CS 33

Machine Programming (4)
Conditional-Branch Example

```c
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x - y;
    } else {
        result = y - x;
    }
    return result;
}
```

```
absdiff:
    movl %esi, %eax
    cmpl %esi, %edi
    jle .L6
    subl %eax, %edi
    movl %edi, %eax
    jmp .L7

.L6:
    subl %edi, %eax

.L7:
    ret
```

x in %edi
y in %esi
Conditional-Branch Example (Cont.)

```c
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    goto Exit;
Else:
    result = y-x;
Exit:
    return result;
}
```

- C allows “goto” as means of transferring control
  - closer to machine-level programming style
- Generally considered bad coding style

```asm
absdiff:
    movl %esi, %eax
    cmpl %esi, %edi
    jle .L6
    subl %eax, %edi
    movl %edi, %eax
    jmp .L7
.L6:
    subl %edi, %eax
.L7:
    ret
```

Body1

Body2a

Body2b
General Conditional-Expression Translation

C Code

\[
\text{val} = \text{Test} ? \text{Then}_\text{Expr} : \text{Else}_\text{Expr};
\]

\[
\text{val} = x>y ? x-y : y-x;
\]

Goto Version

\[
\text{nt} = \neg \text{Test};
\text{if} (\text{nt}) \text{ goto Else;}
\text{val} = \text{Then}_\text{Expr;}
\text{goto Done;}
\text{Else:}
\quad \text{val} = \text{Else}_\text{Expr;}
\text{Done:}
\quad \ldots
\]

- Test is expression returning integer
  - == 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one
“Do-While” Loop Example

C Code

```c
int pcount_do(unsigned x) {
    int result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Goto Version

```c
int pcount_do(unsigned x) {
    int result = 0;
    loop:
    result += x & 0x1;
    x >>= 1;
    if (x)
        goto loop;
    return result;
}
```

- Count number of 1’s in argument x (“popcount”)
- Use conditional branch either to continue looping or to exit loop
“Do-While” Loop Compilation

Goto Version

```c
int pcount_do(unsigned x) {
    int result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x)
            goto loop;
    return result;
}
```

```
movl $0, %eax       # result = 0
.L2:                 # loop:
    movl %edi, %ecx
    andl $1, %ecx      # t = x & 1
    addl %ecx, %eax    # result += t
    shrl %edi          # x >>= 1
    jne .L2            # if !0, goto loop

Registers:
%edi     x
%eax     result
```
General “Do-While” Translation

C Code

```
do
    Body
while (Test);
```

- **Body:**
  ```
  \{ \\
  \hspace{1em} Statement_1; \\
  \hspace{1em} Statement_2; \\
  \hspace{1em} \ldots \\
  \hspace{1em} Statement_n; \\
  \}
  ```

- **Test** returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

Goto Version

```
loop:
    Body
    if (Test)
    goto loop
```
“While” Loop Example

C Code

```c
int pcount_while(unsigned x) {
    int result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Goto Version

```c
int pcount_do(unsigned x) {
    int result = 0;
    if (!x) goto done;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x)
            goto loop;
    done:
    return result;
}
```

- Is this code equivalent to the do-while version?
  - must jump out of loop if test fails
General “While” Translation

While version

```
while (Test)
  Body
```

Do-While Version

```
if (!Test)
  goto done;
do
  Body
while (Test);
done:
```

Goto Version

```
if (!Test)
goto done;
loop:
  Body
  if (Test)
goto loop;
done:
```
“For” Loop Example

C Code

```c
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

• Is this code equivalent to other versions?
“For” Loop Form

General Form

```
for (Init; Test; Update)
   Body
```

Init
```
i = 0
```

Test
```
i < WSIZE
```

Update
```
i++
```

Body
```
for (i = 0; i < WSIZE; i++) {
   unsigned mask = 1 << i;
   result += (x & mask) != 0;
}
```
“For” Loop → While Loop

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```
"For" Loop → ... → Goto

For Version

```c
for (Init; Test; Update)
    Body
```

While Version

```c
Init;
while (Test) {
    Body
    Update;
}
```

```c
Init;
if (!Test)
goto done;
loop:
    Body
    Update
    if (Test)
goto loop;
done:
```

```c
Init;
if (!Test)
goto done;
do
    Body
    Update
while (Test);
done:
```
"For" Loop Conversion Example

C Code

```c
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
    int i;
    int result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    return result;
}
```

Initial test can be optimized away

Goto Version

