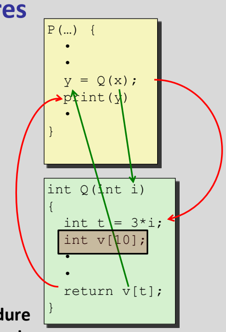
Machine-Level Programming III: Procedures

CSci 2021: Machine Architecture and Organization
February 26th-28th, 2020
Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O'Hallaron

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**

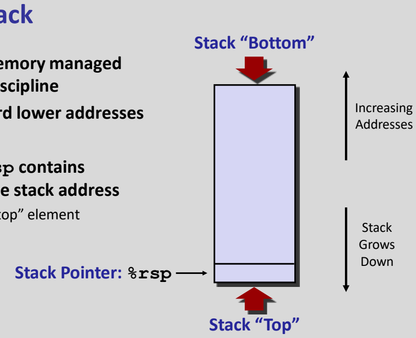


These Slides

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

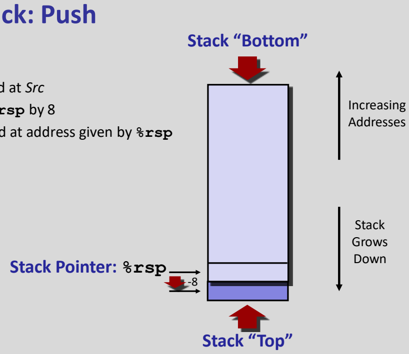
x86-64 Stack

- **Region of memory managed with stack discipline**
- **Grows toward lower addresses**
- **Register `%rsp` contains lowest in-use stack address**
  - address of "top" element



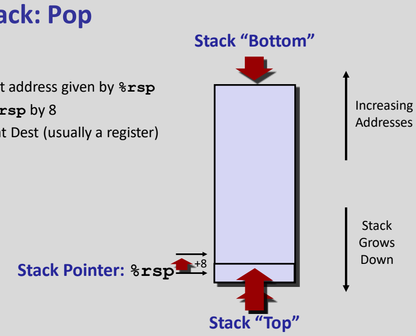
x86-64 Stack: Push

- **`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 (usually a register)



Today

- **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
```

Stack: 0x130, 0x128, 0x120 (• • •)

%rsp: 0x120

%rip: 0x400544

Control Flow Example #2

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

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

Stack: 0x130, 0x128, 0x120 (• • •), 0x118: 0x400549

%rsp: 0x118

%rip: 0x400550

Control Flow Example #3

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

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

Stack: 0x130, 0x128, 0x120 (• • •), 0x118: 0x400549

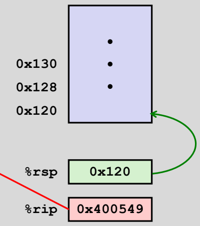
%rsp: 0x118

%rip: 0x400557

Control Flow Example #4

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

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



Today

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

Procedure Data Flow

Registers

- **First 6 arguments**

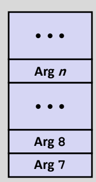
%rdi
%rsi
%rdx
%rcx
%r8
%r9

"Diane's
silk
dress
costs
$8 9"
-- Geoff Kuenning, HMC

- **Return value**

%rax

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
```


Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - **Interlude: binary-level GDB**
    - Managing local data
  - Illustrations of Recursion & Pointers

Overview: GDB without source code

- GDB can also be used just at the instruction level

Source-level GDB	Binary-level GDB
step/next	stepi/nexti
break <line number>	break *<address>
list	disas
print <variable>	print with registers & casts
print <data structure>	examine
info local	info reg
software watch	hardware watch

