Most of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
Not a Quiz ...

**What value ends up in %ecx?**

```assembly
movl $1000, %eax
movl $1, %ebx
movl 2(%eax, %ebx, 4), %ecx
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1009:</td>
<td>0x09</td>
</tr>
<tr>
<td>1008:</td>
<td>0x08</td>
</tr>
<tr>
<td>1007:</td>
<td>0x07</td>
</tr>
<tr>
<td>1006:</td>
<td>0x06</td>
</tr>
<tr>
<td>1005:</td>
<td>0x05</td>
</tr>
<tr>
<td>1004:</td>
<td>0x04</td>
</tr>
<tr>
<td>1003:</td>
<td>0x03</td>
</tr>
<tr>
<td>1002:</td>
<td>0x02</td>
</tr>
<tr>
<td>1001:</td>
<td>0x01</td>
</tr>
</tbody>
</table>

- a) 0x02030405
- b) 0x05040302
- c) 0x06070809
- d) 0x09080706

**Hint:**

%eax → 1000: 0x00
Note that %ebp/%rbp may be used as a base register as on IA32, but they don’t have to be used that way. This will become clearer when we explore how the runtime stack is accessed. The convention on Linux is for the first 6 arguments of a function to be in registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9. The return value of a function is put in %rax.

Note also that each register, in addition to having a 32-bit version, also has an 8-bit (one-byte) version. For the numbered registers, it’s, for example, %r10b. For the other registers it’s the same as for IA32.
On x86-64, for instructions with 32-bit (long) operands that produce 32-bit results going into a register, the register must be a 32-bit register; the higher-order 32 bits are filled with zeroes.
Bytes

- Each register has a byte version
  - e.g., %r10: %r10b
- Needed for byte instructions
  - movb (%rax, %rsi), %r10b
  - sets only the low byte in %r10
    » other seven bytes are unchanged
- Alternatives
  - movzbq (%rax, %rsi), %r10
    » copies byte to low byte of %r10
    » zeroes go to higher bytes
  - movsbq (%rax, %rsi), %r10
    » copies byte to low byte of %r10
    » sign is extended to all higher bits

Note that using single-byte versions of registers has a different behavior from using 4-byte versions of registers. Putting data into the latter using mov causes the upper bytes to be zeroed. But with the byte versions, putting data into them does not affect the upper bytes.
Supplied by CMU.

Note that for the IA32 architecture, arguments are passed on the stack.
No more than six arguments can be passed in registers. If there are more than six arguments (which is unusual), then remaining arguments are passed on the stack, and referenced via %rsp.
64-bit code for long int swap

```c
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- **Set Up**
  - `movq (%rdi), %rdx`
  - `movq (%rsi), %rax`
  - `movq %rax, (%rdi)`
  - `movq %rdx, (%rsi)`

- **Body**
  - `ret`

- **Finish**

- **64-bit data**
  - data held in registers `%rax` and `%rdx`
  - `movq` operation
    » “q” stands for quad-word

Supplied by CMU.
Note that normally one does not ask gcc to produce assembler code, but instead it compiles C code directly into machine code (producing an object file). Note also that the gcc command actually invokes a script; the compiler (also known as gcc) compiles code into either assembler code or machine code; if necessary, the assembler (as) assembles assembler code into object code. The linker (ld) links together multiple object files (containing object code) into an executable program.
Example

```c
int sum(int a, int b) {
    return (a+b);
}
```
Object Code

**Code for sum**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040</td>
<td>&lt;sum&gt;:</td>
</tr>
<tr>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td>0x89</td>
<td></td>
</tr>
<tr>
<td>0xe5</td>
<td></td>
</tr>
<tr>
<td>0x8b</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x0c</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>0x45</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x5d</td>
<td></td>
</tr>
<tr>
<td>0xc3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total of 11 bytes</td>
</tr>
<tr>
<td></td>
<td>Each instruction: 1, 2, or 3 bytes</td>
</tr>
<tr>
<td></td>
<td>Starts at address 0x401040</td>
</tr>
</tbody>
</table>

- **Assembler**
  - translates `.s` into `.o`
  - binary encoding of each instruction
  - nearly-complete image of executable code
  - missing linkages between code in different files

- **Linker**
  - resolves references between files
  - combines with static run-time libraries
    - e.g., code for `printf`
  - some libraries are *dynamically linked*
    - linking occurs when program begins execution

