

SWE2001: System Program

0x07: Machine-Level Representation of Programs: Procedures

Hojoon Lee

Systems Security Lab @ SKKU

Processor State (x86-64, Partial)

- Information about currently executing program

- Temporary data
(**%rax**, ...)
- Location of runtime stack
(**%rsp**)
- Location of current code control point
(**%rip**, ...)
- Status of recent tests
(**CF**, **ZF**, **SF**, **OF**)

Registers

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

%rip Instruction pointer

Current stack top

CF	ZF	SF	OF
----	----	----	----

Condition codes

Condition Codes (Implicit Setting)

- ▶ Single bit registers
 - CF Carry Flag (for unsigned) SF Sign Flag (for signed)
 - ZF Zero Flag OF Overflow Flag (for signed)
- ▶ Implicitly set (think of it as side effect) by arithmetic operations

Example: **addq Src, Dest** \leftrightarrow **t = a+b**

CF set if carry out from most significant bit (unsigned overflow)

ZF set if **t == 0**

SF set if **t < 0** (as signed)

OF set if two's-complement (signed) overflow

(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)

- ▶ Not set by **leaq** instruction

Condition Codes (Explicit Setting: Compare)

- ▶ Explicit Setting by Compare Instruction
 - **cmpq Src2, Src1**
 - **cmpq b, a** like computing **a-b** without setting destination
 - **CF set** if carry out from most significant bit (used for unsigned comparisons)
 - **ZF set** if **a == b**
 - **SF set** if **(a-b) < 0** (as signed)
 - **OF set** if two's-complement (signed) overflow
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

Jumping

► jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~ (SF^OF) & ~ZF	Greater (Signed)
jge	~ (SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF) ZF	Less or Equal (Signed)
ja	~CF & ~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Procedures and Stack

Mechanisms in Procedures

- ▶ Passing control
 - To beginning of procedure code
 - Back to return point
- ▶ Passing data
 - Procedure arguments
 - Return value
- ▶ Memory management
 - Allocate during procedure execution
 - Deallocate upon return
- ▶ Mechanisms all implemented with machine instructions
- ▶ x86-64 implementation of a procedure uses only those mechanisms required

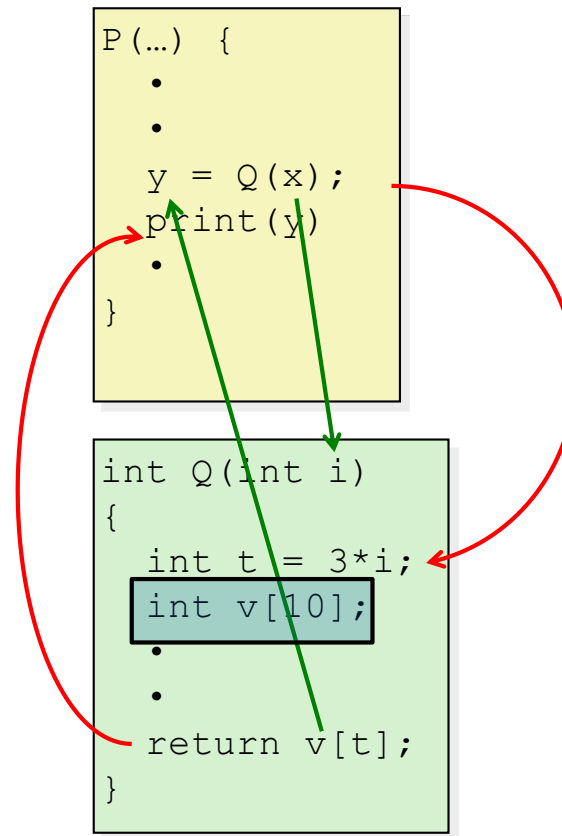
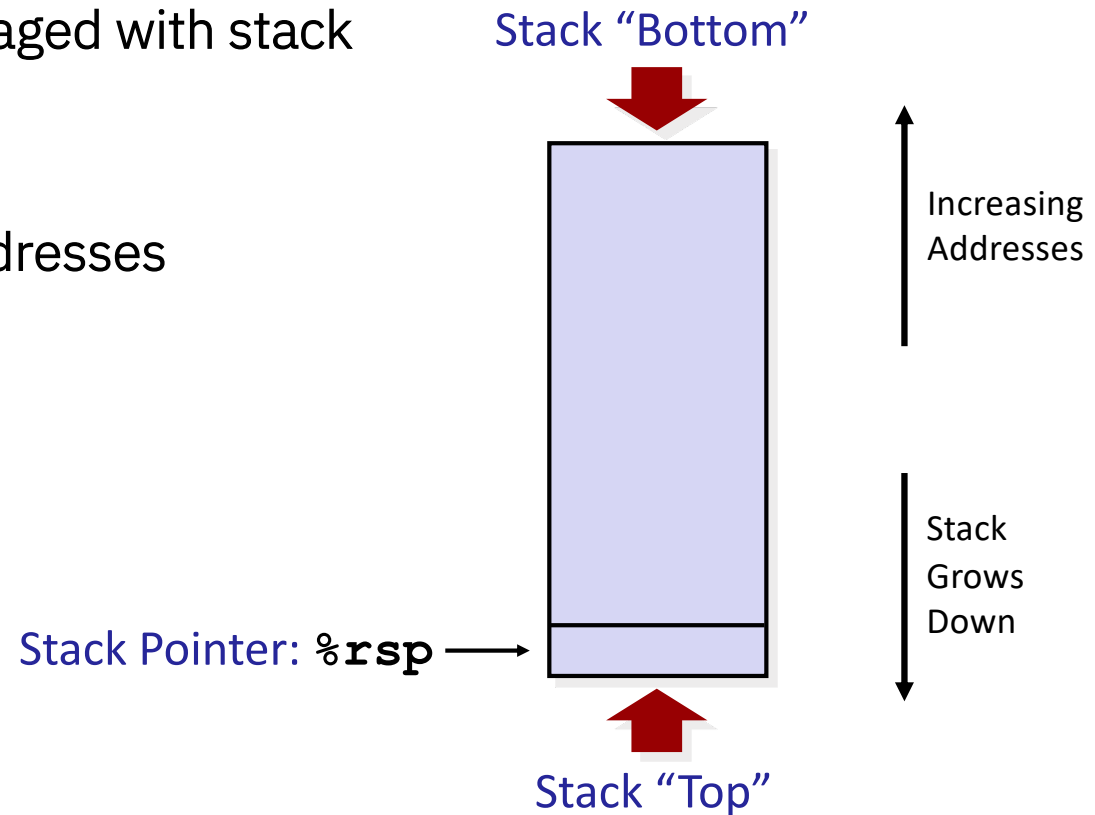


Table of Contents

- ▶ Procedures
 - Stack Structure
 - Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
 - Illustration of Recursion

