procedure call: `call label`**
  - 将返回地址压入栈 Push return address on stack
  - 跳转到标号处 Jump to ***label***
- **返回地址 Return address:**
  - 调用指令之后那条指令的地址 Address of the next instruction right after call
  - 反汇编的示例 Example from disassembly
- **过程返回 Procedure return: `ret`**
  - 从栈弹出地址 Pop address from stack
  - 跳转到该地址 Jump to address

控制流示例#1
Control Flow Example #1

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

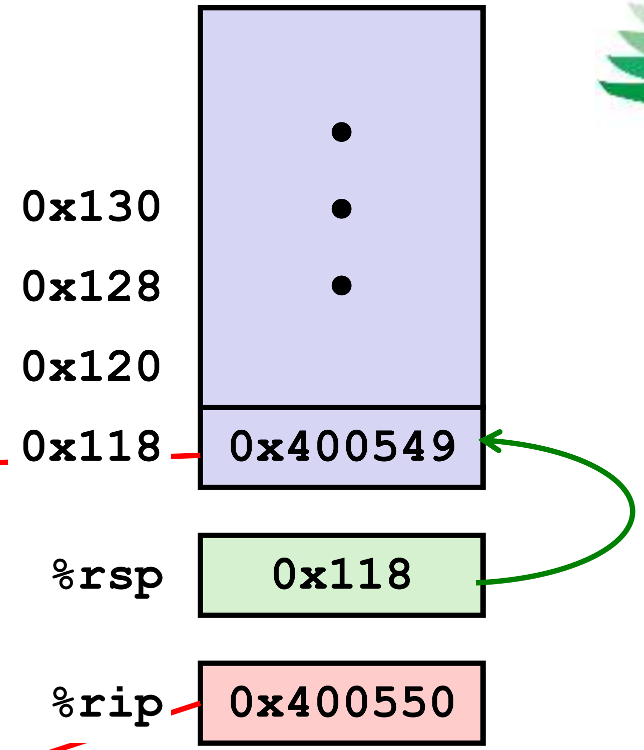



```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

控制流示例#2
Control Flow Example #2

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

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

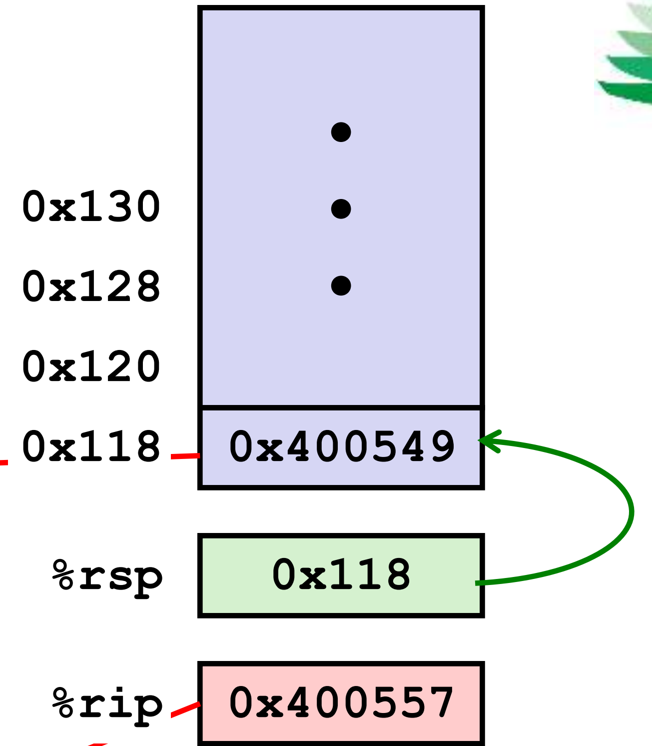


控制流示例#3

Control Flow Example #3

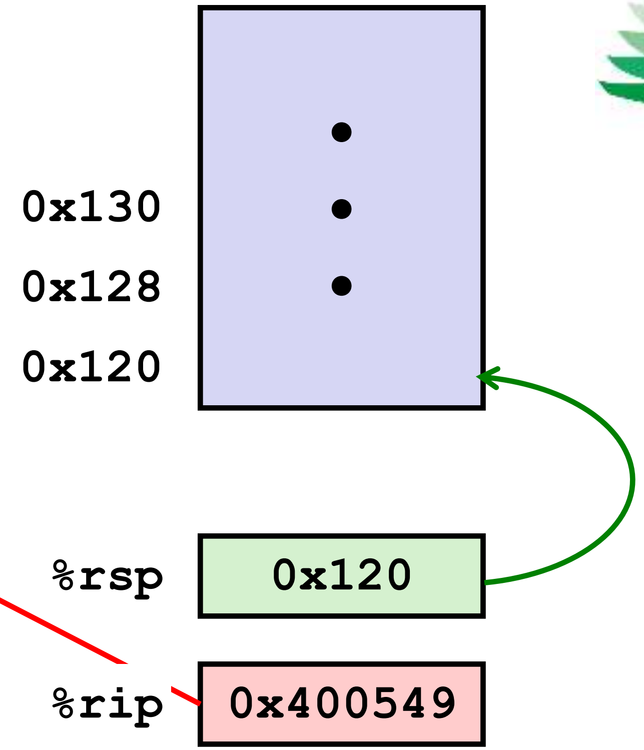
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

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



控制流示例#4
Control Flow Example #4

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




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

议题

- 过程 Procedures
  - 栈结构 Stack Structure
  - 调用规则 Calling Conventions
    - 传递控制 Passing control
    - 传递数据 Passing data
    - 管理局部数据 Managing local data
  - 递归说明 Illustrations of Recursion & Pointers

过程数据流 Procedure Data Flow

寄存器 Registers

- **前6个参数 First 6 arguments**

`%rdi`
`%rsi`
`%rdx`
`%rcx`
`%r8`
`%r9`

- **返回值 Return value**

`%rax`

栈 Stack

• • •
Arg *n*
• • •
Arg 8
Arg 7

- **仅在需要时才分配栈空间**
  Only allocate stack space when needed

数据流示例 Data Flow Examples

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

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  • • •
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)        # Save at 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                   # Return
```

议题

- 过程 Procedures
  - 栈结构 Stack Structure
  - 调用规则 Calling Conventions
    - 传递控制 Passing control
    - 传递数据 Passing data
    - **管理局部数据 Managing local data**
  - 递归说明 Illustration of Recursion

基于栈的语言 Stack-Based Languages

- **支持递归的语言 Languages that support recursion**
  - 例如 e.g., C, Pascal, Java
  - 代码必须是“可重入的” Code must be “*Reentrant*”
    - 单一过程同时有多个实例 Multiple simultaneous instantiations of single procedure
  - 需要有一些地方存储每个实例的状态 Need some place to store state of each instantiation
    - 参数 Arguments
    - 局部变量 Local variables
    - 返回指针 Return pointer
- **栈规则 Stack discipline**
  - 在限定的时间内对于给定的过程需要的状态 State for given procedure needed for limited time
    - 从过程被调用到过程返回 From when called to when return
  - 被调用者在调用者返回之前返回 Callee returns before caller does
- **栈分配以栈帧形式 Stack allocated in *Frames***
  - 单一过程实例的状态 state for single procedure instantiation