---

Supplied by CMU.
Instruction Format

Disassembling Object Code

Disassembled

```
080483c4 <sum>:
  80483c4: 55  push  ebp
  80483c5: e5  mov   esp,ebp
  80483c7: 8b 45 0c  mov   0xc(esp),eax
  80483ca: 03 45 08  add   0x8(esp),eax
  80483cd: 5d  pop   ebp
  80483ce: c3  ret
```

- **Disassembler**
  - `objdump -d <file>`
    - useful tool for examining object code
    - analyzes bit pattern of series of instructions
    - produces approximate rendition of assembly code
    - can be run on either executable or object (.o) file

Supplied by CMU.
### Alternate Disassembly

#### Object

- `0x401040:
  - 0x55
  - 0x89
  - 0xe5
  - 0x8b
  - 0x45
  - 0x0c
  - 0x03
  - 0x45
  - 0x08
  - 0x6d
  - 0xc3`

#### Disassembled

Dump of assembler code for function sum:

- `0x080483c4 <sum+0>:` push %ebp  
- `0x080483c5 <sum+1>` mov %esp,%ebp  
- `0x080483c7 <sum+3>` mov 0xc(%ebp),%eax  
- `0x080483ca <sum+6>` add 0x8(%ebp),%eax  
- `0x080483cd <sum+9>` pop %ebp  
- `0x080483ce <sum+10>` ret

- **Within gdb debugger**
  - `gdb <file>`
  - `disassemble sum`
    - `disassemble procedure`
  - `x/11xb sum`
    - `examine the 11 bytes starting at sum`

---

Supplied by CMU.
How Many Instructions are There?

- We cover ~30
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a

- Total: 198
- Doesn’t count:
  - floating-point instructions
    - ~100
  - SIMD instructions
    - lots
  - AMD-added instructions
  - undocumented instructions

The source for this is http://en.wikipedia.org/wiki/X86_instruction_listings, viewed on 6/20/2017, which comes with the caveat that it may be out of date.
Some Arithmetic Operations

- Two-operand instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest = Dest + Src</td>
<td></td>
</tr>
<tr>
<td>subl</td>
<td>Dest = Dest − Src</td>
<td></td>
</tr>
<tr>
<td>imull</td>
<td>Dest = Dest * Src</td>
<td></td>
</tr>
<tr>
<td>sall</td>
<td>Dest = Dest &lt;&lt; Src</td>
<td>Also called shl</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>shrl</td>
<td>Dest = Dest &gt;&gt; Src</td>
<td>Logical</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest ^ Src</td>
<td></td>
</tr>
<tr>
<td>andl</td>
<td>Dest = Dest &amp; Src</td>
<td></td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
<td></td>
</tr>
</tbody>
</table>

- watch out for argument order!
- no distinction between signed and unsigned int (why?)

Supplied by CMU.

Note that for shift instructions, the Src operand (which is the size of the shift) must either be a immediate operand or be a designator for a one-byte register (e.g., %cl – see the slide on general-purpose registers for IA32).
Some Arithmetic Operations

- **One-operand Instructions**
  
  incl Dest = Dest + 1  
  decl Dest = Dest − 1  
  negl Dest = − Dest  
  notl Dest = “Dest

- See book for more instructions

Supplied by CMU.
Arithmetic Expression Example

```c
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
arith:
leal (%rdi,%rsi), %eax
addl %edx, %eax
leal (%rsi,%rsi,2), %edx
sall $4, %edx
leal 4(%rdi,%rdx), %ecx
imull %ecx, %eax
ret
```
Understanding arith

```c
int arith(int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

leal (%rdi,%rsi), %eax
addl %edx, %eax
leal (%rsi,%rsi,2), %edx
sall $4, %edx
leal 4(%rdi,%rdx), %ecx
imull %ecx, %eax
ret

Supplied by CMU, but converted to x86-64.
By convention, the first three arguments to a procedure are placed in registers rdi, rsi, and rdx, respectively. Note that, also by convention, procedures put their return values in register eax/rax.
Observations about arith

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instructions in different order from C code
- Some expressions might require multiple instructions
- Some instructions might cover multiple expressions