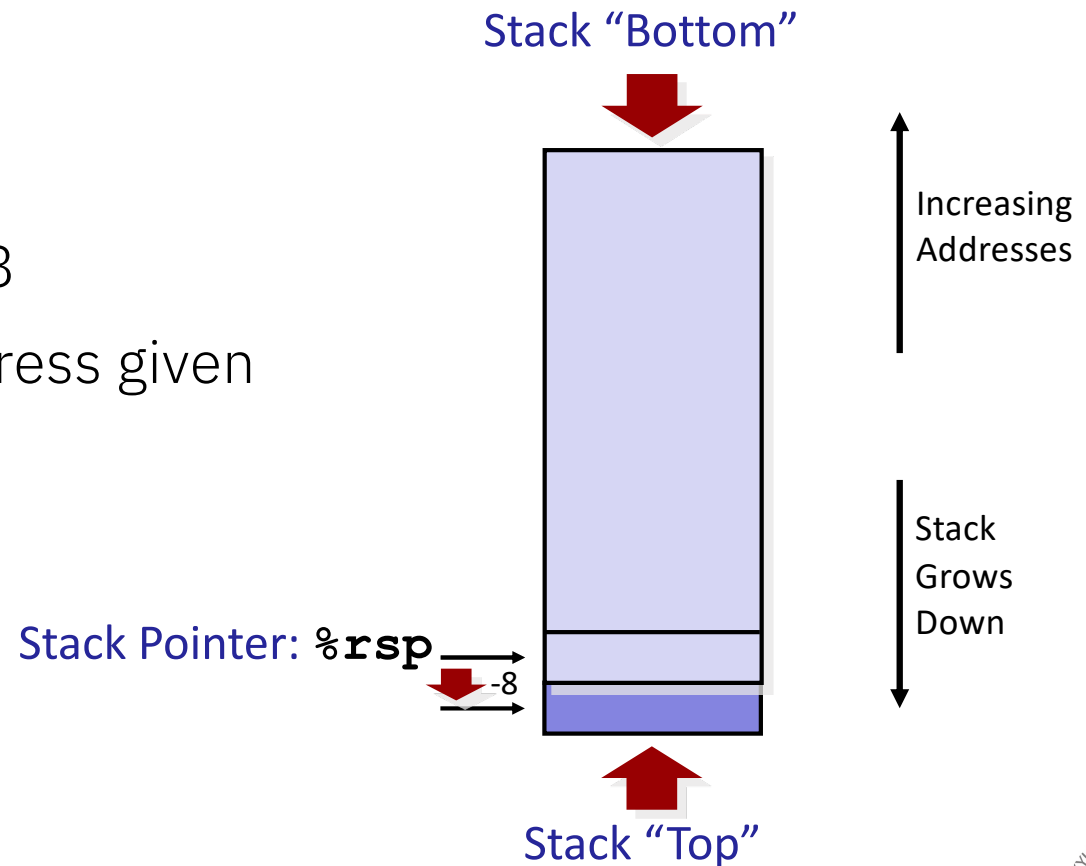
x86-64 Stack

- ▶ Region of memory managed with stack discipline
- ▶ Grows toward lower addresses
- ▶ Register `%rsp` contains lowest stack address
 - address of “top” element



x86-64 Stack: Push

- ▶ **pushq Src**
 - Fetch operand at Src
 - Decrement **%rsp** by 8
 - Write operand at address given by **%rsp**



x86-64 Stack: Pop

► **popq Dest**

- Read value at address given by **%rsp**
- Increment **%rsp** by 8
- Store value at Dest (must be register)

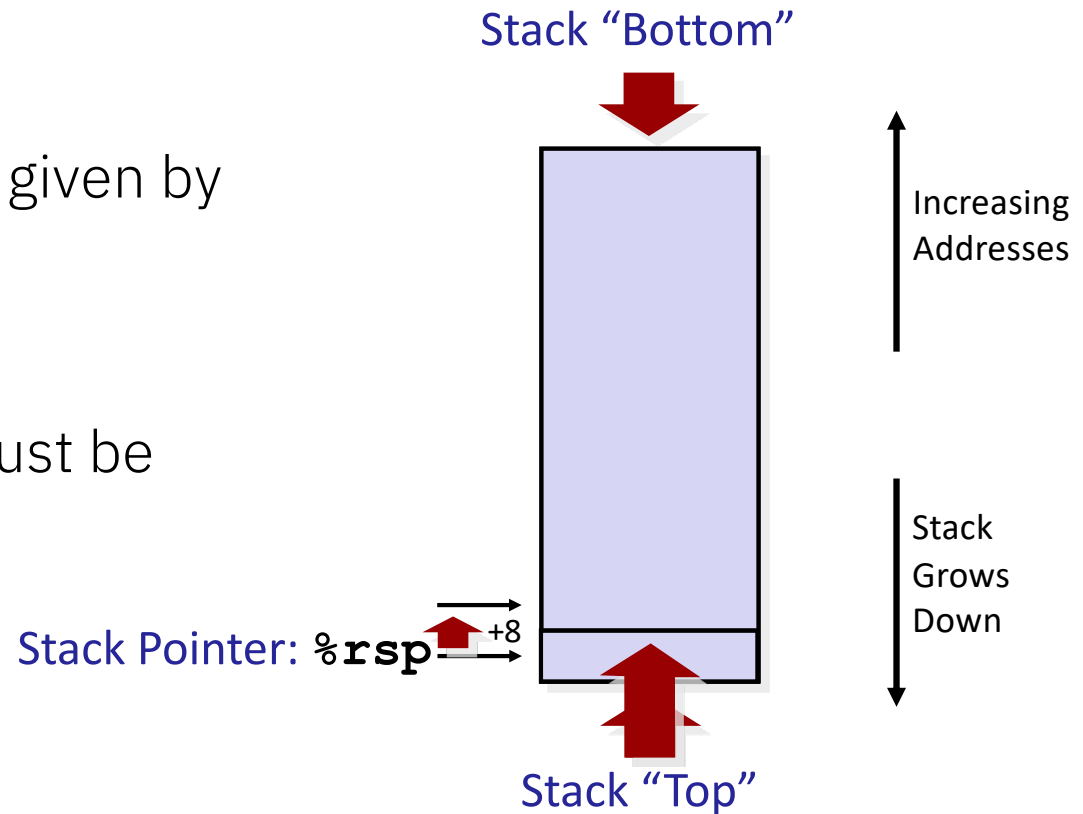


Table of Contents

- ▶ Procedures
 - Stack Structure
 - Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
 - Illustration of Recursion