Disassembly and stepping

- The `disas` command prints the disassembly of instructions
  - Give a function name, or defaults to current function, if available
  - Or, supply range of addresses <start>,<end> or <start>,+<length>
  - If you like TUI mode, "`layout asm`"
  - Shortcut for a single instruction: `x/i` <addr>, `x/i $rip`
  - `disasm/r` shows raw bytes too
- `stepi` and `nexti` are like `step` and `next`, but for instructions
  - Can be abbreviated `si` and `ni`
  - `stepi` goes into called functions, `nexti` stays in current one
  - `continue`, `return`, and `finish` work as normal

Binary-level breakpoints

- All breakpoints are actually implemented at the instruction level
  - `info br` will show addresses of all breakpoints
  - Sometimes multiple instructions correspond to one source location
- To break at an instruction, use `break *`<address>
  - Address usually starts with 0x for hex
- The `until` command is like a temporary breakpoint and a continue
  - Works the same on either source or binary

Binary-level printing

- The `print` command still mostly uses C syntax, even when you don't have source
  - Registers available with `$` names, like `$rax`, `$rip`
  - Often want `p/x`, for hex
- Use casts to indicate types
  - `p (char)$r10`
  - `p (char *)$rbx`
- Use casts and dereferences to access memory
  - `p *(int *)$rcx`
  - `p *(char **)$r8`
  - `p *((int*)$rbx + 1)`
  - `p *(int*)($rbx + 4)`

Examining memory

- The `examine` (`x`) command is a low-level tool for printing memory contents
  - No need to use cast notation
- `x/`<format> <address>
  - Format can include repeat count (e.g., for array)
  - Many format letters, most common are `x` for hex or `d` for decimal
  - Size letter `b/h/w/g` means 1/2/4/8 bytes
- Example: `x/20xg 0x404100`
  - Prints first 20 elements of an array of 64-bit pointers, in hex

More useful printing commands

- `info reg` prints contents of all integer registers, flags
  - In TUI: `layout reg`, will highlight updates
  - Float and vector registers separate, or use `info all-reg`
- `info frame` prints details about the current stack frame
  - For instance, "saved rip" means the return address
- `backtrace` still useful, but shows less information
  - Just return addresses, maybe function names

Hardware watchpoints

- To watch memory contents, use print-like syntax with addresses
  - watch *(int *)0x404170
- GDB's "Hardware watchpoint" indicates a different implementation
  - Much faster than software
  - But limited in number
  - Limited to watching memory locations only
- Watching memory is good for finding memory corruption

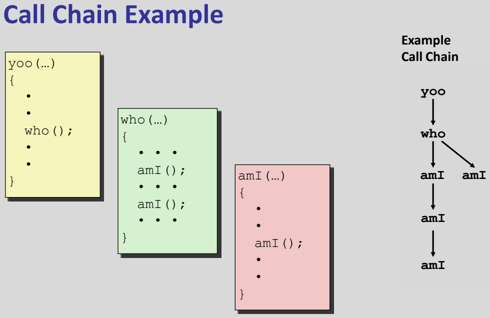
Today

- **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

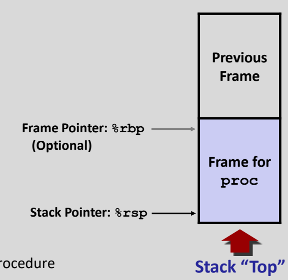
Call Chain Example



Procedure amI() is recursive

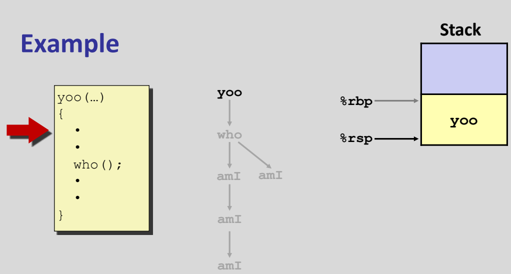
Stack Frames

- **Contents**
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)

