Machine-Level Programming III: Procedures

15-213/18-213: Introduction to Computer Systems
7th Lecture, June 5, 2018

Instructor:
Brian Railing
Reminder: Condition Codes

- **Single bit registers**
  - **CF**  Carry Flag (for unsigned)  **SF**  Sign Flag (for signed)
  - **ZF**  Zero Flag  **OF**  Overflow Flag (for signed)

- **jX and SetX instructions**

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
<td>setsns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
<td>setg</td>
<td>~ (SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jg</td>
<td>~ (SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
<td>setge</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Machine Level Programming – Control

- **C Control**
  - if-then-else
  - do-while
  - while, for
  - switch

- **Assembler Control**
  - Conditional jump
  - Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control

- **Standard Techniques**
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (if-elseif-elseif-else)
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**

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

```c
P(...) { 
    
    y = Q(x);
    print(y)
    
}
```
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**

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

P(...) {
    ...
    y = Q(x);
    print(y)
    ...
}
```
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**
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**

```c
int Q(int i) {
    int t = 3*i;
    int v[10];
    
    return v[t];
}
```
Mechanisms in Procedures

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

- Allocate during procedure execution
  - Deallocate upon return
- **Mechanisms all implemented with machine instructions**
- x86-64 implementation of a procedure uses only those mechanisms required

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

- 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
  - Memory viewed as array of bytes.
  - Different regions have different purposes.
  - (Like ABI, a policy decision)
x86-64 Stack

- Region of memory managed with stack discipline

Stack Pointer: \%rsp

Stack "Top"

Stack "Bottom"
x86-64 Stack

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

Stack Pointer: \%rsp

Stack “Top”

Stack “Bottom”

Increasing Addresses

Stack Grows Down
x86-64 Stack: Push

- `pushq Src`
  - Fetch operand at `Src`
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`
x86-64 Stack: Push

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

Stack Pointer:

Stack “Bottom”

Stack “Top”

Increasing Addresses

Stack Grows Down

x86-64 Stack: Pop

- `popq Dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)
x86-64 Stack: Pop

- `popq Dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)
x86-64 Stack: Pop

- `popq Dest`
  - Read value at address given by `%rsp`
  - Increment `%rsp` by 8
  - Store value at Dest (usually a register)

(The memory doesn’t change, only the value of `%rsp`)
Today

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Illustration of Recursion
```c
long mult2(long a, long b) {
    long s = a * b;
    return s;
}
```

```assembly
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;
}
```

```assembly
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

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

00000000000400550 <mult2>:
  400550: mov %rdi, %rax
  •
  •
  400557: retq
Control Flow Example #3

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

0000000000400550 <mult2>:
  400550: mov %rdi, %rax
  .
  .
  400557: retq
Control Flow Example #4

00000000000400540 <multstore>:
•
•
400544: callq 400550 <mult2>
400549: mov %rax, (%rbx)
•
•

00000000000400550 <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

- Return value
  - %rax

Stack

- Only allocate stack space when needed
  - Arg 7
  - Arg 8
  - Arg n
Data Flow Examples

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

```assembly
0000000000400540 <multstore>:
    # x in %rdi, y in %rsi, dest in %rdx
    ...
0400541: mov %rdx,%rbx       # Save dest
0400544: callq 400550 <mult2> # mult2(x,y)
    # t in %rax
0400549: mov %rax,(%rbx)     # Save at dest
    ...
```

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

```assembly
0000000000400550 <mult2>:
    # a in %rdi, b in %rsi
0400550:  mov %rdi,%rax       # a
0400553:  imul %rsi,%rax      # a * b
    # s in %rax
0400557:  retq               # Return
```
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

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

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

```
stack
```

```
%rbp
%rsp
```

```
yoo
```

```
who
```

```
ami ami
ami
ami
```
Example

```c
yoo() {
    who(...) {
        ...
        amI();
        ...
        amI();
        ...
    }
}
```
Example

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

Stack

%rbp

%rsp

yoo

who

amI

amI

amI
Example

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

Stack

```
<table>
<thead>
<tr>
<th>yoo</th>
<th>%rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>who</td>
<td>%rsp</td>
</tr>
<tr>
<td>amI</td>
<td></td>
</tr>
<tr>
<td>amI</td>
<td></td>
</tr>
</tbody>
</table>
```
Example

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

Stack

```
%rbp
%
%rsp
```
Example

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

Stack
```
%rbp
%rsp
amI
amI
who
yoo
```
Example

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

Stack

yoo

who

amI

%rbp

%rsp
```
Example

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

Stack

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

```c
yoo( )
{
    who (...)
    {
        amI (...)
        {
            •
            •
            amI();
            •
            •
        }
    }
}
```
Example

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

Stack

```
%rbp

%rsp
```

```
\text{who}
```

```
\text{yoo}
```
Example

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

Stack

```
%rbp --> %rsp --> yoo
```

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**

```c
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
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument p</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument val, y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, Return value</td>
</tr>
</tbody>
</table>
Example: Calling `incr #1`

```c
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
```
Example: Calling incr #2

```c
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
- Unused

Register Use(s):
- `%rdi` &v1
- `%rsi` 3000
Example: Calling `incr` #2

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

Aside 1: **movl $3000, %esi**
- Remember, movl -> %exx zeros out high order 32 bits.
- Why use movl instead of movq? 1 byte shorter.