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
```

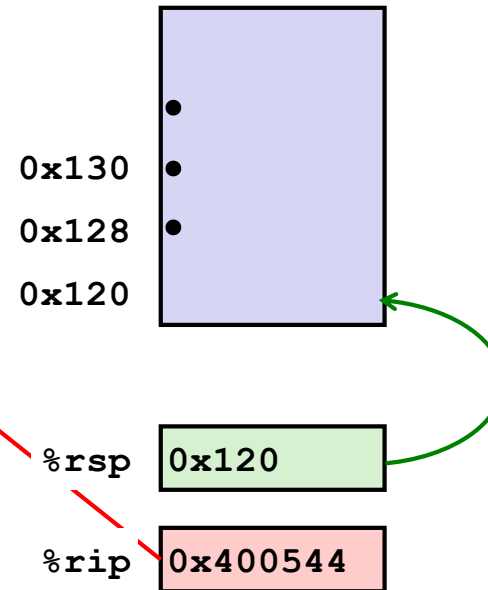
Procedure Control Flow

- ▶ Use stack to support procedure call and return
- ▶ Procedure call: **call label**
 - Push return address on stack
 - Jump to **label**
- ▶ Return address:
 - Address of the next instruction right after call
 - Example from disassembly
- ▶ Procedure return: **ret**
 - Pop address from stack
 - Jump to address

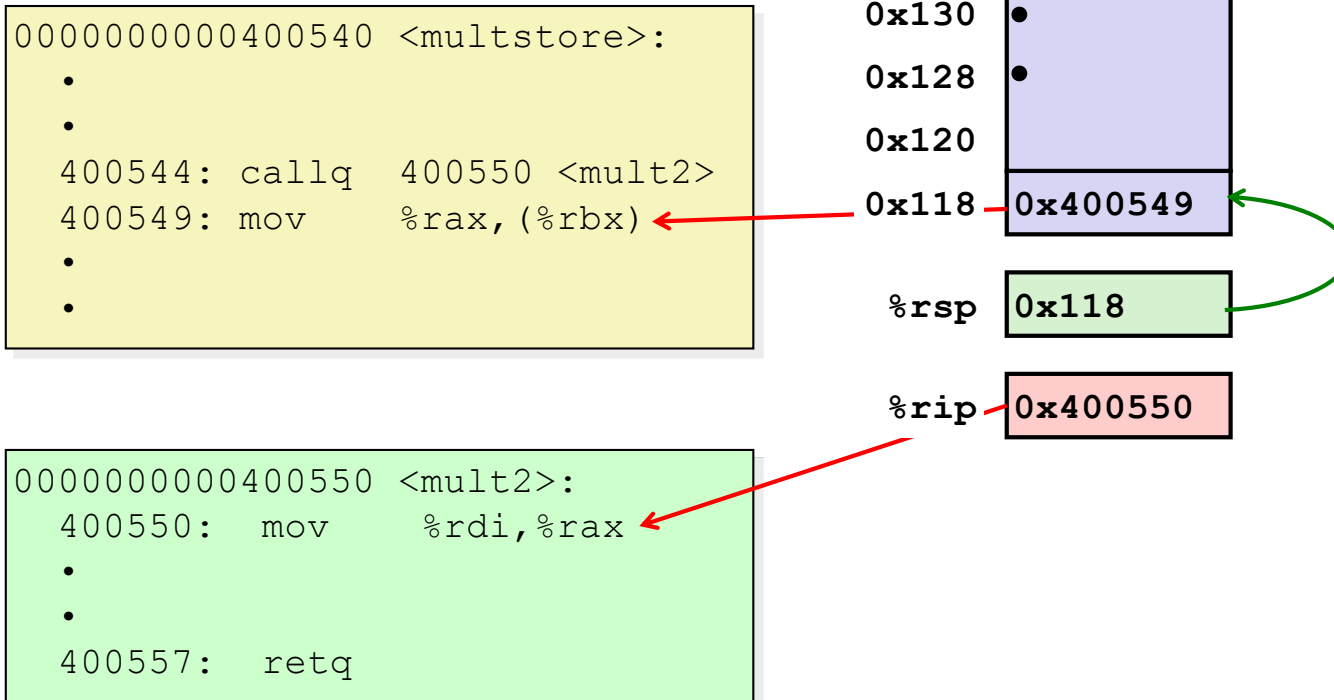
Control Flow Example #1

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

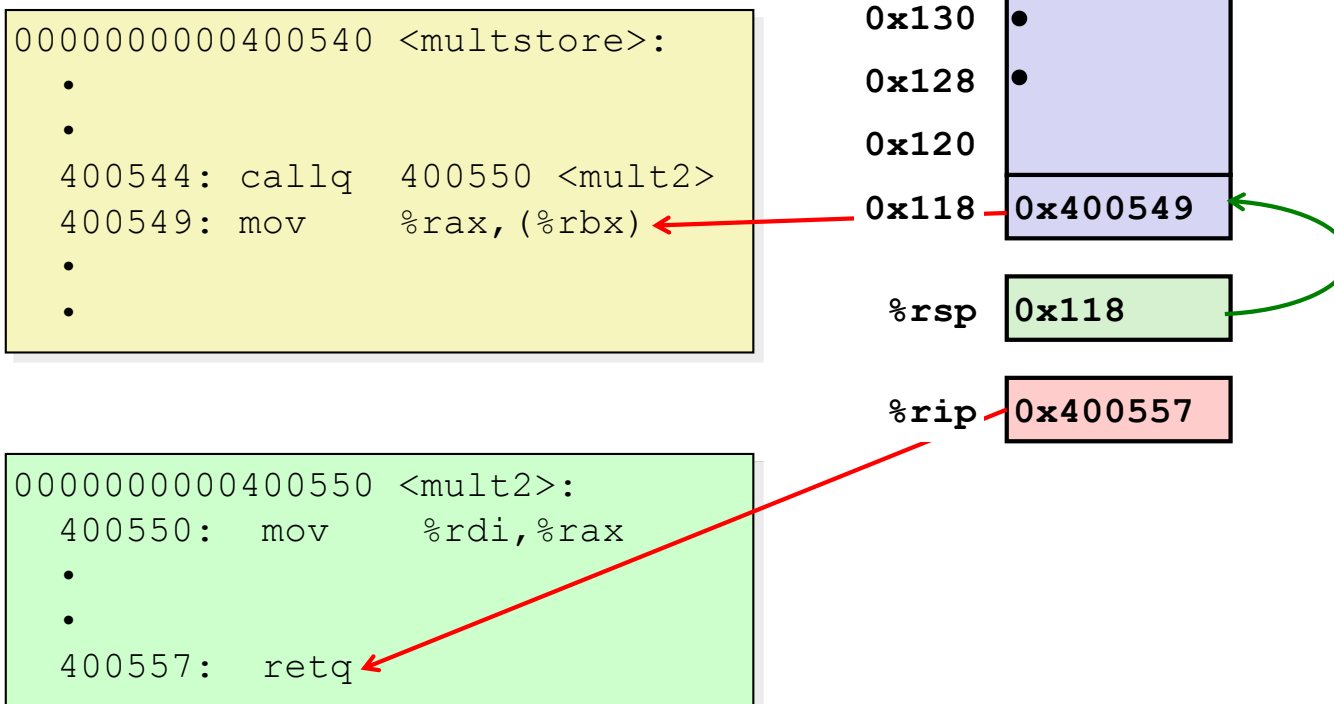
```
0000000000400550 <mult2>:  
400550: mov    %rdi, %rax  
.  
.  
400557: retq
```



Control Flow Example #2



Control Flow Example #3



Control Flow Example #4

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

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

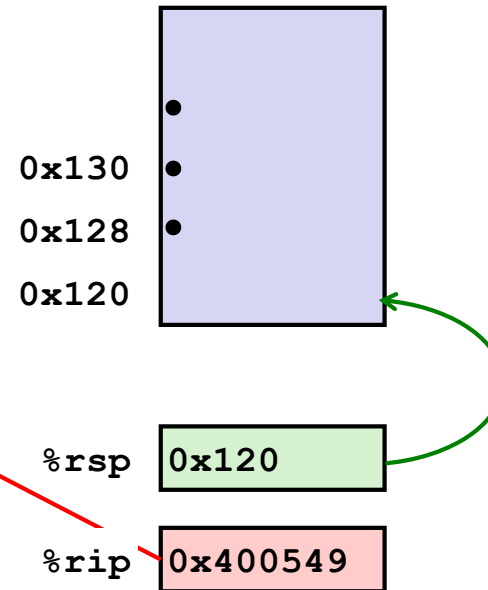
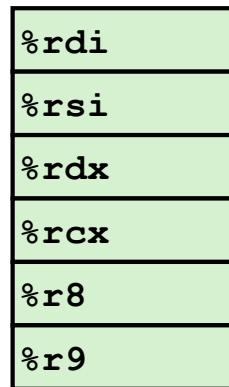


Table of Contents

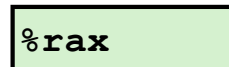
- ▶ Procedures
 - Stack Structure
 - Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
 - Illustrations of Recursion & Pointers

Procedure Data Flow

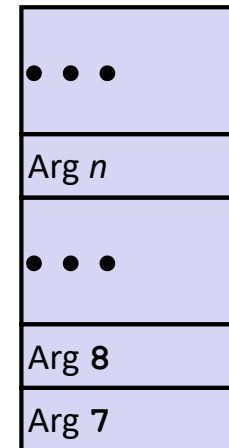
Registers: First 6 arguments



Return Value



Stack



- ▶ Only allocate stack space when needed

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
```

Table of Contents

- ▶ Procedures
 - Stack Structure
 - Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
 - Illustration of Recursion

Stack-Based Languages

- ▶ Languages that support recursion
 - e.g., C, Pascal, Java
 - Code must be “**Reentrant**”
 - Multiple simultaneous instantiations of single procedure
 - Need some place to store state of each instantiation
 - Arguments
 - Local variables
 - Return pointer
- ▶ Stack discipline
 - State for given procedure needed for limited time
 - From when called to when return
 - Callee returns before caller does
- ▶ Stack allocated in **Frames**
 - state for single procedure instantiation