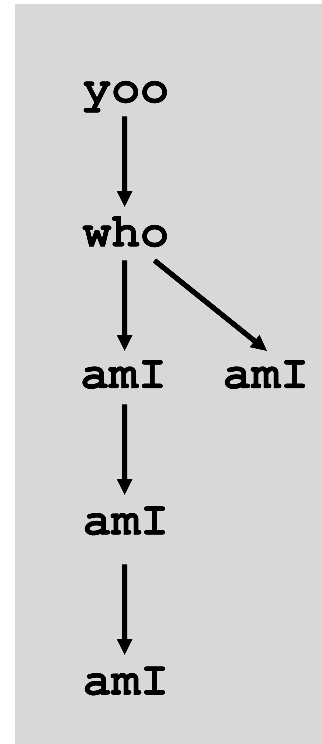
调用链的示例 Call Chain Example

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

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

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

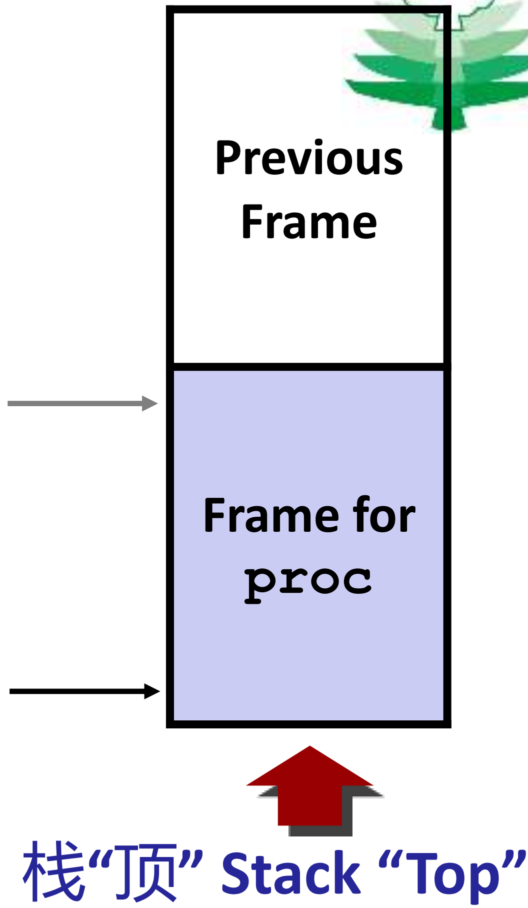
示例 Example
调用链 Call Chain



过程**amI**是递归的 **Procedure amI() is recursive**

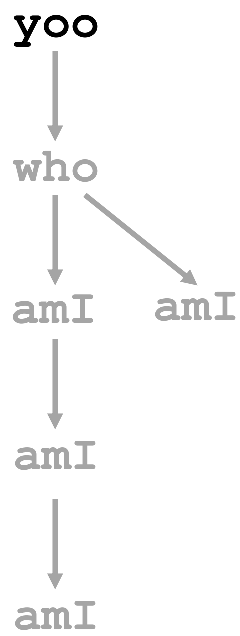
栈帧 Stack Frames

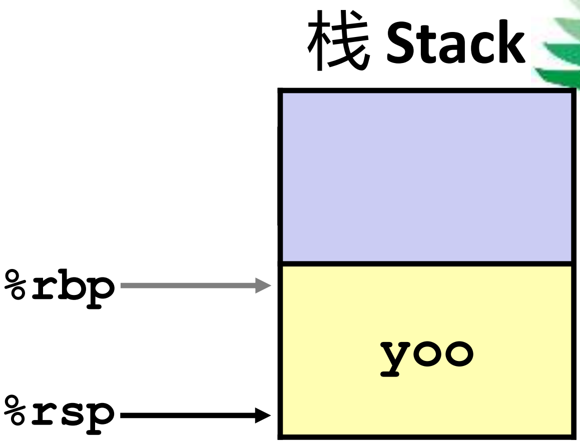
- **内容 Contents**
  - 返回信息 Return information
  - 局部存储（如果需要）
    - Local storage (if needed)
  - 临时空间（如果需要）
    - Temporary space (if needed)
- **管理 Management**
  - 当进入过程时分配空间 Space allocated when enter procedure
    - “初始”代码 “Set-up” code
    - 包括call指令的压栈 Includes push by **`call`** instruction
  - 当返回时释放空间 Deallocated when return
    - “结束”代码 “Finish” code
    - 包括ret指令的弹出栈 Includes pop by **`ret`** instruction



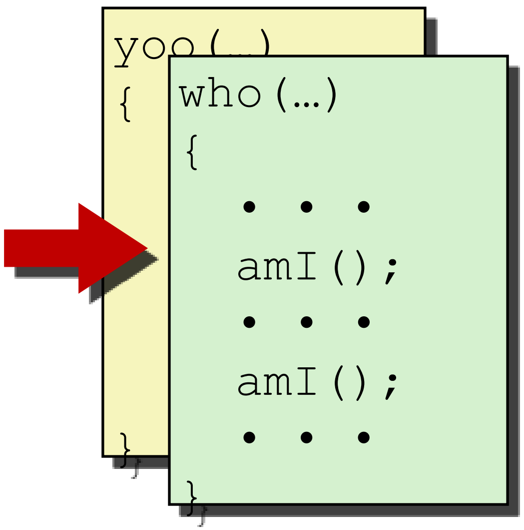
示例 Example

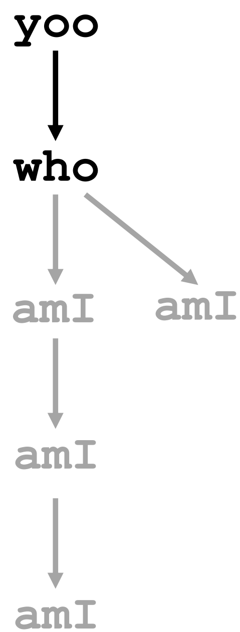
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```