```
lea (%rdi,%rsi), %eax  # eax = x+y  (t1)
addl %edx, %eax        # eax = t1+z  (t2)
lea (%rsi,%rsi,2), %edx # edx = 3*y  (t4)
sall $4, %edx          # edx = t4*16  (t4)
lea 4(%rdi,%rdx), %ecx # ecx = x+4+t4  (t5)
imull %ecx, %eax       # eax *= t5  (rval)
ret
```

Supplied by CMU, but converted to x86-64.
Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
xorl %esi, %edi       # edi = x^y  (t1)
sar1 $17, %edi        # edi = t1>>17 (t2)
movl %edi, %eax       # eax = edi
andl $8185, %eax      # eax = t2 & mask (rval)
```

Supplied by CMU, but converted to x86-64.
Quiz 1

- What is the final value in %ecx?

```
xorl %ecx, %ecx
incl %ecx
sall %cl, %ecx  # %cl is the low byte of %ecx
addl %ecx, %ecx
```

a) 2
b) 4
c) 8
d) indeterminate
### Processor State (x86-64, Partial)

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>eax</td>
</tr>
<tr>
<td>rbx</td>
<td>ebx</td>
</tr>
<tr>
<td>rcx</td>
<td>ecx</td>
</tr>
<tr>
<td>rdx</td>
<td>edx</td>
</tr>
<tr>
<td>rsi</td>
<td>esi</td>
</tr>
<tr>
<td>rdi</td>
<td>edi</td>
</tr>
<tr>
<td>rsp</td>
<td>esp</td>
</tr>
<tr>
<td>rip</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td>r8d</td>
</tr>
<tr>
<td>r9</td>
<td>r9d</td>
</tr>
<tr>
<td>r10</td>
<td>r10d</td>
</tr>
<tr>
<td>r11</td>
<td>r11d</td>
</tr>
<tr>
<td>r12</td>
<td>r12d</td>
</tr>
<tr>
<td>r13</td>
<td>r13d</td>
</tr>
<tr>
<td>r14</td>
<td>r14d</td>
</tr>
<tr>
<td>r15</td>
<td>r15d</td>
</tr>
</tbody>
</table>

- **Condition Codes**
  - CF
  - ZF
  - SF
  - OF
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: `addl/addq` Src, Dest ← t = a+b
  - **CF set** if carry out from most significant bit or borrow (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 lea instruction**
Condition Codes (Explicit Setting: Compare)

- Explicit setting by compare instruction
  
  ```
  cmp1/cmpq src2, src1
  compares src1:src2
  cmp1 b, a like computing a−b without setting destination
  ```

  **CF set** if carry out from most significant bit or borrow (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
  \( (a>0 \&\& b<0 \&\& (a−b)<0) \mid \mid (a<0 \&\& b>0 \&\& (a−b)>0) \)
Condition Codes (Explicit Setting: Test)

- Explicit setting by test instruction
  \[ \text{testl/testq src2, src1} \]
  \[ \text{testl b, a like computing } a\&b \text{ without setting destination} \]

  - sets condition codes based on value of Src1 & Src2
  - useful to have one of the operands be a mask

\[ \text{ZF set when } a\&b = 0 \]
\[ \text{SF set when } a\&b < 0 \]
Reading Condition Codes

- **SetX instructions**
  - set single byte based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Recall that the first argument to a function is passed in %rdi (%edi) and the second in %rsi (%esi).
Jumping

- **jX instructions**
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Conditional-Branch Example

```
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

Supplied by CMU, but converted to x86-64.
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

Supplied by CMU, but converted to x86-64.
General Conditional-Expression Translation

C Code

```c
val = Test ? Then_Expr : Else_Expr;
val = x>y ? x-y : y-x;
```

Goto Version

```c
nt = !Test;
if (nt) goto Else;
val = Then_Expr;
goto Done;
Else:
val = Else_Expr;
Done:
...
```

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

Supplied by CMU.
"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
Supplied by CMU.

Note that the condition codes are set as part of the execution of the shr1 instruction.
General “Do-While” Translation

C Code

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

- **Body:**
  ```
  { 
    Statement_1;
    Statement_2;
    ...
    Statement_n;
  }
  ```

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

Goto Version

```c
loop:
    Body
    if (Test)
        goto loop
```

Supplied by CMU.
“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

Supplied by CMU.
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?

Supplied by CMU.
"For" Loop Form

General Form

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

- **Init**
  ```
i = 0
```
- **Test**
  ```
i < WSIZE
```
- **Update**
  ```
i++
```
- **Body**
  ```
  { 
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  ```