Call Chain Example

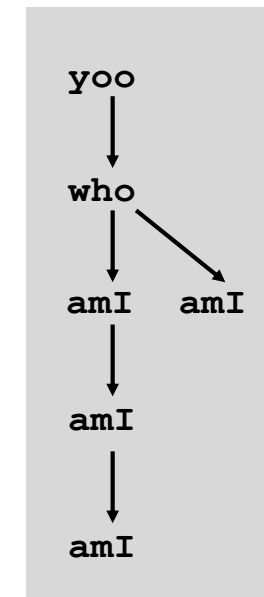
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

```
who (...)  
{  
  . . .  
  amI ();  
  . . .  
  amI ();  
  . . .  
}
```

```
amI (...)  
{  
  .  
  .  
  amI ();  
  .  
  .  
}
```

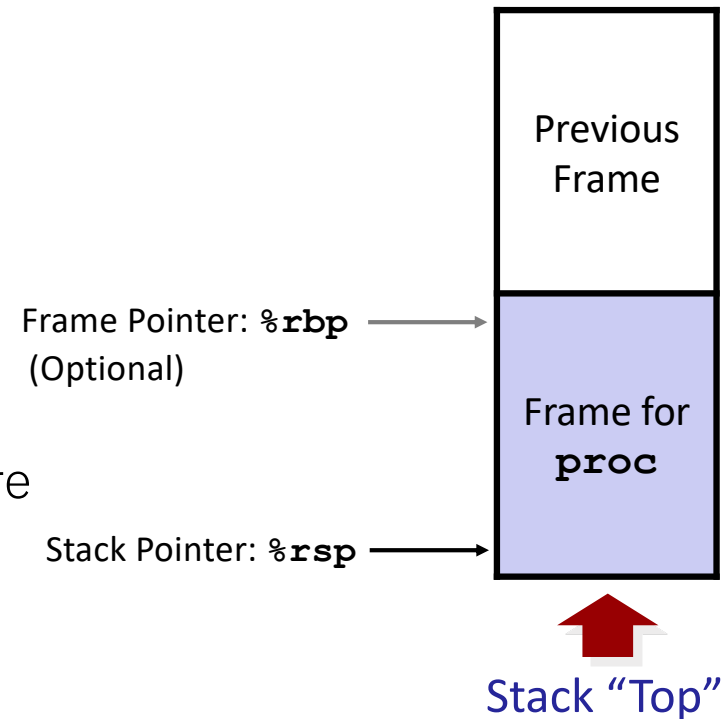
Procedure **amI ()** is recursive

Example
Call Chain

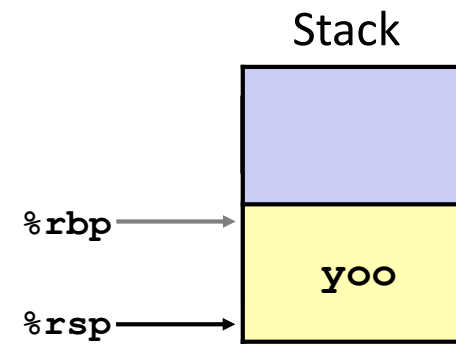
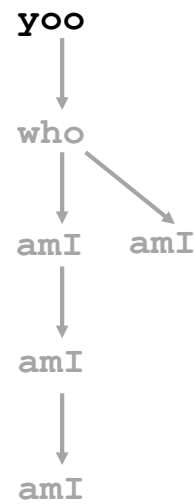
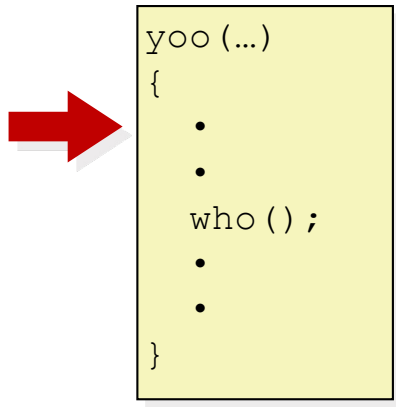


Stack Frames

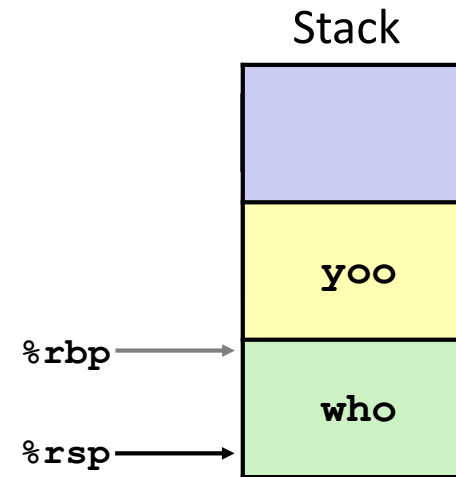
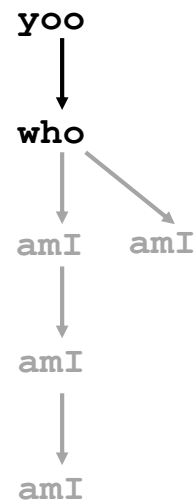
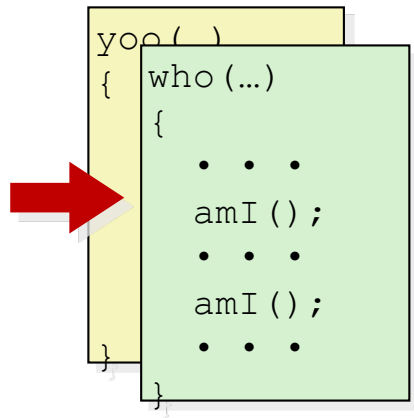
- ▶ Contents
 - Return information
 - Local storage (if needed)
 - Temporary space (if needed)
- ▶ Management
 - Space allocated when enter procedure
 - “Set-up” code
 - Includes push by **call** instruction
 - Deallocated when return
 - “Finish” code
 - Includes pop by **ret** instruction