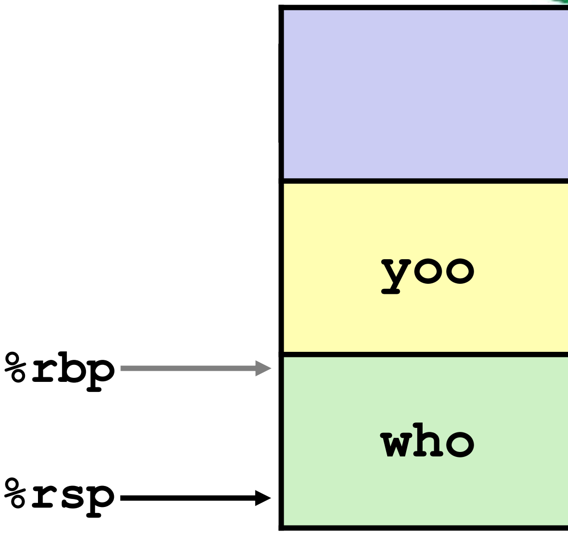


示例 Example

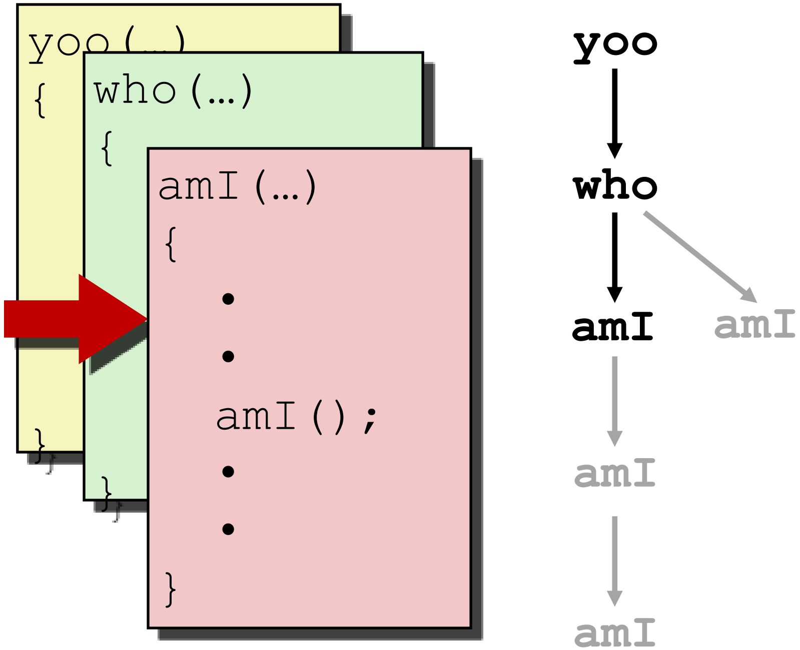




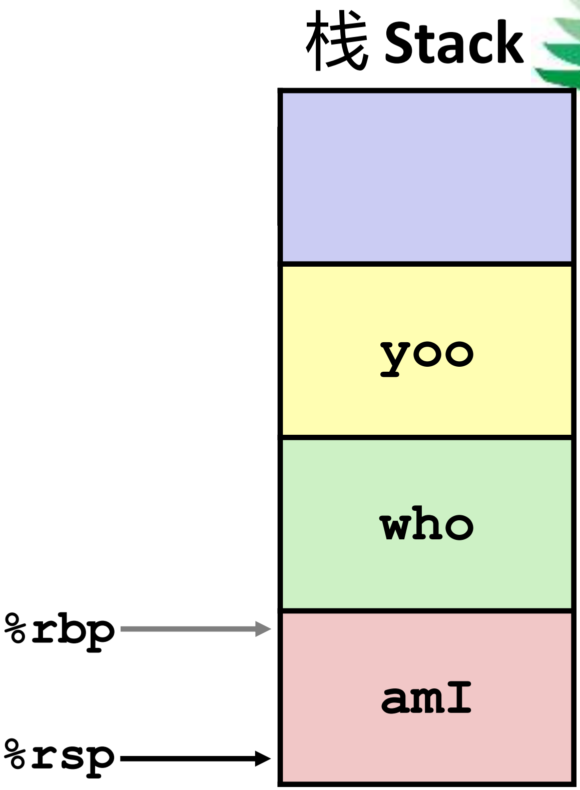
栈 Stack



示例 Example

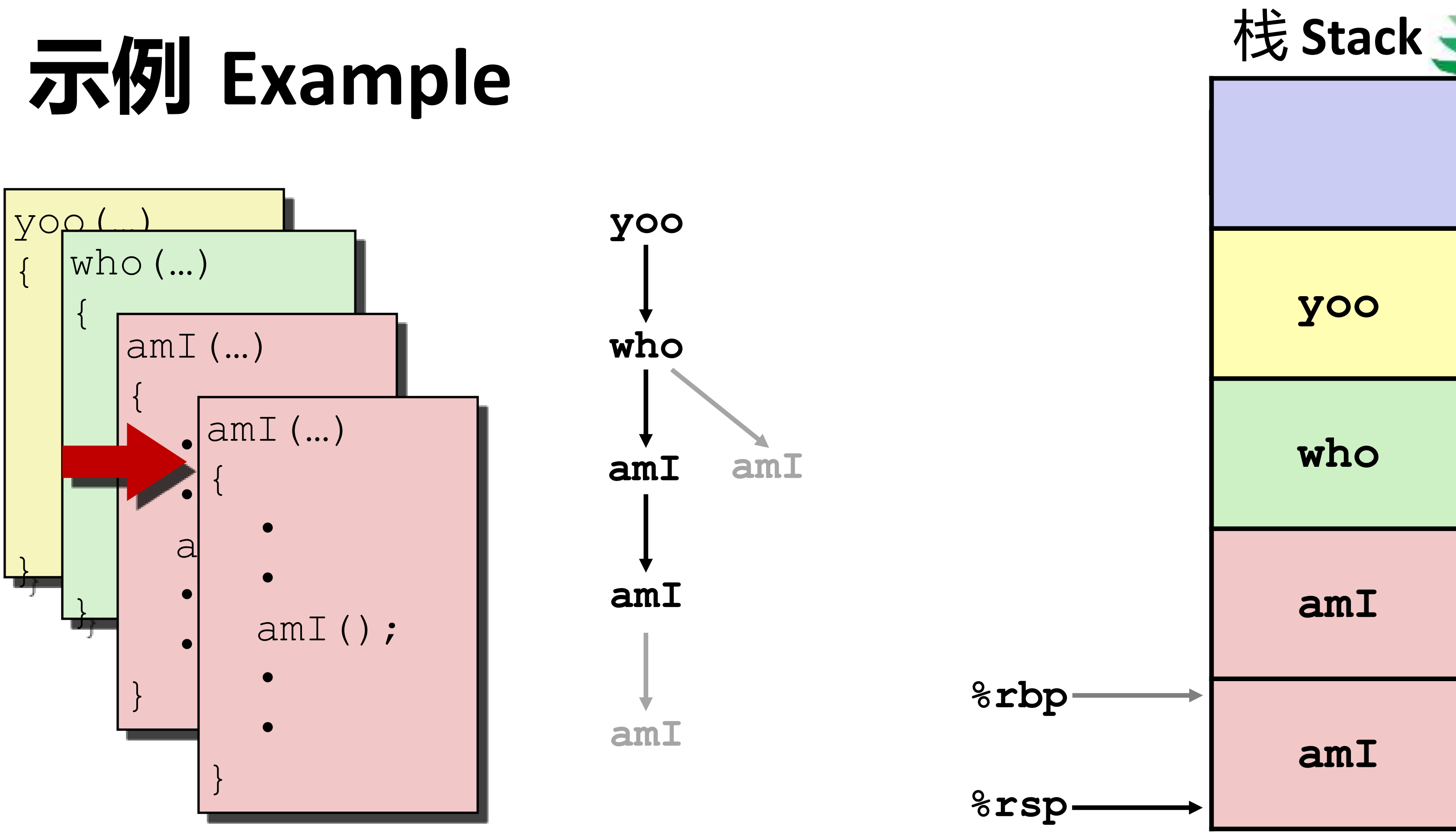


yoo (…)
{
}
who (…)
{
}
amI (…)
{
•
•
amI ();
•
•
}
yoo
who
amI
amI
amI
amI



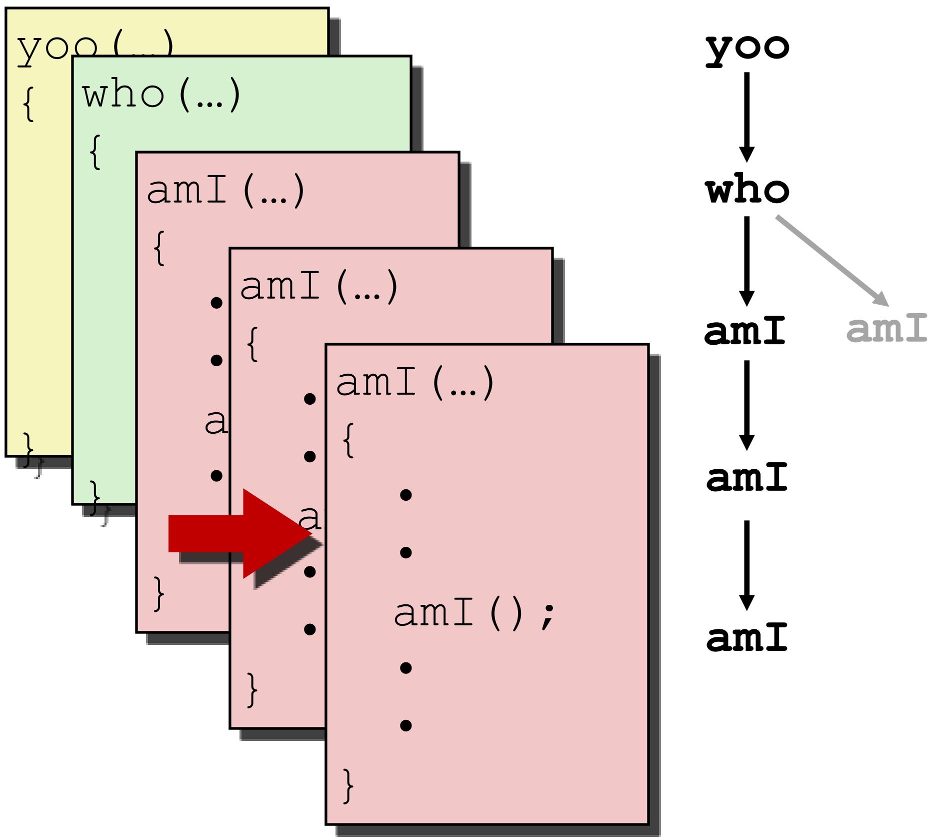
栈 Stack
yoo
who
amI
%rbp
%rsp

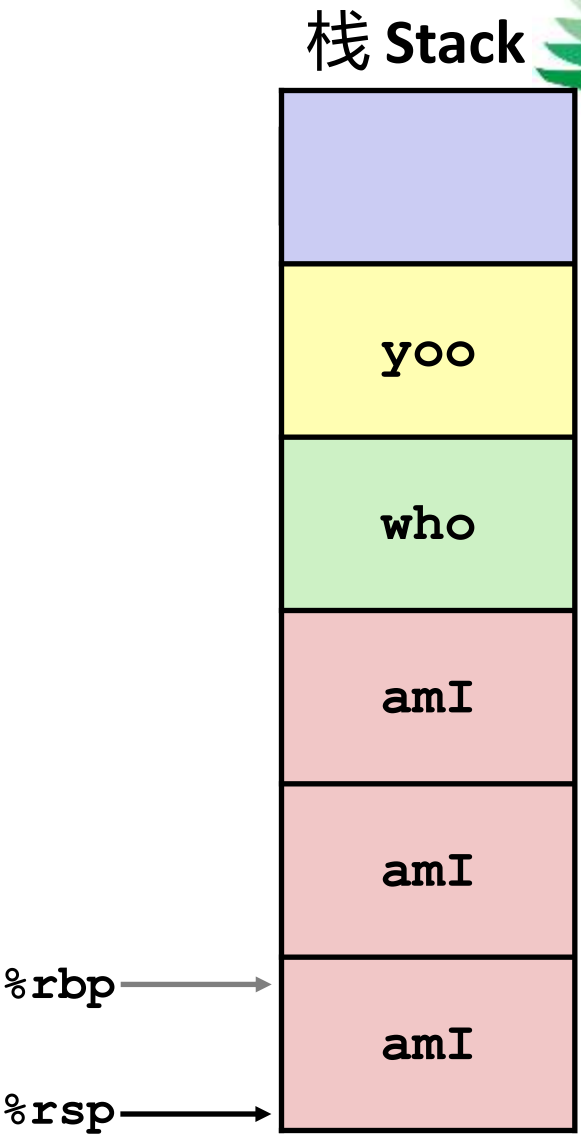
示例 Example



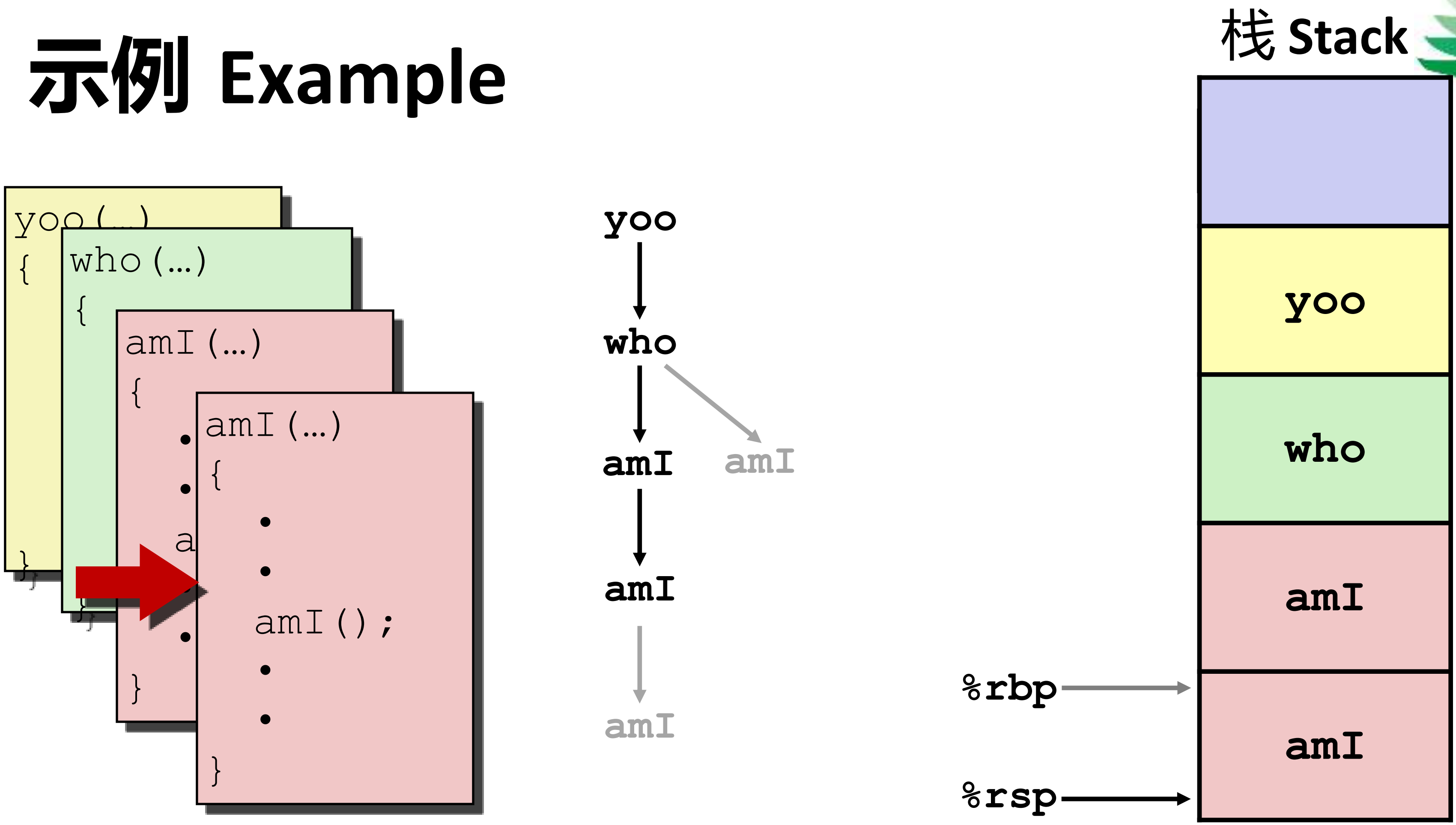
yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
栈 Stack
yoo
who
amI
amI