```asm
movl $3000, %esi  
leaq 8(%rsp), %rdi  
call incr  
addq 8(%rsp), %rax  
addq $16, %rsp  
ret
```

<table>
<thead>
<tr>
<th>%rdi</th>
<th>&amp;v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rsi</td>
<td>3000</td>
</tr>
</tbody>
</table>
Example: Calling incr #2

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

Aside 2: `leaq 8(%rsp), %rdi`

- Computes %rsp+8
- Actually, used for what it is meant!

```assembly
leaq 8(%rsp), %rdi
call incr
addq 8(%rsp), %rax
addq $16, %rsp
ret
```
Example: Calling `incr #2`

```c
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
- Unused

### Register Use(s):

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>3000</td>
</tr>
</tbody>
</table>
Example: Calling `incr` #3

```c
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`
- Unused

### Register Use(s)
<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>3000</td>
</tr>
</tbody>
</table>

Example: Calling `incr #4`

```c
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`

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Example: Calling `incr #5a`

```c
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
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

Updated Stack Structure

Stack Structure

- Rtn address
- %rsp
- 18213
- Unused

Register Use(s)
- %rax: Return value
Example: Calling `incr` #5b

```c
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
```
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 \texttt{yoo} calls \texttt{who}:
  - \texttt{yoo} is the \textit{caller}
  - \texttt{who} is the \textit{callee}

- Can register be used for temporary storage?

- Conventions
  - "\textit{Caller Saved}"
    - Caller saves temporary values in its frame before the call
  - "\textit{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
Small Exercise

```c
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4, long a5, long a6, long a7, long a8, long a9) {
    return add5(a0, a1, a2, a3, a4) +
           add5(a5, a6, a7, a8, a9);
}
```

- Where are `a0,..., a9` passed?
  - `rdi, rsi, rdx, rcx, r8, r9, stack`

- Where are `b0,..., b4` passed?
  - `rdi, rsi, rdx, rcx, r8`

- Which registers do we need to save?
  - Ill-posed question. Need assembly.
    - `rbx, rbp, r9` (during first call to `add5`)
Small Exercise

```c
long add5(long b0, long b1, long b2, long b3, long b4) {
    return b0+b1+b2+b3+b4;
}

long add10(long a0, long a1, long a2, long a3, long a4, long a5, long a6, long a7, long a8, long a9) {  
    return add5(a0, a1, a2, a3, a4) +  
        add5(a5, a6, a7, a8, a9);
}
```

```
call add5
```

```
add10:
    pushq   %rbp
    pushq   %rbx
    movq    %r9, %rbp
    call    add5
    movq    %rax, %rbx
    movq    48(%rsp), %r8
    movq    40(%rsp), %rcx
    movq    32(%rsp), %rdx
    movq    24(%rsp), %rsi
    movq    %rbp, %rdi
    call    add5
    addq    %rbx, %rax
    popq    %rbx
    popq    %rbp
    ret
```

```
add5:
    addq    %rsi, %rdi
    addq    %rdi, %rdx
    addq    %rdx, %rcx
    leaq    (%rcx,%r8), %rax
    ret
```
Callee-Saved Example #1

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

- X comes in register `%rdi`.
- We need `%rdi` for the call to `incr`.
- Where should be put `x`, so we can use it after the call to `incr`?
Callee-Saved Example #2

```c
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
```
Callee-Saved Example #3

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

call_incr2:
```assembly
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
```
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
```

Stack Structure

- X saved in %rbx.
- A callee saved register.
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

Stack Structure

- X saved in %rbx.
- A callee saved register.
Callee-Saved Example #6

```c
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

Stack Structure

- Rtn address
- Saved %rbx
- 18213 %rsp+8
- Unused %rsp

- X Is safe in %rbx
- Return result in %rax
Callee-Saved Example #7

```c
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

Stack Structure
```

• Return result in %rax

```

Rtn address
Saved %rbx
18213
Unused
```
Callee-Saved Example #8

```c
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
- Saved %rbx
- 18213
- Unused

**final Stack Structure**
- Rtn address
- Saved %rbx
- 18213
- Unused
Callee-Saved Example #2

```c
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
```
Today

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

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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x &gt;&gt; 1</td>
<td>Rec. argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Recursive call</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return value</td>
<td></td>
</tr>
</tbody>
</table>
/* 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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>x &amp; 1</td>
<td>Callee-saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
<td></td>
</tr>
</tbody>
</table>
Recursive Function Completion

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

### Register Usage

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```
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
```
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
    - Put function arguments at top of stack
    - Result return in `%rax`

- **Pointers are addresses of values**
  - On stack or global