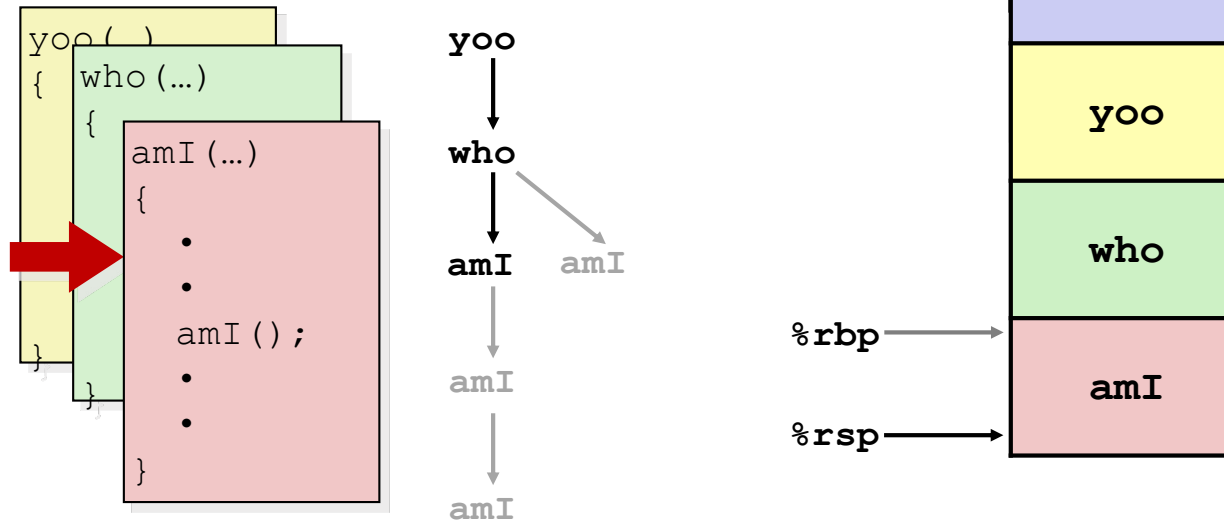
Example



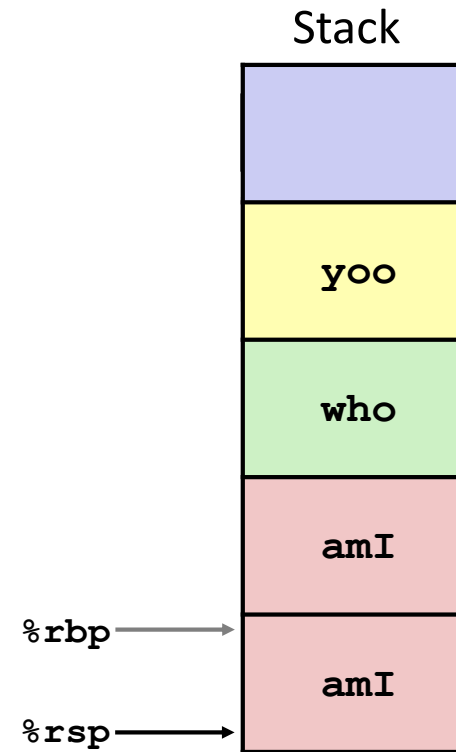
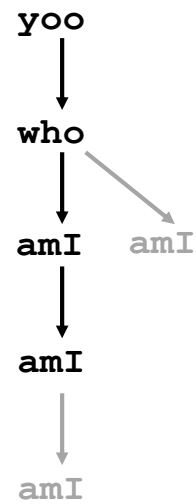
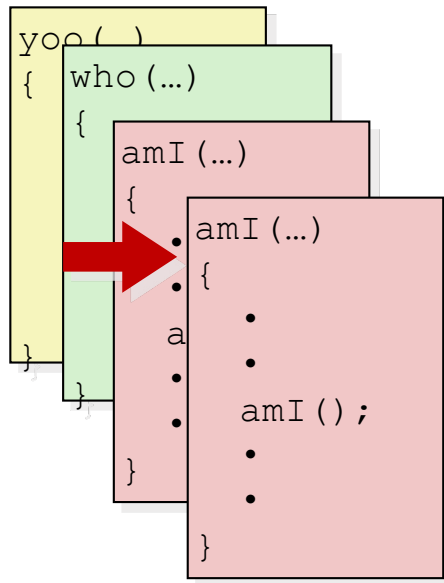
Example



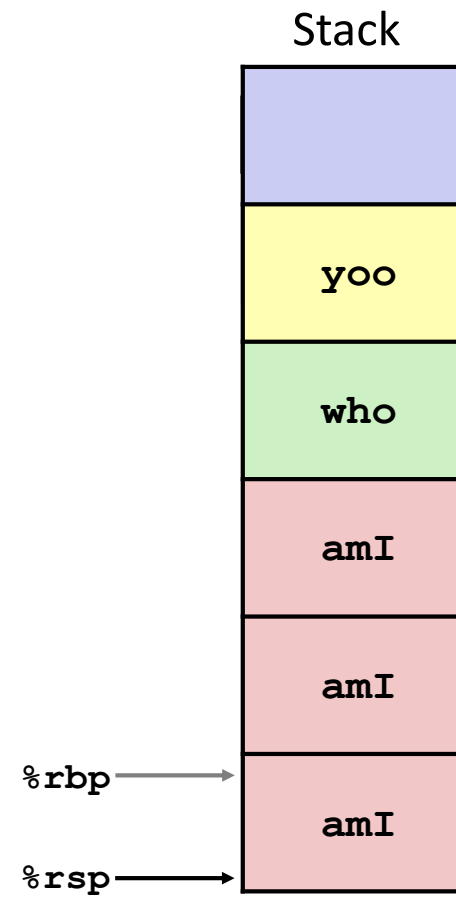
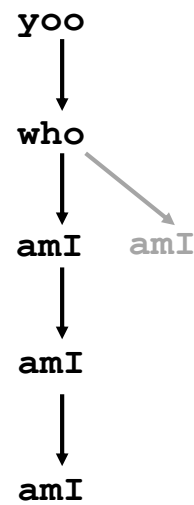
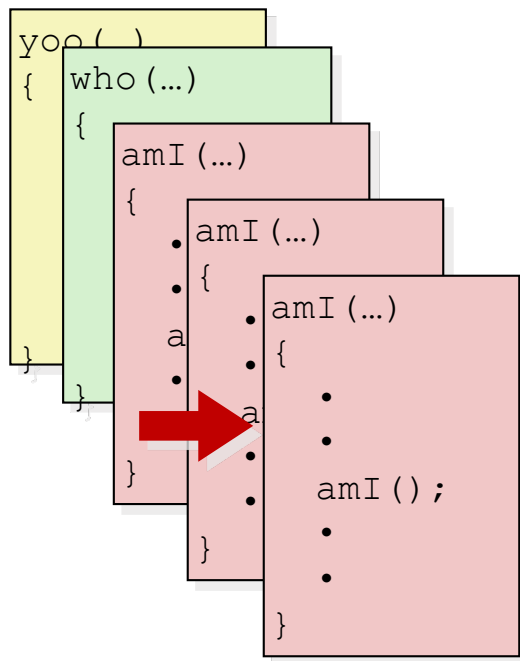
Example



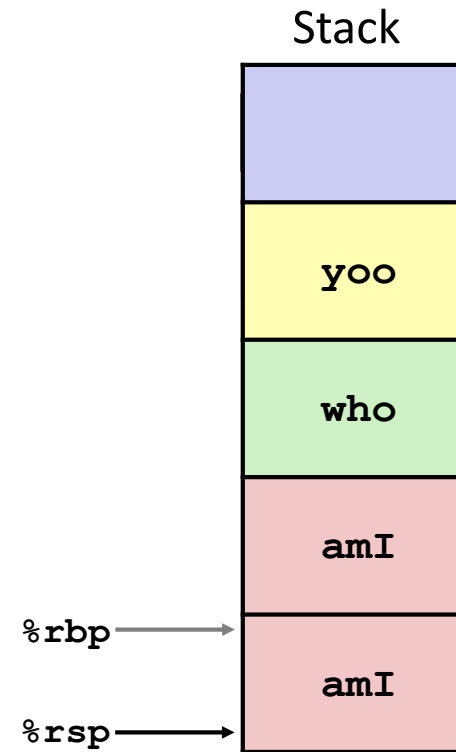
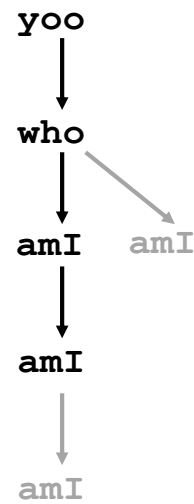
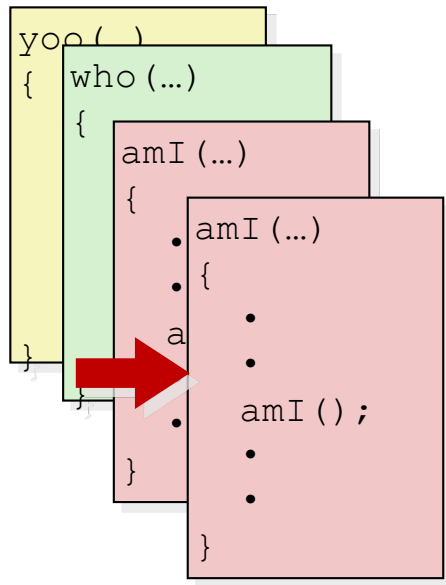
Example