%rbp
%rsp

示例 Example


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

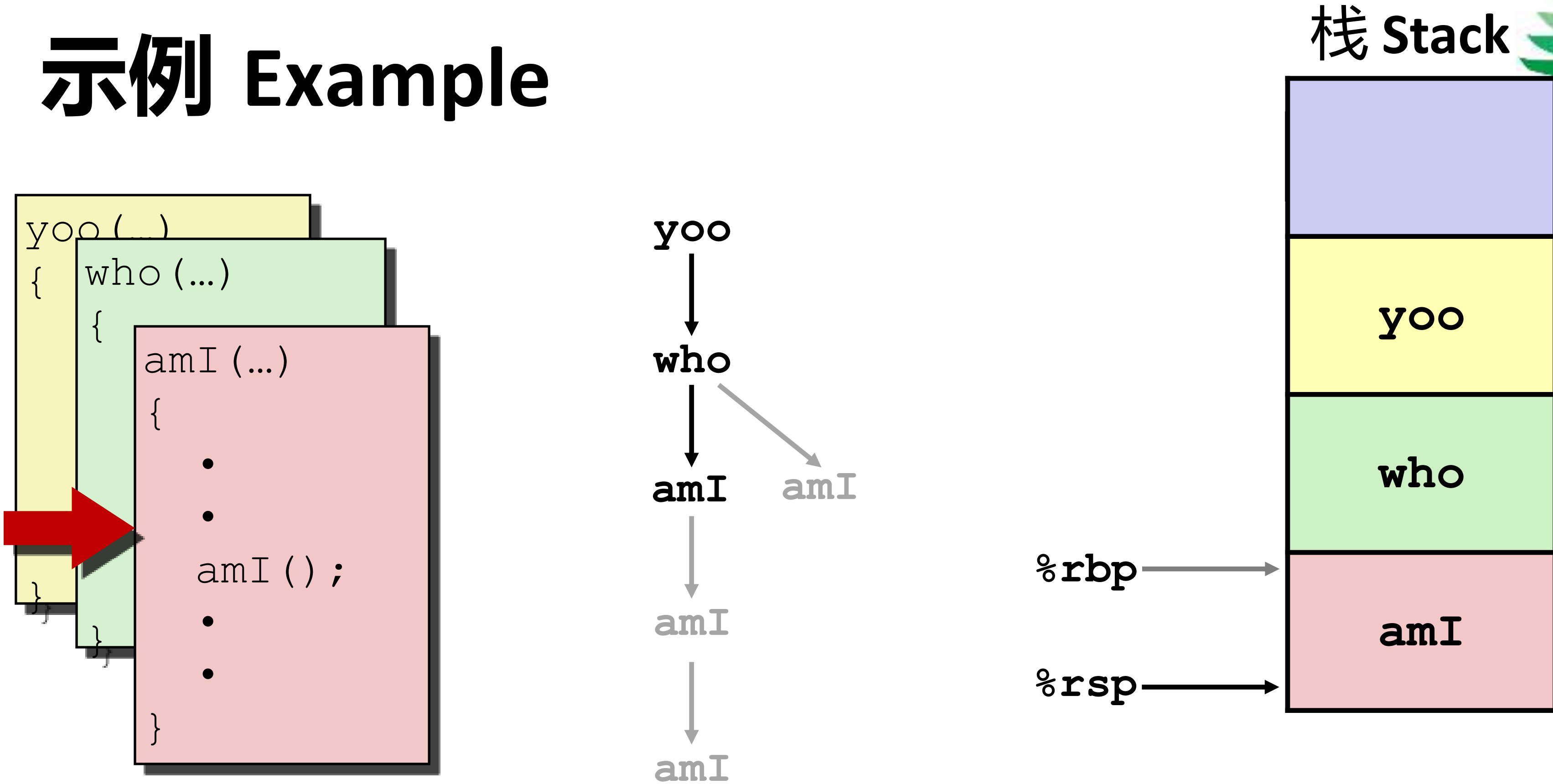

栈 Stack
yoo
who
amI
amI
amI
%rbp
%rsp

示例 Example



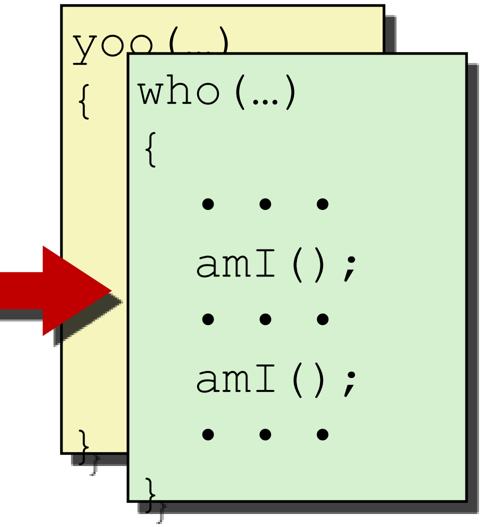
yoo(…)
{
who(…)
{
amI(…)
{
a
}
amI(…)
{
amI();
}
yoo
who
amI
amI
amI
amI
栈 Stack
yoo
who
amI
amI
%rbp
%rsp

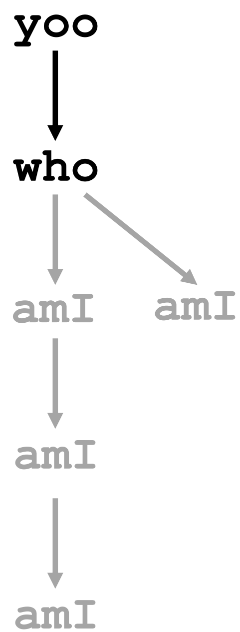
示例 Example

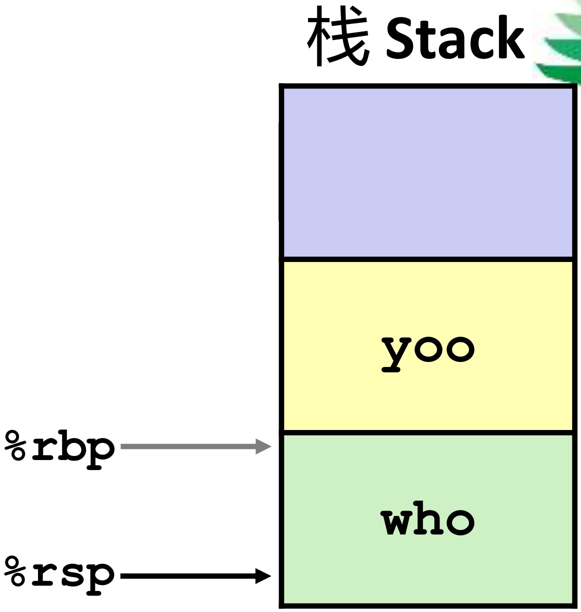


yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}
yoo
who
amI
amI
amI
amI
栈 Stack
yoo
who
amI
%rbp
%rsp