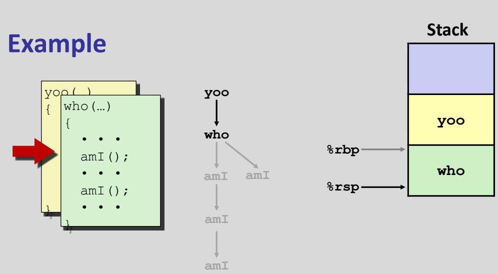


- **Management**
  - Space allocated when enter procedure
    - "Set-up" code, also called "prolog"
    - Includes push by `call` instruction
  - Deallocated when return
    - "Finish" code, also called "epilog"
    - Includes pop by `ret` instruction

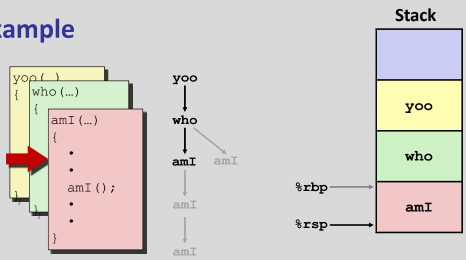
Example



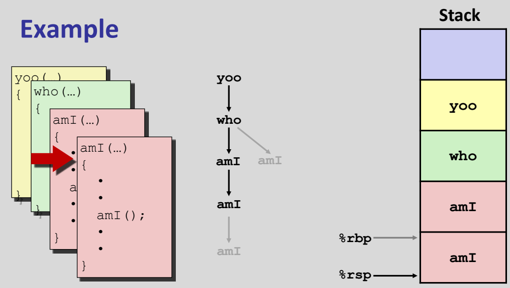
Example



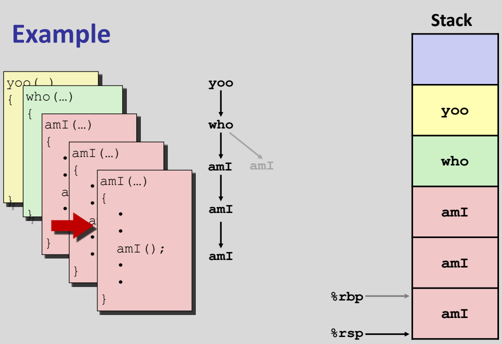
Example



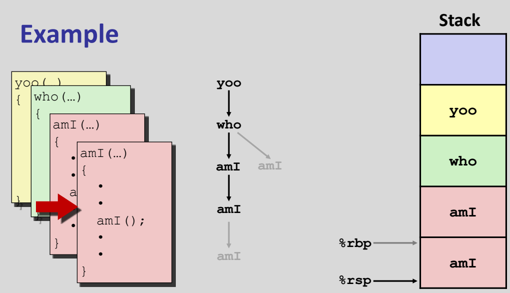
Example



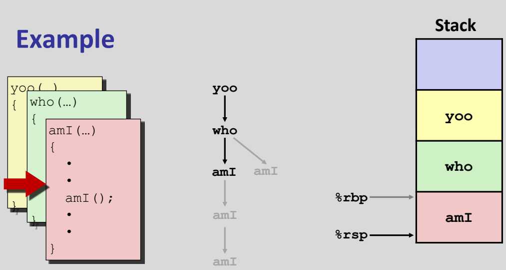
Example



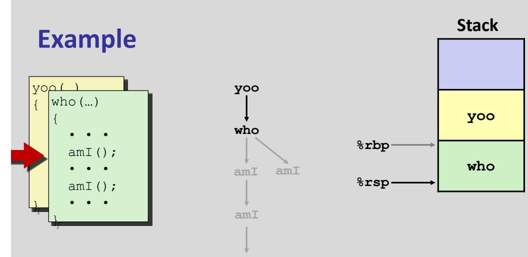
Example



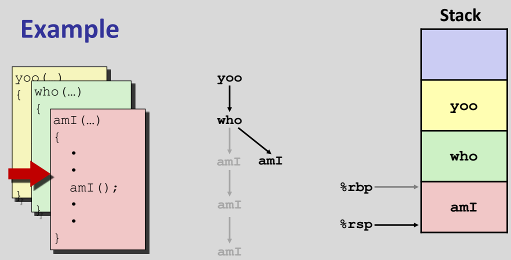
Example



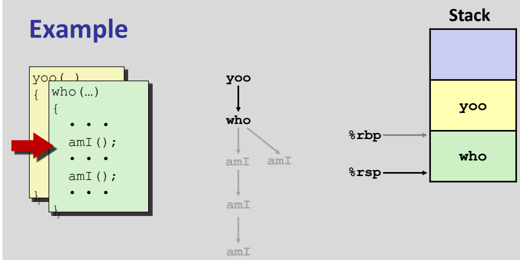
Example



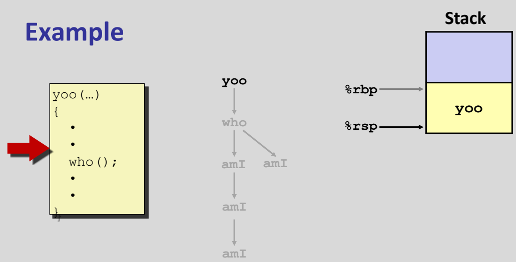
Example



Example

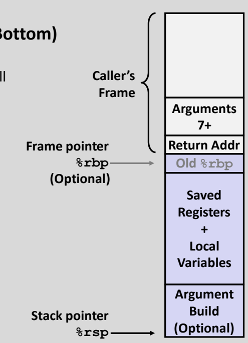


Example



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;
}
```

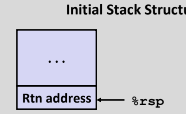
```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val`, `y`
`%rax`	`x`, Return value

Example: Calling `incr` #1

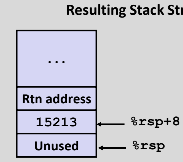
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

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

```
| ...         |
| Rtn address |
| 15213       | <-- %rsp+8
| Unused      | <-- %rsp
```

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

```
| ...         |
| Rtn address |
| 18213       | <-- %rsp+8
| Unused      | <-- %rsp
```

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

```
| ...         |
| Rtn address |
| 18213       | <-- %rsp+8
| Unused      | <-- %rsp
```

Register	Use(s)
%rax	Return value

Updated Stack Structure

```
| ...         |
| Rtn address | <-- %rsp
```


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

```
| ...         |
| Rtn address | <-- %rsp
```

Register	Use(s)
%rax	Return value

Final Stack Structure

```
| ...         |
              <-- %rsp
```


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", a.k.a. "scratch"*