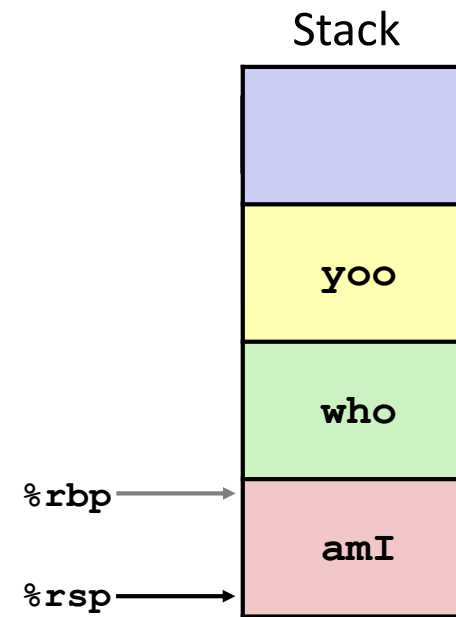
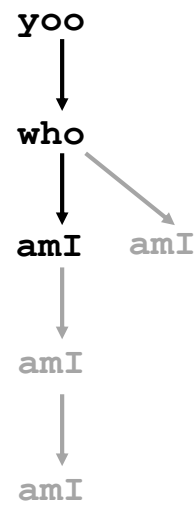
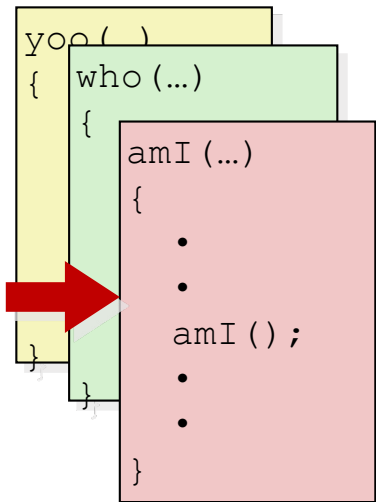
Example



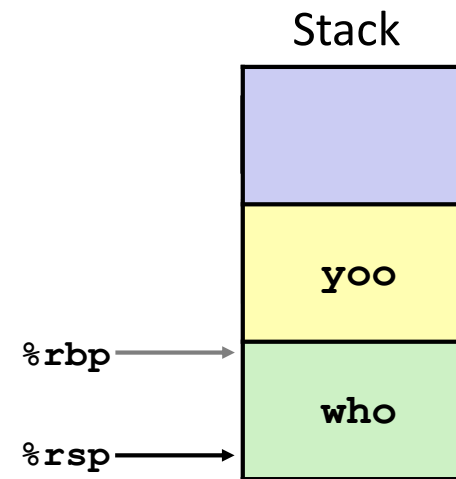
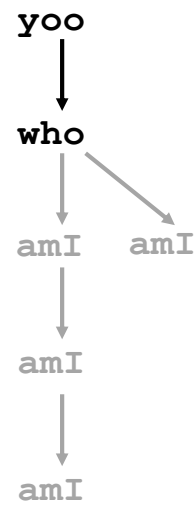
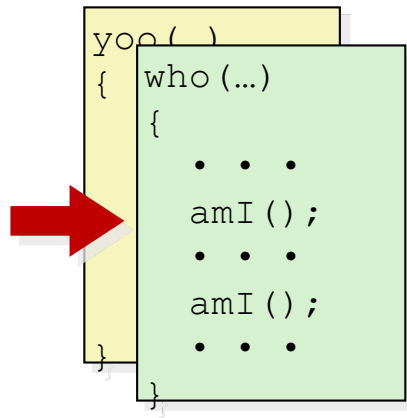
Example



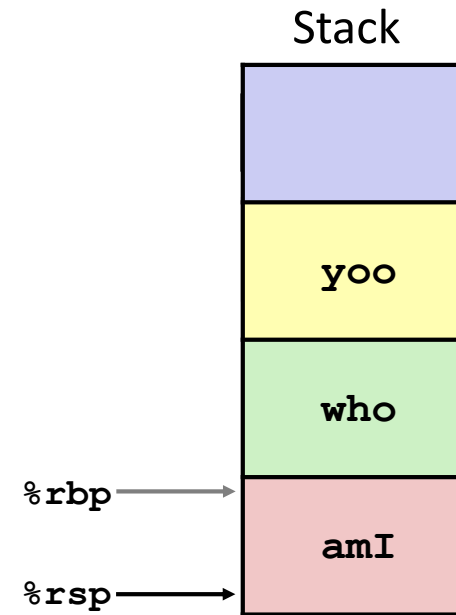
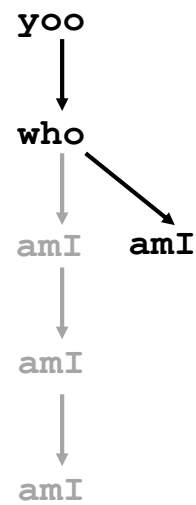
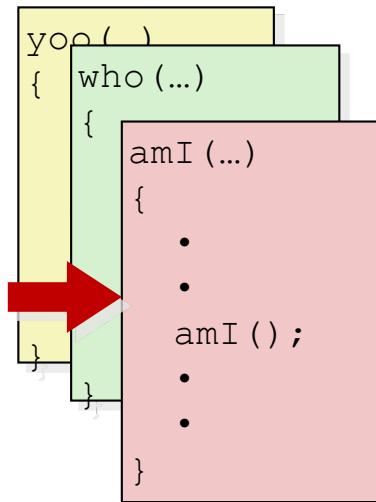
Example



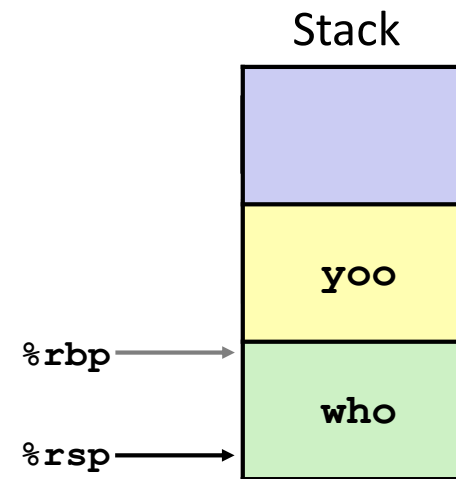
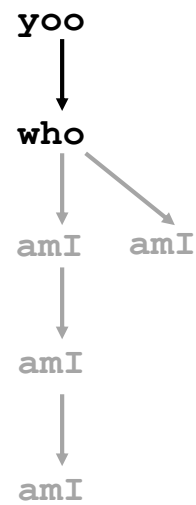
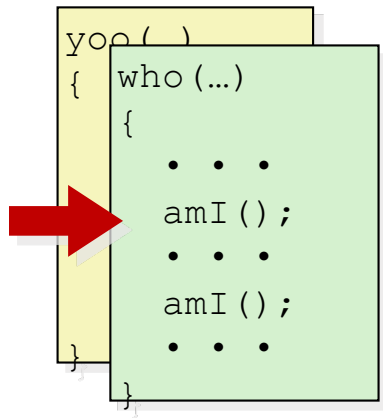
Example