Supplied by CMU.
“For” Loop → While Loop

For Version

for (Init; Test; Update)
  Body

While Version

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

For Version

for (Init; Test; Update)
  Body

While Version

Init;
while (Test) {
  Body
  Update;
}

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

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

Supplied by CMU.
“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;
}
```

Goto Version

```c
int pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    i = 0;
    if (i < WSIZE) \textcolor{red}{Init}
        goto done;
    \textcolor{red}{Test}
    \textcolor{green}{loop:}
    \textcolor{red}{Body}
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
    i++;
    if (i < WSIZE) \textcolor{green}{Test}
        goto loop;
    done:
    return result;
}
```

Initial test can be optimized away

Supplied by CMU.
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;
}
```
Supplied by CMU.

The translation is “approximate” because C doesn’t have the notion of the target of a goto being a variable. But, if it did, then the translation is what we’d want!
Supplied by CMU, but converted to x86-64.

Note that the `ja` in the slide causes a jump to occur if the previous comparison is interpreted as being performed on unsigned values, and the result is that `x` is greater than (above) `6`. Given that `x` is declared to be a `signed` value, for what range of values of `x` will `ja` cause a jump to take place?

Note that the assembler code shown in the examples was produced by compiling the C code using gcc with the “-O1” flag.
Switch-Statement Example

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

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]
```
The `jmp` instruction is doing a couple of things that require explanation: The asterisk means it’s an *indirect jump* (such indirection is allowed only in jumps). The address specified after the asterisk is the address of an entry in the *jump table*. The asterisk means, rather than jumping directly to that entry, jump to the address that’s in that table entry. “.L7” is a label that’s being used as a displacement in the address computation. The value of .L7 is the address of the area of memory it labels. In this case, it’s the address of the jump table. Thus, an unconditional jump is to take place to the address contained in the 8-byte entry of the jump table indexed by the contents of `%rdi`. Thus, if `%rdi` is, say, 2, then a jump will take place to address in the location starting 16 bytes beyond the beginning of the table. This will be a jump to .L4. .L4 itself is a label of code specified elsewhere, the reference to the label is replaced by the assembler with the address of the code labelled with .L4.

The jump table is separate from the code (it’s not executable). This is specified by the “.section” directive, which also specifies that it should be placed in memory that’s made read-only (“.rodata” indicates this). The “.align 4” says that the address of the start of the table should be divisible by four (why this is important is something we’ll get to in a week or two).
Supplied by CMU, but converted to x86-64.
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;  
    }
```

```assembly
.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
```

Supplied by CMU, but converted to x86-64.
Handling Fall-Through

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

Supplied by CMU, but converted to x86-64.
The code following the .L4 label requires some explanation. The `idivq` instruction is special in that it takes a 128-bit dividend that is implicitly assumed to reside in registers `rdx` and `rax`. Its single operand specifies the divisor. The quotient is always placed in the `rax` register, and the remainder in the `rdx` register. In our example, `y`, which we want to be the dividend, is copied into both the `rax` and `rdx` registers. The `sarq` (shift arithmetic right quadword) instruction propagates the sign bit of `rdx` across the entire register, replacing its original contents. Thus, if one considers `rdx` to contain the most-significant bits of the dividend and `rax` to contain the least-significant bits, the pair of registers now contains the 128-bit version of `y`. The `idivq` instruction computes the quotient from dividing this 128-bit value by the 64-bit value contained in register `rcx` (containing `z`). The quotient is stored register `rax` (implicitly) and the remainder is stored in register `rdx` (and is ignored in our example). This illustrated in the next slide.
idivq

127 64 63 0

rax

rdx

dividend

idivq

rcx

divisor

63

0

r dx

0 0

r ax

remainder

quotient
Disassembly was accomplished using “objdump –d”. Note that the text enclosed in angle brackets (“<“, “>”) is essentially a comment, relating the address (4004e5) to a symbolic location (0x39 bytes after the beginning of switch_eg).
x86-64 Object Code (cont.)