```c
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;  
    i = 0;
    if (!(i < WSIZE))
        goto done;

    loop:
    {
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }
    i++;
    if (i < WSIZE)
        goto loop;

done:
    return result;
}
```
Switch-Statement Example

- Multiple case labels
  - here: 5 & 6

- Fall-through cases
  - here: 2

- Missing cases
  - here: 4

```c
long switch_eg
    (long x, long y, long z) {
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
    }
```
Jump-Table Structure

Switch Form

```c
switch(x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ...  
    case val_{n-1}:
        Block {n-1}
}
```

Jump Table

```
jtab:
    Targ0
    Targ1
    Targ2
    ...  
    Targ_{n-1}
```

Jump Targets

```
Targ0:  Code Block 0
Targ1:  Code Block 1
Targ2:  Code Block 2
...  
Targ_{n-1}:  Code Block n-1
```

Approximate Translation

```c
target = JTab[x];
goto *target;
```
Switch-Statement Example (x86-64)

```c
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        ...
    }
    return w;
}
```

Setup:

```assembly
switch_eg:
    ... # Setup
    movq (%rdx, %rcx) # %rcx = z
    cmpq $6, %rdi # Compare x:6
    ja .L8 # If unsigned > goto default
    jmp *.L7(%rdi,8) # Goto *JTab[x]
```

What range of values is covered by the default case?

Note that w not initialized here
Switch-Statement Example

```c
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch(x) {
        ...
    }
    return w;
}
```

Jump table

```
.section .rodata
.align 4
.L7:
.quad .L8 # x = 0
.quad .L3 # x = 1
.quad .L4 # x = 2
.quad .L9 # x = 3
.quad .L8 # x = 4
.quad .L6 # x = 5
.quad .L6 # x = 6
```

Setup:

```
switch_eg:
    ...
    # Setup
    movq %rdx, %rcx      # %rcx = z
    cmpq $6, %rdi        # Compare x:6
    ja .L8               # If unsigned > goto default
    jmp *:.L7(,%rdi,8)   # Goto *JTab[x]
```
Assembly-Setup Explanation

• Table structure
  – each target requires 8 bytes
  – base address at .L7

• Jumping
  
  direct: jmp .L8  
  – jump target is denoted by label .L8

  indirect: jmp *.*.L7(,%rdi,8)  
  – start of jump table: .L7 
  – must scale by factor of 8 (labels have 8 bytes on x86-64) 
  – fetch target from effective address .L7 + rdi*8
    » only for 0 ≤ x ≤ 6

Jump table

```assembly
.section .rodata
.align 4
.L7:
    .quad .L8  # x = 0
    .quad .L3  # x = 1
    .quad .L4  # x = 2
    .quad .L9  # x = 3
    .quad .L8  # x = 4
    .quad .L6  # x = 5
    .quad .L6  # x = 6
```
Jump Table

Jump table

```
.switch(x) {  
    case 1:    // .L3  
        w = y*z;  
        break;  
    case 2:    // .L4  
        w = y/z;  
        /* Fall Through */  
    case 3:    // .L9  
        w += z;  
        break;  
    case 5:  
    case 6:    // .L6  
        w -= z;  
        break;  
    default:    // .L8  
        w = 2;  
}  
```
Code Blocks (Partial)

```c
switch(x) {
  case 1:  // .L3
    w = y*z;
    break;
  . . .
  case 5:  // .L6
  case 6:  // .L6
    w = z;
    break;
  default: // .L8
    w = 2;
}
```

```
.L3:   # x == 1
  movl  %rsi, %rax  # y
  imulq %rdx, %rax  # w = y*z
  ret

.L6:   # x == 5, x == 6
  movl  $1, %eax  # w = 1
  subq  %rdx, %rax  # w -= z
  ret

.L8:   # Default
  movl  $2, %eax  # w = 2
  ret
```
Handling Fall-Through

```c
long w = 1;
...
switch (x) {
    ...
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    ...
}
```

```c
    case 2:
        w = y/z;
        goto merge;
```

```c
    case 3:
        w = 1;
    merge:
        w += z;
```
Code Blocks (Rest)

```c
switch(x) {
    ...
    case 2: // .L4
        w = y/z;
        /* Fall Through */
    case 3: // .L9
        w += z;
        break;
    ...
}

.L4:    # x == 2
    movq %rsi, %rax
    movq %rsi, %rdx
    sarq $63, %rdx
    idivq %rcx    # w = y+z
    jmp .L5
.L9:    # x == 3
    movl $1, %eax # w = 1
.L5:    # merge:
    addq %rcx, %rax # w += z
    ret
```
The diagram illustrates the `idivq` instruction, which performs a division operation. The dividend is stored in `rdx` and `rax`, and the divisor is stored in `rcx`. The result consists of the quotient and remainder, which are stored in the same registers after the division operation. The diagram shows the division process with the relevant register values.
x86-64 Object Code

• Setup
  – label .L8 becomes address 0x4004e5
  – label .L7 becomes address 0x4005c0

Assembly code