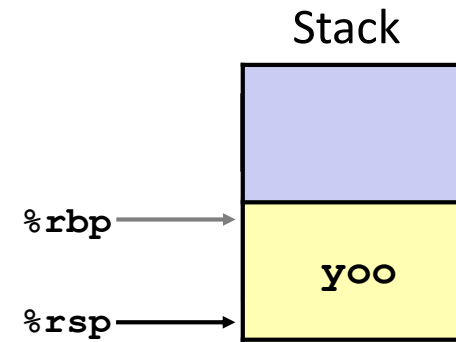
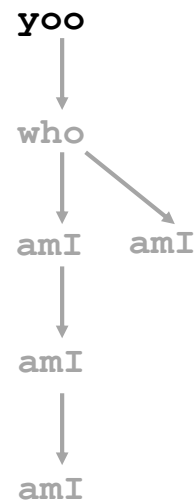
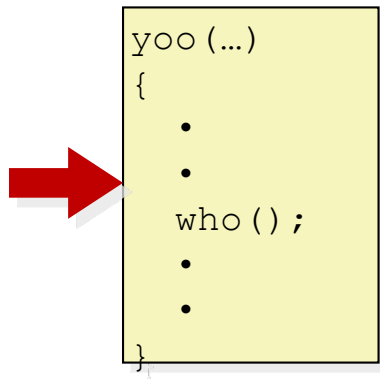
Example



Example



Example



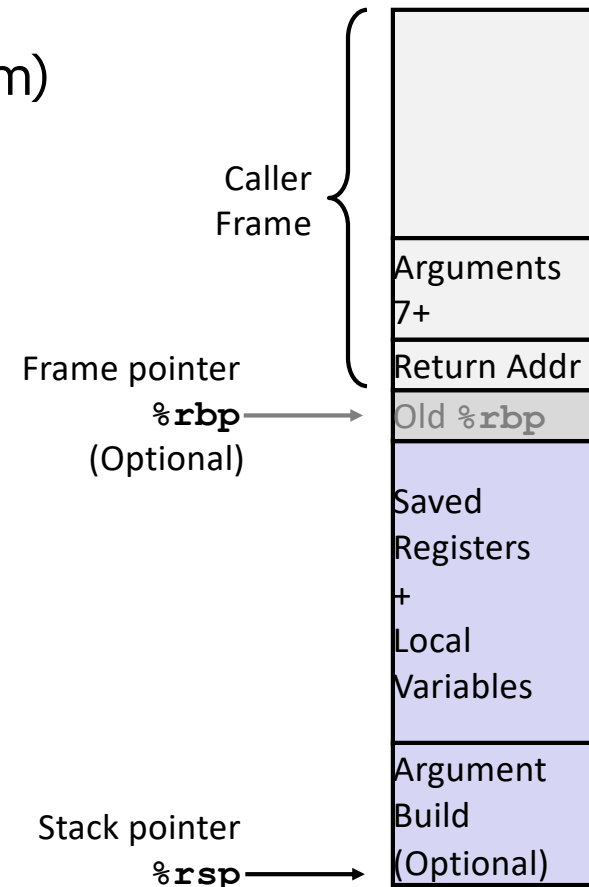
x86-64/Linux Stack Frame

▶ Current Stack Frame (“Top” to Bottom)

- “Argument build:”
Parameters for function about to call
- Local variables
If can’t keep in registers
- Saved register context
- Old frame pointer (optional)

▶ Caller Stack Frame

- Return address
 - Pushed by **call** instruction
- Arguments for this call



Example: `incr`

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

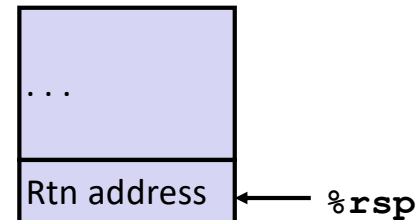
```
incr:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Register	Use(s)
%rdi	Argument <code>p</code>
%rsi	Argument <code>val</code> , <code>y</code>
%rax	<code>x</code> , Return value

Example: Calling `incr` #1

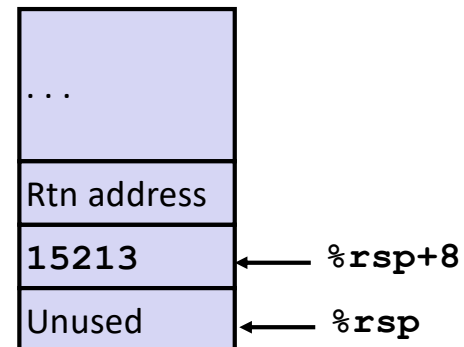
```
long call_incr() {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return v1+v2;  
}
```

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
```

Resulting Stack Structure

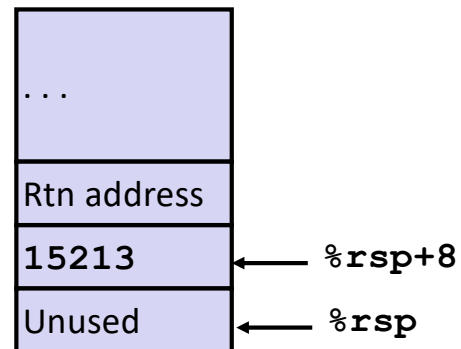


Example: Calling `incr` #2

```
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
```

Stack Structure



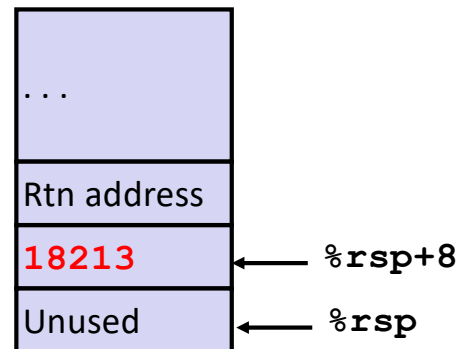
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `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
```

Stack Structure



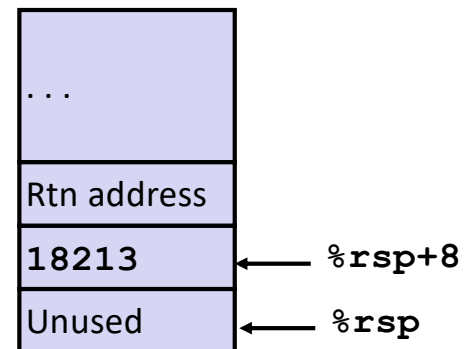
Register	Use(s)
%rdi	&v1
%rsi	3000

Example: Calling `incr` #4

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

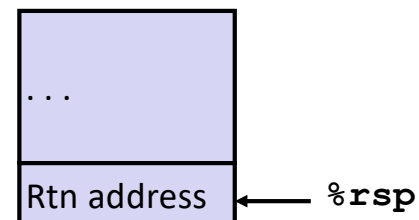
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Structure



Register	Use(s)
<code>%rax</code>	Return value

Updated Stack Structure

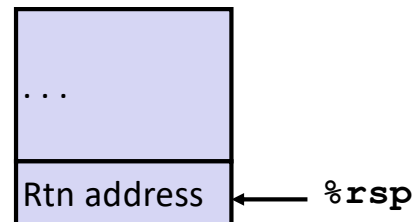


Example: Calling `incr` #5

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

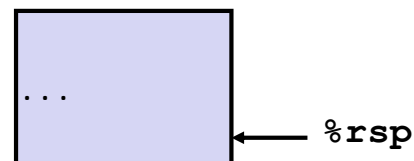
```
call_incr:  
    subq    $16, %rsp  
    movq    $15213, 8(%rsp)  
    movl    $3000, %esi  
    leaq    8(%rsp), %rdi  
    call    incr  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



Register Saving Conventions

- ▶ When procedure **yoo** calls **who**:
 - **yoo** is the **caller**
 - **who** is the **callee**
- ▶ Can register be used for temporary storage?