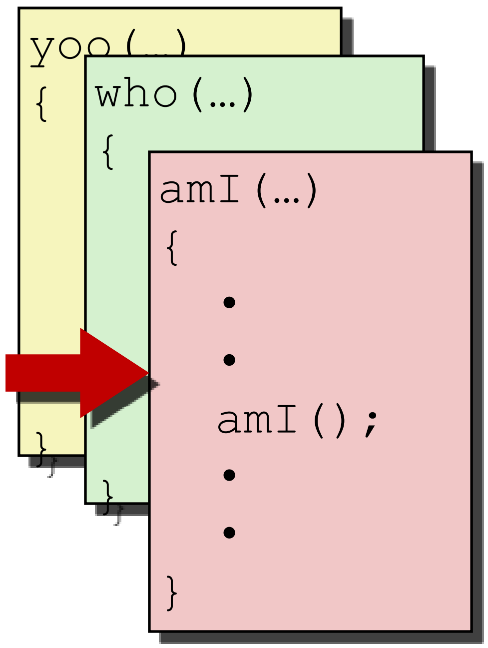
示例 Example



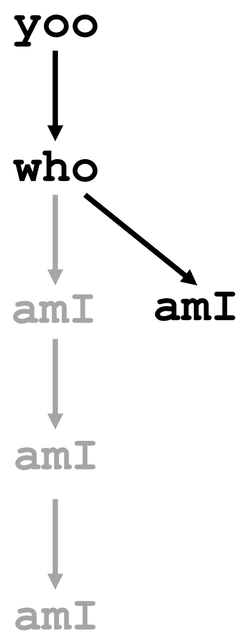




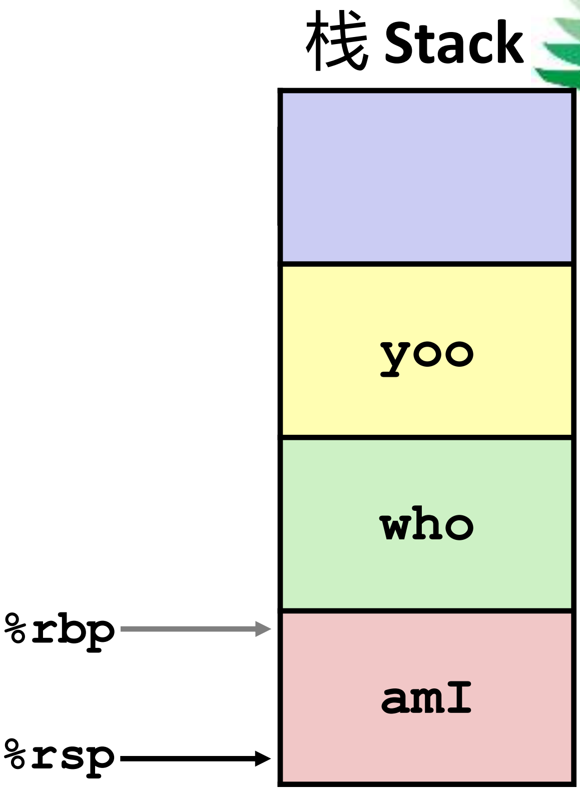
示例 Example



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

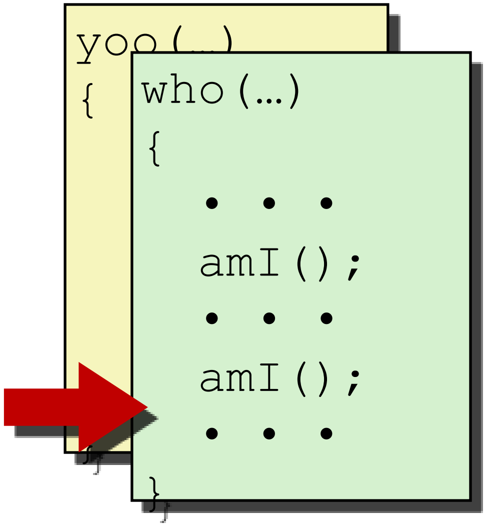


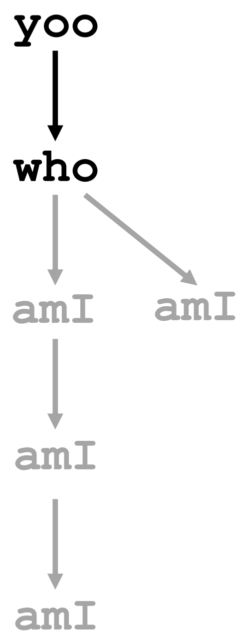
yoo
who
amI
amI
amI
amI

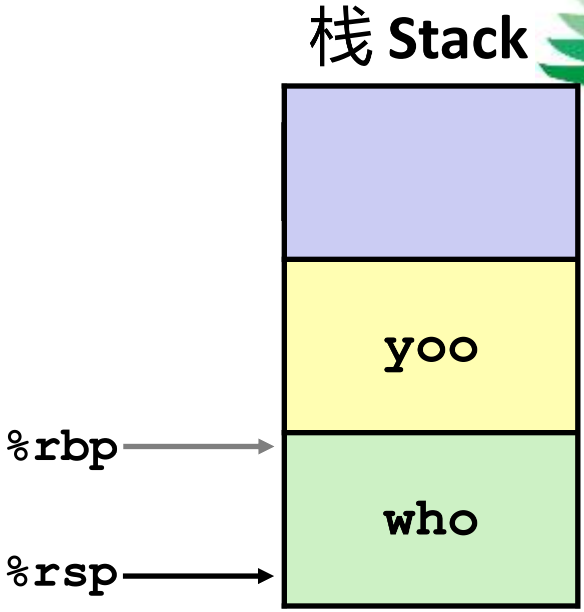


栈 Stack
yoo
who
amI
%rbp
%rsp

示例 Example

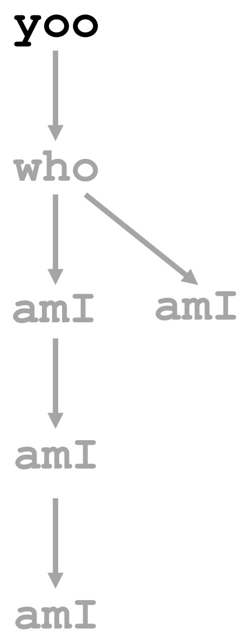


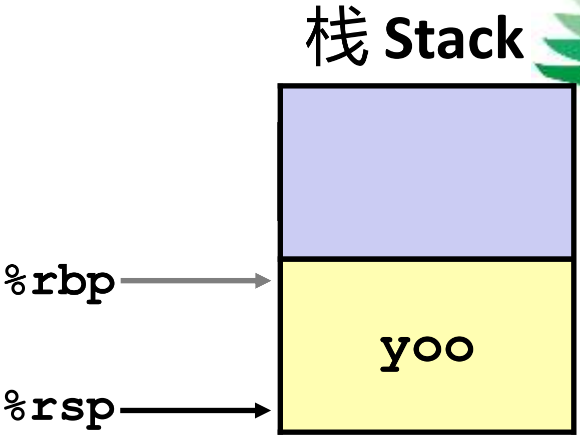




示例 Example

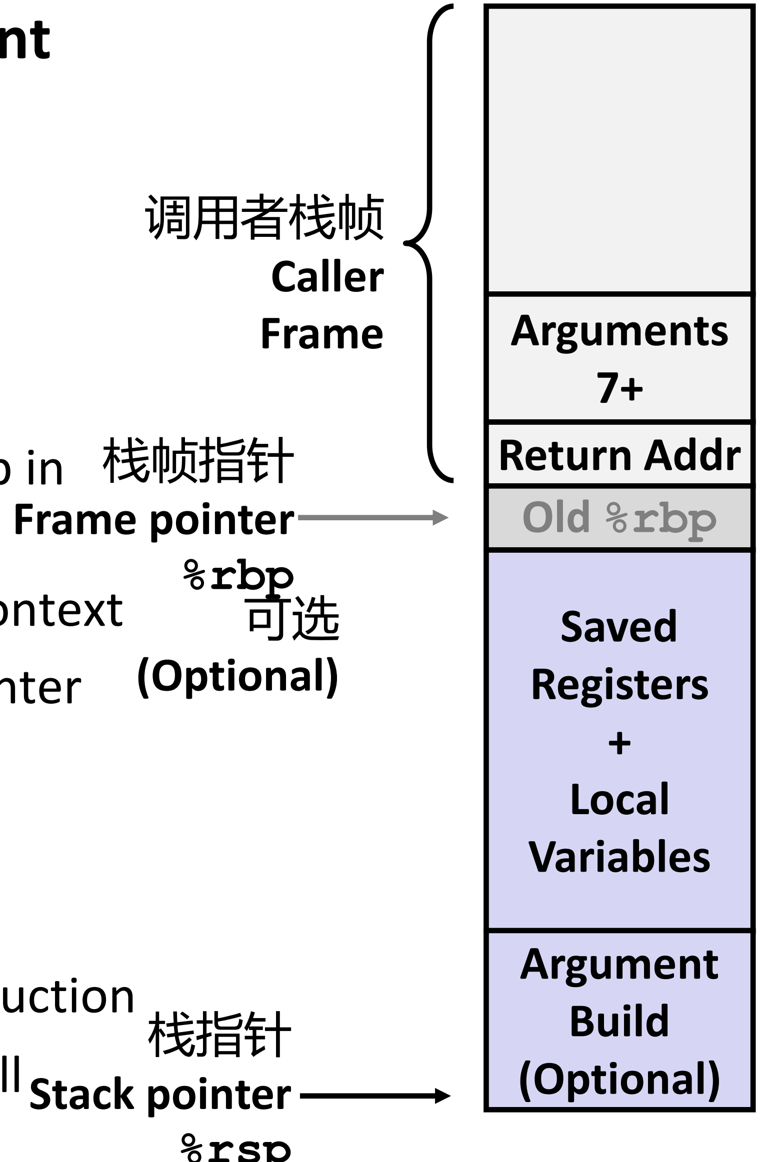
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```





x86-64/Linux栈帧
x86-64/Linux Stack Frame

- **当前栈帧（自“顶”向下）Current Stack Frame (“Top” to Bottom)**
  - “参数构建：” 有关调用的函数参数 “Argument build:” Parameters for function about to call
  - 局部变量 Local variables 如果不能存储在寄存器中 If can’t keep in registers
  - 保存的寄存器上下文 Saved register context
  - 老的栈帧指针（可选） Old frame pointer (optional)
- **调用者栈帧 Caller Stack Frame**
  - 返回地址 Return address
    - Call指令压栈 Pushed by `call` instruction
  - 本次调用的参数 Arguments for this call