```
switch_eg:
  ...
  ja   .L8          # If unsigned > goto default
  jmp  *.L7(,%rdi,8) # Goto *JTab[x]
```

Disassembled object code

```
00000000004004ac <switch_eg>:
  ...
  4004b3: 77 30      ja  4004e5 <switch_eg+0x39>
  4004b5: ff 24 fd c0 05 40 00 jmpq *0x4005c0(,%rdi,8)
```
• Jump table
  – doesn’t show up in disassembled code
  – can inspect using gdb

  gdb switch

  (gdb) x/7xg 0x4005c0
    » examine 7 hexadecimal format “giant” words (8-bytes each)
    » use command “help x” to get format documentation

0x4005c0: 0x000000000004004e5 0x00000000004004bc
0x4005d0: 0x000000000004004c4 0x00000000004004d3
0x4005e0: 0x000000000004004e5 0x00000000004004dc
0x4005f0: 0x000000000004004dc
x86-64 Object Code (cont.)

- Deciphering jump table

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4005c0</td>
<td>0x00000000004004e5</td>
<td>0x00000000004004bc</td>
</tr>
<tr>
<td>0x4005d0</td>
<td>0x00000000004004c4</td>
<td>0x00000000004004d3</td>
</tr>
<tr>
<td>0x4005e0</td>
<td>0x00000000004004e5</td>
<td>0x00000000004004dc</td>
</tr>
<tr>
<td>0x4005f0</td>
<td>0x00000000004004dc</td>
<td></td>
</tr>
</tbody>
</table>
Disassembled Targets

```
(gdb) disassemble 0x4004bc,0x4004eb
Dump of assembler code from 0x4004bc to 0x4004eb
  0x00000000004004bc <switch_eg+16>:  mov  %rsi,%rax
  0x00000000004004bf <switch_eg+19>:  imul %rdx,%rax
  0x00000000004004c3 <switch_eg+23>:  retq
  0x00000000004004c4 <switch_eg+24>:  mov  %rsi,%rax
  0x00000000004004c7 <switch_eg+27>:  mov  %rsi,%rdx
  0x00000000004004ca <switch_eg+30>:  sar  $0x3f,%rdx
  0x00000000004004ce <switch_eg+34>:  idiv %rcx
  0x00000000004004d1 <switch_eg+37>:  jmp  0x4004d8 <switch_eg+44>
  0x00000000004004d3 <switch_eg+39>:  mov  $0x1,%eax
  0x00000000004004d8 <switch_eg+44>:  add  %rcx,%rax
  0x00000000004004db <switch_eg+47>:  retq
  0x00000000004004dc <switch_eg+48>:  mov  $0x1,%eax
  0x00000000004004e1 <switch_eg+53>:  sub  %rdx,%rax
  0x00000000004004e4 <switch_eg+56>:  retq
  0x00000000004004e5 <switch_eg+57>:  mov  $0x2,%eax
  0x00000000004004ea <switch_eg+62>:  retq
```
Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Value</th>
<th>(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4004e5</td>
<td>0</td>
</tr>
<tr>
<td>0x4004bc</td>
<td>1</td>
</tr>
<tr>
<td>0x4004c4</td>
<td>2</td>
</tr>
<tr>
<td>0x4004d3</td>
<td>3</td>
</tr>
<tr>
<td>0x4004e5</td>
<td>4</td>
</tr>
<tr>
<td>0x4004dc</td>
<td>5</td>
</tr>
<tr>
<td>0x4004dc</td>
<td>6</td>
</tr>
</tbody>
</table>

- \(0x0000000000004004bc\): mov %rsi,%rax
- \(0x0000000000004004bf\): imul %rdx,%rax
- \(0x0000000000004004c3\): retq
- \(0x0000000000004004c4\): mov %rsi,%rax
- \(0x0000000000004004c7\): mov %rsi,%rdx
- \(0x0000000000004004ca\): sar $0x3f,%rdx
- \(0x0000000000004004ce\): idiv %rcx
- \(0x0000000000004004d1\): jmp 0x4004d8
- \(0x0000000000004004d3\): mov $0x1,%eax
- \(0x0000000000004004d8\): add %rcx,%rax
- \(0x0000000000004004db\): retq
- \(0x0000000000004004dc\): mov $0x1,%eax
- \(0x0000000000004004e1\): sub %rdx,%rax
- \(0x0000000000004004e4\): retq
- \(0x0000000000004004e5\): mov $0x2,%eax
- \(0x0000000000004004ea\): retq
Quiz 1

What C code would you compile to get the following assembler code?