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

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

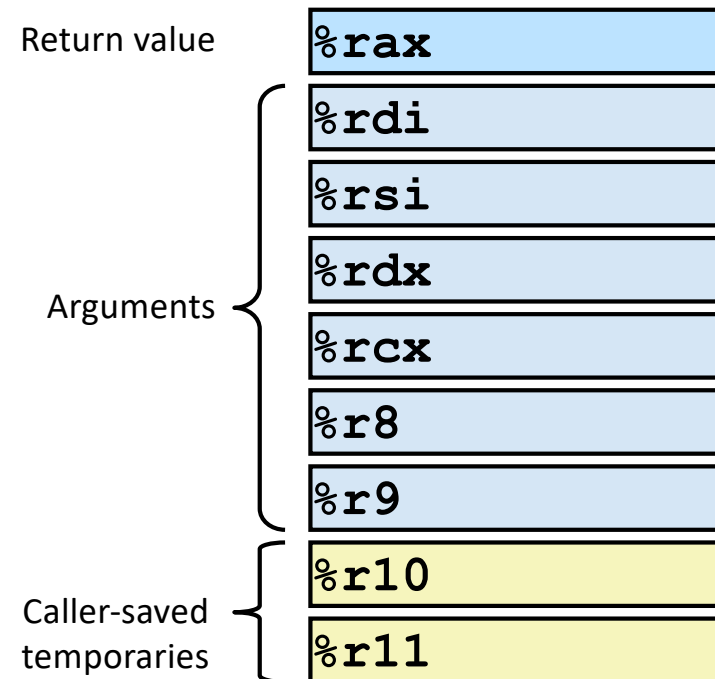
- Contents of register **%rdx** overwritten by **who**
- This could be trouble → something should be done!
 - Need some coordination

Register Saving Conventions

- ▶ When procedure **yoo** calls **who**:
 - **yoo** is the **caller**
 - **who** is the **callee**
- ▶ Can register be used for temporary storage?
- ▶ Conventions
 - “**Caller Saved**”
 - Caller saves temporary values in its frame before the call
 - “**Callee Saved**”
 - Callee saves temporary values in its frame before using
 - Callee restores them before returning to caller

x86-64 Linux Register Usage #1

- ▶ **%rax**
 - Return value
 - Also caller-saved
 - Can be modified by procedure
- ▶ **%rdi, ..., %r9**
 - Arguments
 - Also caller-saved
 - Can be modified by procedure
- ▶ **%r10, %r11**
 - Caller-saved
 - Can be modified by procedure



x86-64 Linux Register Usage #2

- ▶ **%rbx, %r12, %r13, %r14**

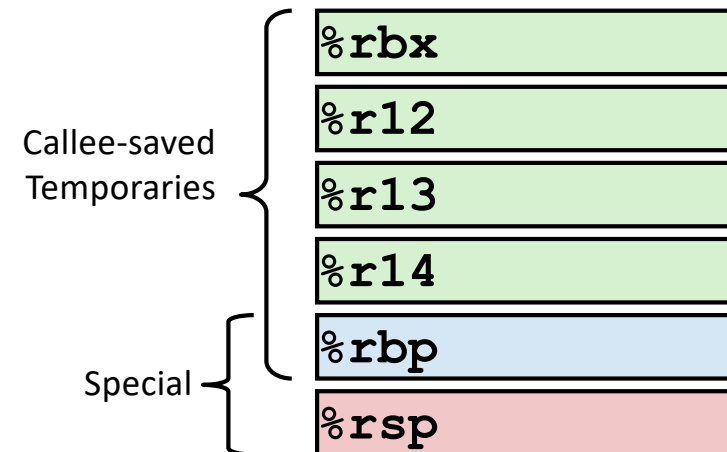
- Callee-saved
- Callee must save & restore

- ▶ **%rbp**

- Callee-saved
- Callee must save & restore
- May be used as frame pointer
- Can mix & match

- ▶ **%rsp**

- Special form of callee save
- Restored to original value upon exit from procedure

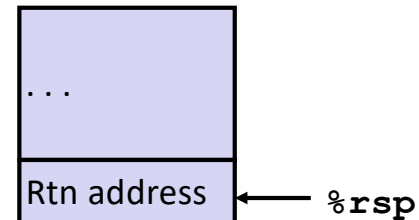


Callee-Saved Example #1

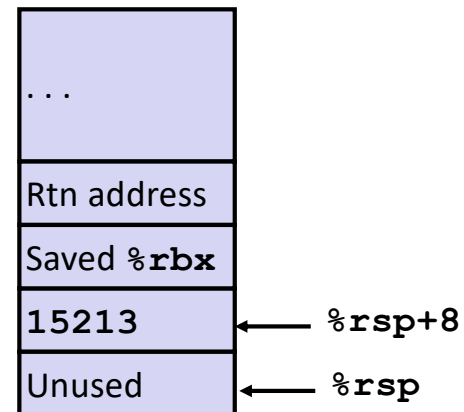
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
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
```

Initial Stack Structure



Resulting Stack Structure

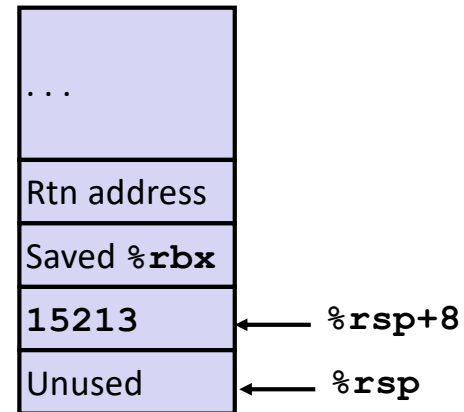


Callee-Saved Example #2

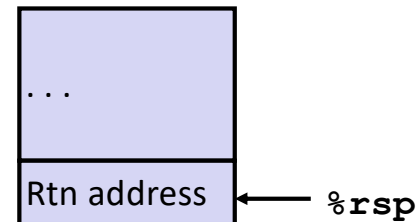
```
long call_incr2(long x) {  
    long v1 = 15213;  
    long v2 = incr(&v1, 3000);  
    return x+v2;  
}
```

```
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
```

Resulting Stack Structure



Pre-return Stack Structure



That's it for Today

- ▶ Homework2:
 - Do textbook HW problems 3.55 3.59
- ▶ "Bomb Lab" will be out this week
- ▶ Your Task
 - Reverse engineer the given program and find out the right combination of inputs that can defuse the bomb

Table of Contents

- ▶ Procedures
 - Stack Structure
 - Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
 - Illustration of Recursion

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
```

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
```

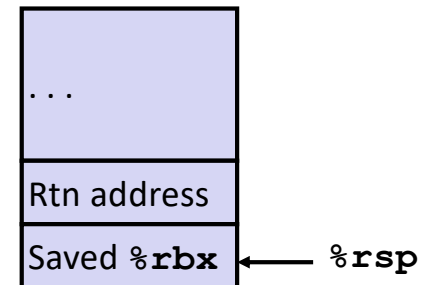
Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

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
```

Register	Use(s)	Type
%rdi	x	Argument



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
```

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

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
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

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
```

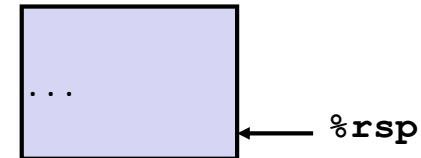
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

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
```

Register	Use(s)	Type
%rax	Return value	Return value



Observations About Recursion

- ▶ **Handled Without Special Consideration**
 - Stack frames mean that each function call has private storage
 - Saved registers & local variables
 - Saved return pointer
 - Register saving conventions prevent one function call from corrupting another's data
 - Unless the C code explicitly does so (e.g., buffer overflow in Lecture 9)
 - Stack discipline follows call / return pattern
 - If P calls Q, then Q returns before P
 - Last-In, First-Out
- ▶ **Also works for mutual recursion**
 - P calls Q; Q calls P

x86-64 Procedure Summary

► Important Points

- Stack is the right data structure for procedure call / return
 - If P calls Q, then Q returns before P

► Recursion (& mutual recursion) handled by normal calling

conventions

- Can safely store values in local stack frame and in callee-saved registers (Optional)
 - Put function arguments at top of stack
 - Result return in **%rax**
-
- Pointers are addresses of values
 - On stack or global