示例：incr　Example: `incr`

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

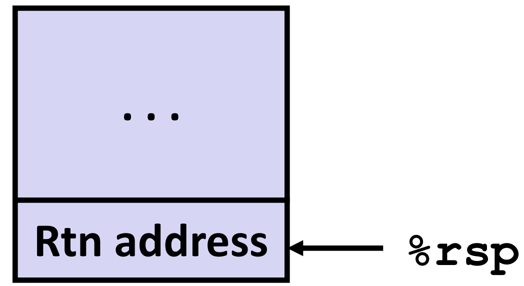
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

寄存器 Register	用途 Use(s)
`%rdi`	参数p Argument `p`
`%rsi`	参数val,y Argument `val`, `y`
`%rax`	`x`, 返回值 Return value

示例：调用incr #1
Example: Calling `incr` #1

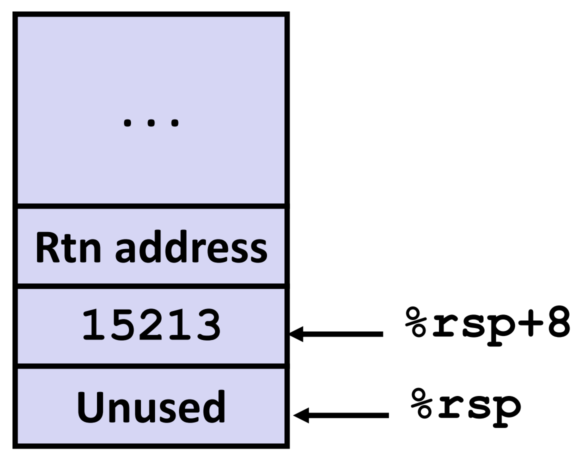
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

初始栈结构 **Initial Stack Structure**




```
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
```

结果栈结构 **Resulting Stack Structure**

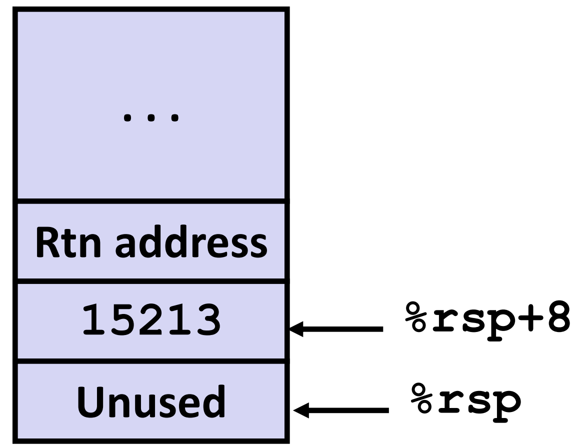


示例：调用incr #2
Example: Calling `incr` #2

```
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
```

栈结构 Stack Structure

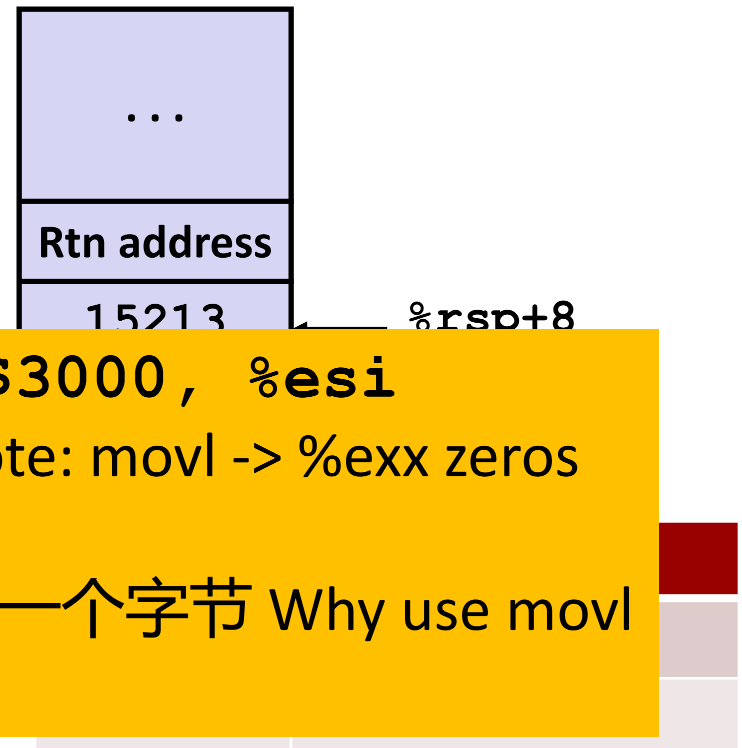


寄存器 Register	用途 Use(s)
%rdi	&v1
%rsi	3000

示例：调用incr #2
Example: Calling `incr` #2

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

栈结构 **Stack Structure**



旁注1：Aside 1: **`movl   $3000, %esi`**

- 注意：movl指令把高32位置零 Note: movl -> %exx zeros out high order 32 bits.
- 为何不使用movq指令？这样节省一个字节 Why use movl instead of movq? 1 byte shorter.

```
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

示例：调用incr #2
Example: Calling `incr` #2

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

栈结构 Stack Structure

...
Rtn address
15213 ← %rsp+8
← %rsp

旁注2：Aside 2: `leaq   8(%rsp), %rdi`

- 计算%rsp+8 Computes %rsp+8
- 实际上，用于它的含义 Actually, used for what it is meant!

```
ca
    ...
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp
    ret
```

	se(s)
	v1
	000

示例：调用incr #2
Example: Calling `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
```

栈结构 **Stack Structure**

...	
Rtn address	
15213	← %rsp+8
Unused	← %rsp

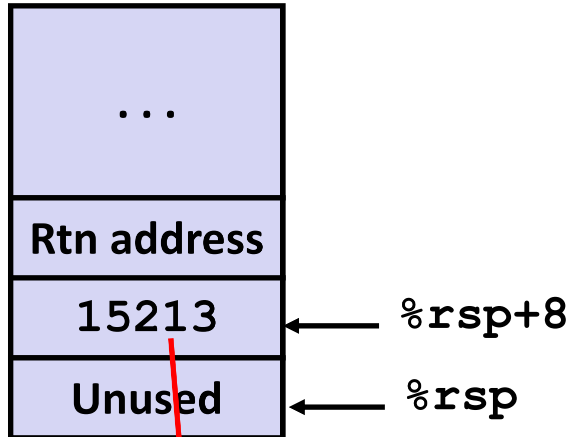
寄存器 Register	用途 Use(s)
%rdi	&v1
%rsi	3000

示例：调用incr #3a
Example: Calling `incr` #3a

```
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
```

栈结构 Stack Structure



寄存器 Register	用途 Use(s)
%rdi	&v1
%rsi	3000

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

示例：调用incr #3b

Example: Calling `incr` #3b

```
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
```

栈结构 **Stack Structure**

. . .
Rtn address
18213 ← %rsp+8
Unused ← %rsp

寄存器 Register	用途 Use(s)
%rdi	&v1
%rsi	3000

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

示例：调用incr #4

Example: Calling `incr` #4

```
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
```

栈结构 Stack Structure

Stack	
...	
Rtn address	
18213	← %rsp+8
Unused	← %rsp

寄存器 Register	用途 Use(s)
%rax	返回值 Return value, 15213

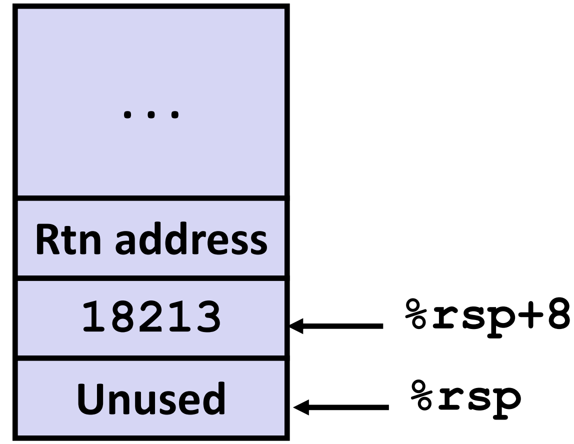
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