```
movq  $0, %rax
.L2:
    movq  %rax, a(,%rax,8)
    addq  $1, %rax
    cmpq  $10, %rax
    jne   .L2
    ret
```

```
long a[10];
void func() {
    long i;
    for (i=0; i<10; i++)
        a[i] = 1;
}
```

```
long a[10];
void func() {
    long i=0;
    while (i<10)
        a[i] = i++;
}
```

```
long a[10];
void func() {
    long i=0;
    switch (i) {
    case 0:
        a[i] = 0;
        break;
    default:
        a[i] = 10
    }
}
```
IA32 Stack

- Region of memory managed “last-in, first-out”
- Grows toward lower addresses
- Register $\%esp$ contains lowest stack address
  - address of “top” element

Stack pointer: $\%esp$

Stack “bottom”

Stack “top”

Increasing addresses

Stack grows down
IA32 Stack: Push

- `pushl src`
  - fetch operand at `src`
    » immediate, register, or memory location
  - decrement `%esp` by 4
  - store operand at address given by `%esp`
IA32 Stack: Pop

- `popl dest`
  - fetch operand from address given by `%esp`
  - put operand in dest
    » register or memory location
  - increment `%esp` by 4
Digression (Again): Where Stuff Is (Roughly)

Virtual Memory

0: Code (aka text)

Global and Static Local Data

Stack

$2^n - 1$:
Function Control Flow

• Use stack to support function call and return

  • **Function call:** `call sub`
    - push return address on stack
    - jump to `sub`

  • **Return address:**
    - address of the next instruction after call
    - example from disassembly
      
```assembly
804854e:   e8 3d 06 00 00    call  8048b90 <sub>
8048553:   50     pushl %eax
```
    - return address = 0x8048553

• **Function return:** `ret`
  - pop address from stack
  - jump to address
Function Call

804854e: e8 3d 06 00 00 call 8048b90 <sub>
8048553: 50 pushl %eax

call 8048b90

%esp 0x108 %esp 0x104
%eip 0x804854e %eip 0x8048b90

%eip: program counter
Function Return

```
8048591: c3 ret
```

```
<table>
<thead>
<tr>
<th>%esp</th>
<th>0x104</th>
<th>%esp</th>
<th>0x108</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>%eip</th>
<th>0x8048591</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>%eip</th>
<th>0x8048553</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>ret</th>
<th></th>
</tr>
</thead>
</table>
```

```
<table>
<thead>
<tr>
<th>%eip</th>
<th>0x8048553</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

%eip: program counter
The IA32 Stack Frame

- arg n
- ...
- arg 1
- return address
- saved frame pointer
- saved registers
- local variables

%ebp
%esp
Passing Arguments

```c
int x;
int res;
int main() {
    ...
    res = subr(3, x);
    ...
}
```

```
main:
    ...
    pushl x
    pushl $3
    call subr
    movl %eax, res
    ...
```
Retrieving Arguments

```c
int subr(int a, int b) {
    return a + b;
}
```

subr:
```
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
addl 8(%ebp), %eax
popl %ebp
ret
```
Space for Local Variables

```c
int subr(int a, int b) {
    int array[20];
    ...
}
```

subr:

```
pushl %ebp
movl %esp, %ebp
subl $80, %esp
...
addl $80, %esp
popl %ebp
ret
```
Register-Saving Conventions

• When function \texttt{yoo} calls \texttt{who}:  
  – \texttt{yoo} is the caller  
  – \texttt{who} is the callee

• Can registers be used for temporary storage?

\texttt{yoo:}
\begin{align*}
  &\ldots \\
  &\text{movl} \ $33, \ %edx \\
  &\text{call} \ \texttt{who} \\
  &\text{addl} \ %edx, \ %eax \\
  &\ldots \\
  &\text{ret}
\end{align*}

\texttt{who:}
\begin{align*}
  &\ldots \\
  &\text{movl} \ 8(%ebp), \ %edx \\
  &\text{addl} \ $32, \ %edx \\
  &\ldots \\
  &\text{ret}
\end{align*}

– contents of register \%edx overwritten by \texttt{who}  
– this could be trouble: something should be done!

» need some coordination
Register-Saving Conventions

• When function **yoo** calls **who**:
  – **yoo** is the caller
  – **who** is the callee

• Can registers be used for temporary storage?