- 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
    ```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4005c0</td>
<td>0x00000000004004e5</td>
<td>0x00000000004004bc</td>
<td></td>
</tr>
<tr>
<td>0x4005d0</td>
<td>0x00000000004004c4</td>
<td>0x00000000004004d3</td>
<td></td>
</tr>
<tr>
<td>0x4005e0</td>
<td>0x00000000004004e5</td>
<td>0x00000000004004dc</td>
<td></td>
</tr>
<tr>
<td>0x4005f0</td>
<td>0x00000000004004dc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Supplied by CMU, but converted to x86-64. We assume that the switch_eg function was included in a program whose name is `switch`. Hence, gdb is invoked from the shell with the argument “switch”.
### 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>0x4004e5</td>
<td>0</td>
</tr>
<tr>
<td>0x4005c8</td>
<td>0x4004bc</td>
<td>1</td>
</tr>
<tr>
<td>0x4005d0</td>
<td>0x4004c4</td>
<td>2</td>
</tr>
<tr>
<td>0x4005d8</td>
<td>0x4004d3</td>
<td>3</td>
</tr>
<tr>
<td>0x4005e0</td>
<td>0x4004e5</td>
<td>4</td>
</tr>
<tr>
<td>0x4005e8</td>
<td>0x4004dc</td>
<td>5</td>
</tr>
<tr>
<td>0x4005f0</td>
<td>0x4004dc</td>
<td>6</td>
</tr>
</tbody>
</table>

Supplied by CMU, but converted to x86-64.
Disassembled Targets

```
(gdb) disassemble 0x4004bc,0x4004eb
Dump of assembler code from 0x4004bc to 0x4004eb
0x000000000004004bc <switch_eg+16>: mov  %rsi,%rax
0x000000000004004bf <switch_eg+19>: imul  %rdx,%rax
0x000000000004004c3 <switch_eg+23>: retq
0x000000000004004c4 <switch_eg+24>: mov  %rsi,%rax
0x000000000004004c7 <switch_eg+27>: mov  %rsi,%rdx
0x000000000004004ca <switch_eg+30>: sar  $0x3f,%rdx
0x000000000004004ce <switch_eg+34>: idiv  %rcx
0x000000000004004d1 <switch_eg+37>: jmp  0x4004d8 <switch_eg+44>
0x000000000004004d3 <switch_eg+39>: mov  $0x1,%eax
0x000000000004004d8 <switch_eg+44>: add  %rcx,%rax
0x000000000004004db <switch_eg+47>: retq
0x000000000004004dc <switch_eg+48>: mov  $0x1,%eax
0x000000000004004e1 <switch_eg+53>: sub  %rdx,%rax
0x000000000004004e4 <switch_eg+56>: retq
0x000000000004004e5 <switch_eg+57>: mov  $0x2,%eax
0x000000000004004ea <switch_eg+62>: retq
```
### Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Value</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000040000</td>
<td>0</td>
</tr>
<tr>
<td>0x0000000000400015</td>
<td>1</td>
</tr>
<tr>
<td>0x0000000000400024</td>
<td>2</td>
</tr>
<tr>
<td>0x0000000000400035</td>
<td>3</td>
</tr>
<tr>
<td>0x0000000000400046</td>
<td>4</td>
</tr>
<tr>
<td>0x0000000000400057</td>
<td>5</td>
</tr>
<tr>
<td>0x0000000000400068</td>
<td>6</td>
</tr>
</tbody>
</table>

```
0x0000000000040000:  mov  %rsi,%rax
0x0000000000040004:  mov  %rsi,%rax
0x0000000000040015:  imul %rdx,%rax
0x0000000000040024:  retq
0x0000000000040035:  mov  %rsi,%rax
0x0000000000040046:  mov  %rsi,%rdx
0x0000000000040057:  sar  $0x3f,%rdx
0x0000000000040068:  div  %rcx
0x0000000000040078:  jmp  0x4004d8
0x0000000000040089:  mov  %eax,%rdx
0x000000000004009a:  mov  %eax,%rax
0x00000000000400ab:  add  %rcx,%rax
0x00000000000400bc:  retq
0x00000000000400cd:  mov  $0x1,%eax
0x00000000000400de:  sub  %rdx,%eax
0x00000000000400ef:  retq
0x00000000000400ff:  mov  $0x2,%eax
0x0000000000040100:  retq
```
Quiz 3

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

```
.L2:
    movq  $0, %rax
    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;
    switch (i) {
    case 0:
        a[i] = 0;
        break;
    default:
        a[i] = 10
    }
}
```

---

a  b  c