示例：调用incr #5a
Example: Calling `incr` #5a

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

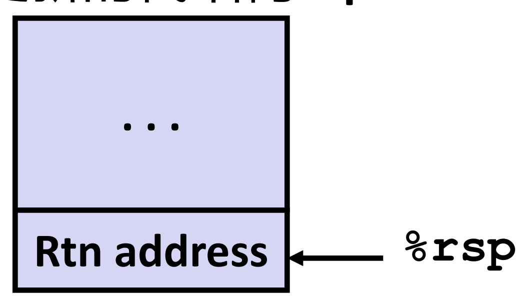
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

栈结构 Stack Structure



寄存器 Register	用途 Use(s)
`%rax`	Return value

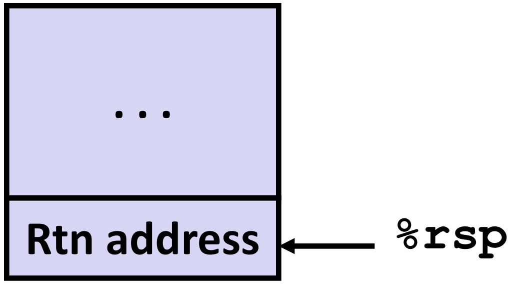
更新的栈结构 Updated Stack Structure



示例：调用incr #5b
Example: Calling `incr` #5b

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

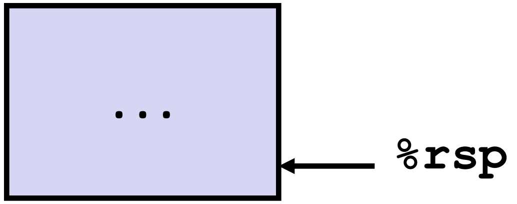
更新的栈结构 Updated Stack Structure




```
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
```

寄存器 Register	用途 Use(s)
`%rax`	返回值 Return value

最终栈结构 Final Stack Structure



寄存器保存规则
Register Saving Conventions

- **当过程yoo调用who时 When procedure `yoo` calls `who`:**
  - `Yoo`是调用者 `yoo` is the ***caller***
  - `Who`是被调用者 `who` is the ***callee***
- **寄存器可以用于临时存储吗？ Can register be used for temporary storage?**

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

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

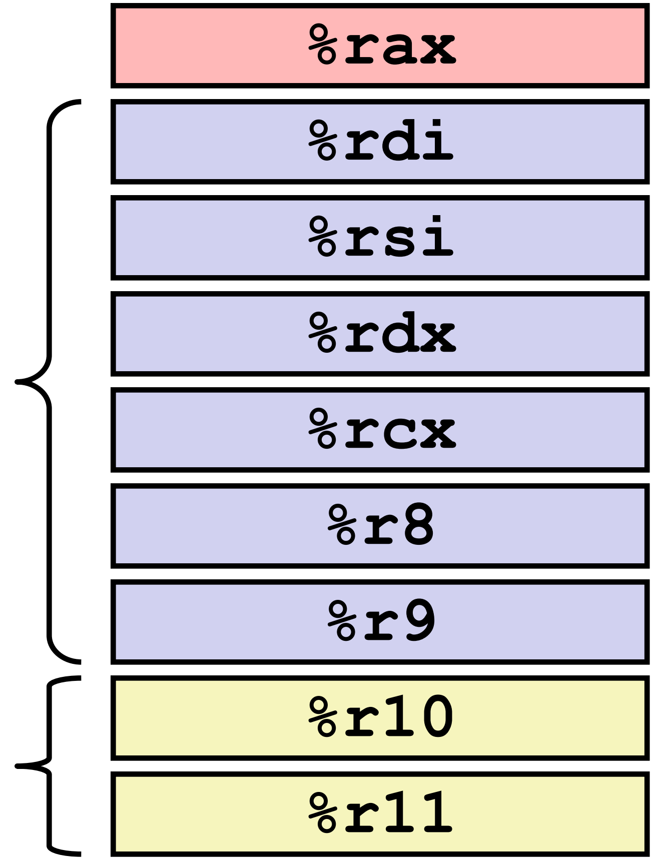
- 寄存器%rdx的内容被who写覆盖 Contents of register `%rdx` overwritten by `who`
- 这样会有麻烦，需要做些事情 This could be trouble → something should be done!
  - 需要一些协作 Need some coordination

寄存器保存规则
Register Saving Conventions

- **当过程yoo调用who时 When procedure `yoo` calls `who`:**
  - `Yoo`是调用者 `yoo` is the ***caller***
  - `Who`是被调用者 `who` is the ***callee***
- **寄存器可以用于临时存储吗? Can register be used for temporary storage?**
- **规则 Conventions**
  - *"调用者负责保存" **"Caller Saved"***
    - 调用者在调用前在其栈帧中保存临时值 Caller saves temporary values in its frame before the call
  - *"被调用者负责保存" **"Callee Saved"***
    - 被调用者在使用前在其栈帧中保存临时值 Callee saves temporary values in its frame before using
    - 被调用者在返回到调用者之前恢复临时值 Callee restores them before returning to caller

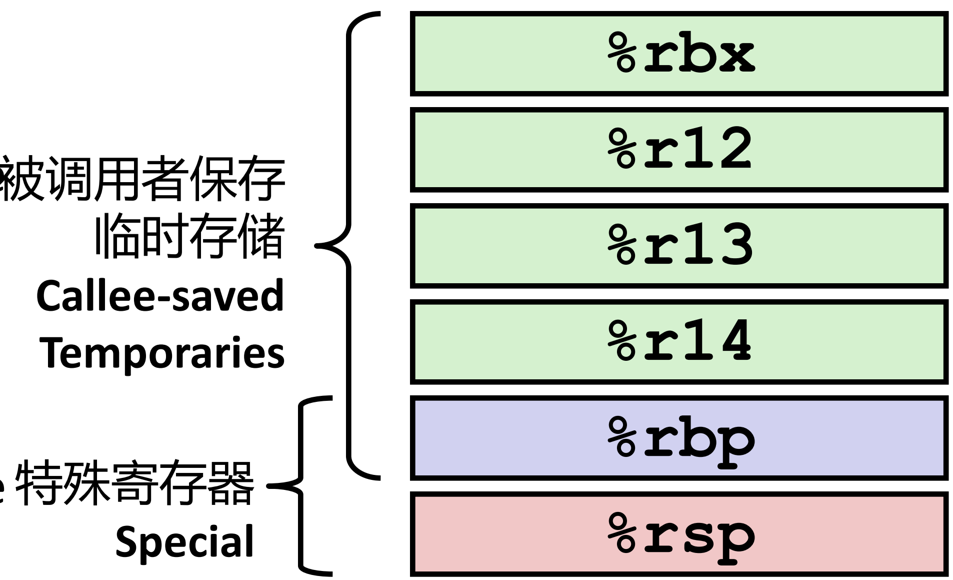
x86-64Linux寄存器用法 #1
x86-64 Linux Register Usage #1

- **`%rax`**
  - 返回值 Return value
  - 也是调用者保存 Also caller-saved
  - 可以被过程修改 Can be modified by procedure
- **`%rdi, ..., %r9`**
  - 参数 Arguments
  - 也是调用者保存 Also caller-saved
  - 可以被过程修改Can be modified by procedure
- **`%r10, %r11`**
  - 调用者保存 Caller-saved
  - 可以被过程修改 Can be modified by procedure



x86-64Linux寄存器用法 #2
x86-64 Linux Register Usage #2

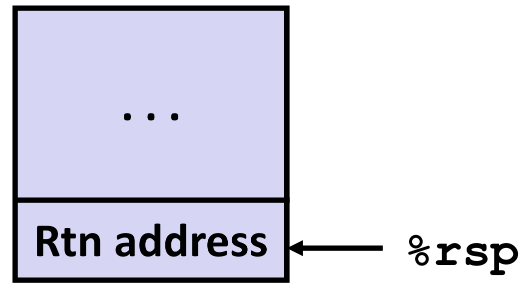
- **%rbx, %r12, %r13, %r14**
  - 被调用者保存 Callee-saved
  - 被调用者必须保存和恢复 Callee must save & restore
- **%rbp**
  - 被调用者保存 Callee-saved
  - 被调用者必须保存和恢复 Callee must save & restore
  - 可能用作栈帧指针 May be used as frame pointer
  - 能够混合和匹配 Can mix & match
- **%rsp**
  - 被调用者保存的特殊形式 Special form of callee save
  - 从过程退出时恢复到原始值 Restored to original value upon



被调用者保存示例#1
Callee-Saved Example #1

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

初始栈结构 **Initial Stack Structure**



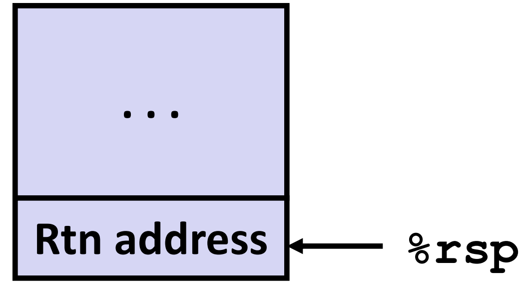
- x保存在%rdi寄存器 **`x`** comes in register **`%rdi`**.
- 调用incr时需要%rdi We need **`%rdi`** for the call to **`incr`**.
- x应该放在哪里，才能在调用incr后可以使用它 Where should be put **`x`**, so we can use it after the call to **`incr`**?