    - Caller saves temporary values in its frame before the call
  - *"Callee Saved", a.k.a. "preserved"*
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

x86-64 Linux Register Usage #1 (scratch)

- %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

Return value: %rax

Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9

Caller-saved temporaries: %r10, %r11

x86-64 Linux Register Usage #2 (preserved)

- %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 Temporaries: %rbx, %r12, %r13, %r14

Special: %rbp, %rsp

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

...
Rtn address ← %rsp

Resulting Stack Structure

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

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

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

Pre-return Stack Structure

...
Rtn address ← %rsp

Today

- **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

...
Rtn address
Saved %rbx ← %rsp

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; ret0
```

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

... ← %rsp

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

Discussion interlude

- **Does a recursive function always have to save one or more registers on the stack?**
  - If yes, why?
  - If no, what's an example of a function that doesn't need to?

Recursive function examples

- `void loop(void) { loop(); }`

```
int fact(unsigned n, int prod) {
    if (n == 0)
        return prod;
    else
        return fact(n - 1, n * prod);
}
```

- **But if storing a value across a call, the stack is needed**
  - If caller-save, need to save because callee will use it
  - If callee-save, need to save caller's value
  - Changing the calling convention would not help

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
  - Put function arguments at top of stack
  - Result return in `%rax`
- **Pointers are addresses of values**
  - **On stack or global**