• Conventions
  – “**caller save**”
    » caller saves registers containing temporary values on stack before the call
    » restores them after call
  – “**callee save**”
    » callee saves registers on stack before using
    » restores them before returning
IA32/Linux+Windows Register Usage

- `%eax`, `%edx`, `%ecx`
  - caller saves prior to call if values are used later

- `%eax`
  - also used to return integer value

- `%ebx`, `%esi`, `%edi`
  - callee saves if wants to use them

- `%esp`, `%ebp`
  - special form of callee-save
  - restored to original values upon exit from function
Register-Saving Example

**yoo:**

```
... 
  movl $33, %edx
  pushl %edx
  call who
  popl %edx
  addl %edx, %eax
... 
ret
```

**who:**

```
... 
  pushl %ebx
  ... 
  movl 4(%ebp), %ebx
  addl %53, %ebx
  movl 8(%ebp), %edx
  addl $32, %edx
  ... 
  popl %ebx
  ... 
ret
```
/* Recursive popcount */

```c
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

**Recursive Function**

**Registers**

- `%eax`, `%edx` used without first saving
- `%ebx` used, but saved at beginning & restored at end

**pcount_r:**

```assembly
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
movl 8(%ebp), %ebx
movl $0, %eax
testl %ebx, %ebx
je .L3
movl %ebx, %eax
shrl $1, %eax
movl %eax, (%esp)
call pcount_r
movl %ebx, %edx
andl $1, %edx
leal (%edx,%eax), %eax
.L3:
addl $4, %esp
popl %ebx
popl %ebp
ret
```
/* Recursive popcount */
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else return (x & 1) + pcount_r(x >> 1);
}

• Actions
  – save old value of %ebx on stack
  – allocate space for argument to recursive call
  – store x in %ebx

  pcount_r:
  pushl %ebp
  movl %esp, %ebp
  pushl %ebx
  subl $4, %esp
  movl 8(%ebp), %ebx
  ...

  • Actions
  – save old value of %ebx on stack
  – allocate space for argument to recursive call
  – store x in %ebx

  %ebp
  x
  Rtn adr
  Old %ebp
  Old %ebx
  %esp
Recursive Call #2

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

• Actions
  – if x == 0, return
    » with %eax set to 0

```assembly
  movl $0, %eax
  testl %ebx, %ebx
  je .L3
      ...
  .L3:
      ...
  ret
```

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

• Actions
  – store x >> 1 on stack
  – make recursive call

• Effect
  – %eax set to function result
  – %ebx still has value of x

movl %ebx, %eax
shrl $1, %eax
movl %eax, (%esp)
call pcount_r

• • •
/* Recursive popcount */

```c
int pcount_r(unsigned x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

### Recursive Call #4

- **Assume**
  - `%eax` holds value from recursive call
  - `%ebx` holds `x`
- **Actions**
  - Compute `(x & 1) +` computed value
- **Effect**
  - `%eax` set to function result

```assembly
movl %ebx, %edx
andl $1, %edx
leal (%edx,%eax), %eax
```

---

*CS33 Intro to Computer Systems*  
*XII–50*
Recursive Call #5

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

• Actions
  – restore values of %ebx and %ebp
  – restore %esp

L3:
    addl $4, %esp
    popl %ebx
    popl %ebp
    ret

%ebp

%esp

%ebx

Old %ebx

Old %ebp

Rtn adr

CS33 Intro to Computer Systems

XII–51
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
  – 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
Why Bother with a Frame Pointer?

• It (%rbp) points to the beginning of the stack frame
  – making it easy for people to figure out where things are in the frame
  – but people don’t execute the code ...

• The stack pointer always points somewhere within the stack frame
  – it moves about, but the compiler knows where it is pointing
    » a local variable might be at 8(%rsp) for one instruction, but at 16(%rsp) for a subsequent one
    » tough for people, but easy for the compiler

• Thus the frame pointer is superfluous
  – it can be used as a general-purpose register
### x86-64 General-Purpose Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>Return value</td>
<td><code>%r8</code></td>
<td>Argument #5</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td>Callee saved</td>
<td><code>%r9</code></td>
<td>Argument #6</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument #4</td>
<td><code>%r10</code></td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Argument #3</td>
<td><code>%r11</code></td>
<td>Caller Saved</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument #2</td>
<td><code>%r12</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>Argument #1</td>
<td><code>%r13</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Stack pointer</td>
<td><code>%r14</code></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td>Callee saved</td>
<td><code>%r15</code></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>