被调用者保存示例#2
Callee-Saved Example #2

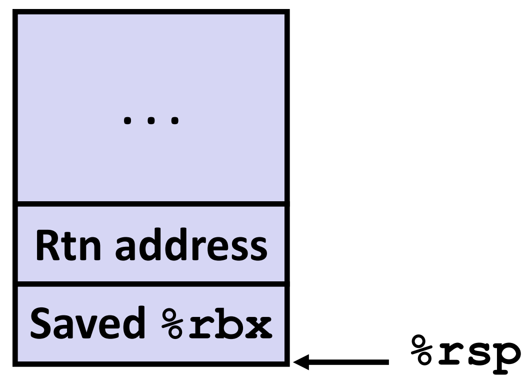
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

初始栈结构 Initial Stack Structure



结果栈结构 Resulting Stack Structure

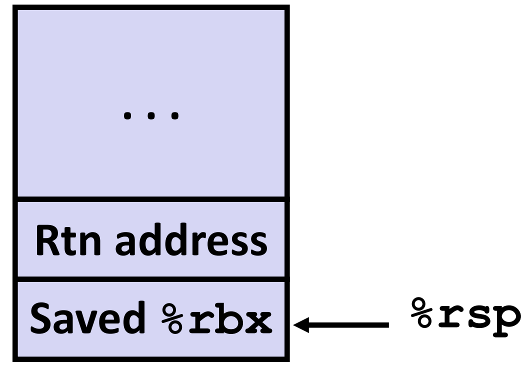


被调用者保存示例#3
Callee-Saved Example #3

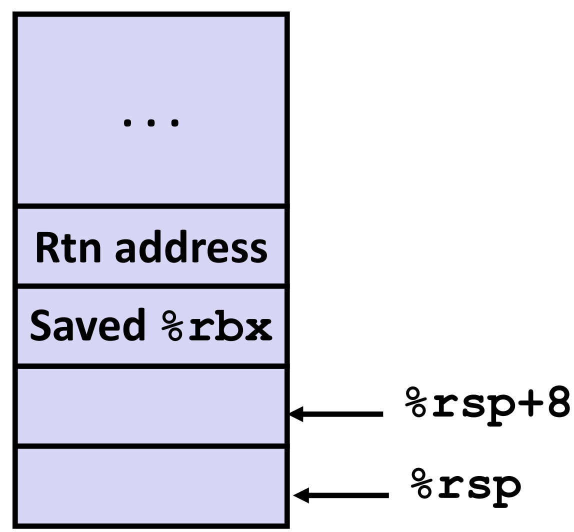
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
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
```

初始栈结构 **Initial Stack Structure**



结果栈结构 **Resulting Stack Structure**

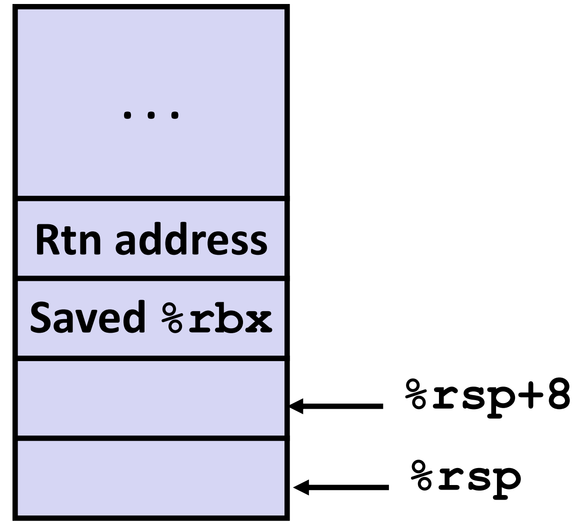


被调用者保存示例#4
Callee-Saved Example #4

```
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
```

栈结构 **Stack Structure**



- x保存在%rbx中，这是由被调用者保存的寄存器 **x** is saved in **%rbx**, a callee saved register

被调用者保存示例#5
Callee-Saved Example #5

```
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
```

栈结构 **Stack Structure**

...
Rtn address
Saved %rbx
15213 ← %rsp+8
Unused ← %rsp

- x保存在%rbx中，这是由被调用者保存的寄存器 **x** is saved in **%rbx**, a callee saved register

被调用者保存示例#6
Callee-Saved Example #6

```
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
```

栈结构 **Stack Structure**

Stack	
...	
Rtn address	
Saved `%rbx`	
18213	← `%rsp+8`
Unused	← `%rsp`

incr返回后 Upon return from `incr`:

- x安全保存在%rbx中 `x` safe in `%rbx`
- 返回值v2在%rax中 Return val `v2` in `%rax`
- 计算 Compute `x+v2`: `addq %rbx, %rax`

被调用者保存示例#7
Callee-Saved Example #7

```
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
```

栈结构 Stack Structure

...	
Rtn address	
Saved %rbx	← %rsp
18213	
Unused	

- 返回结果在%rax中
  Return result in %rax

被调用者保存示例#8

Callee-Saved Example #8

```
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
```

初始栈结构 Initial Stack Structure

...	
Rtn address	
Saved %rbx	
18213	← %rsp
Unused	

最终栈结构 final Stack Structure

...	
Rtn address	
Saved %rbx	← %rsp
18213	
Unused	

议题

- 过程 Procedures
  - 栈结构 Stack Structure
  - 调用规则 Calling Conventions
    - 传递控制 Passing control
    - 传递数据 Passing data
    - 管理局部数据 Managing local data
  - 递归说明 Illustration of Recursion

递归函数 Recursive Function

```
/* Recursive popcount */
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
```

递归函数终止情况
Recursive Function Terminal Case

```
/* Recursive popcount */
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
```

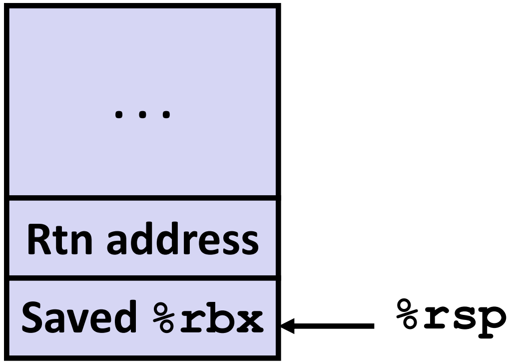
寄存器 Register	用途 Use(s)	类型 Type
%rdi	x	参数 Argument
%rax	返回值 Return value	返回值 Return value

递归函数寄存器保存
Recursive Function Register Save

```
/* Recursive popcount */
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
```

寄存器 Register	用途 Use(s)	类型 Type
%rdi	x	参数 Argument



递归函数调用设置
Recursive Function Call Setup

```
/* Recursive popcount */
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
```

寄存器 Register	用途 Use(s)	类型 Type
%rdi	x >> 1	递归参数 Rec. argument
%rbx	x & 1	调用者保存 Callee-saved

递归函数调用 Recursive Function Call

```
/* Recursive popcount */
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
```

寄存器 Register	用途 Use(s)	类型 Type
`%rbx`	`x & 1`	被调用者保存 Callee-saved
`%rax`	递归调用返回值 Recursive call return value	

递归调用结果 Recursive Function Result

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

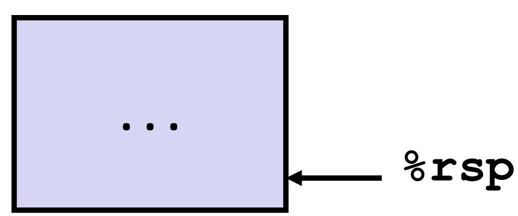
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

寄存器 Register	用途 Use(s)	类型 Type
%rbx	x & 1	调用者保存 Callee-saved
%rax	返回值 Return value	

递归函数完成 Recursive Function Completion

```
/* Recursive popcount */
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
```



寄存器 Register	用途 Use(s)	类型 Type
%rax	返回值 Return value	返回值 Return value

关于递归的观察
Observations About Recursion

- **无需特殊考虑进行处理 Handled Without Special Consideration**
  - 栈帧意味着每个函数调用都有私有的存储空间 Stack frames mean that each function call has private storage
    - 保存寄存器和局部变量 Saved registers & local variables
    - 保存返回指针 Saved return pointer
  - 寄存器保存惯例防止了一个函数调用破坏另一个的数据 Register saving conventions prevent one function call from corrupting another's data
    - 除非C语言代码显式地这么做（例如第九讲的缓冲区溢出攻击） Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
  - 栈规则遵循调用/返回模式 Stack discipline follows call / return pattern
    - 如果P调用Q，那么Q在P之前返回 If P calls Q, then Q returns before P
    - 后进，先出 Last-In, First-Out
- **同样适用于相互递归调用 Also works for mutual recursion**
  - P调用Q；Q调用P P calls Q; Q calls P

x86-64过程小结
x86-64 Procedure Summary

- **重点 Important Points**
  - 栈是过程调用/返回最合适的数据结构 Stack is the right data structure for procedure call / return
    - 如果P调用Q，那么Q在P之前返回 If P calls Q, then Q returns before P
- **递归（和相互递归）按照正常调用规则处理 Recursion (& mutual recursion) handled by normal calling conventions**
  - 可以安全存储值在局部栈帧中和被调用者保存的寄存器中 Can safely store values in local stack frame and in callee-saved registers
  - 函数参数放栈顶Put function arguments at top of stack
  - 结果通过%rax返回 Result return in `%rax`
- **指针是值的地址 Pointers are addresses of values**
  - 在栈内或全局空间 **On stack or global**