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.
**Processor State (x86-64, Partial)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Register</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>$rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>$rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>$rip</td>
<td>CF</td>
</tr>
<tr>
<td></td>
<td>ZF</td>
</tr>
<tr>
<td></td>
<td>SF</td>
</tr>
<tr>
<td></td>
<td>OF</td>
</tr>
</tbody>
</table>

**Condition codes**
Condition Codes (Implicit Setting)

- **Single-bit registers**
  - CF  carry flag (for unsigned)  SF  sign flag (for signed)
  - ZF  zero flag  OF  overflow flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**
  - Example: `addl/addq Src,Dest ← t = a+b`
  - **CF set if carry out from most significant bit (unsigned overflow)**
  - **ZF set if t == 0**
  - **SF set if t < 0 (as signed)**
  - **OF set if two’s-complement (signed) overflow**
    - `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- **Not set by `lea` instruction**
Condition Codes (Explicit Setting: Compare)

- Explicit setting by compare instruction
  
  `cmp r1/cmpq src2, src1`
  
  `cmp b, a` like computing `a-b` without setting destination

  **CF set** if carry out from most significant bit (used for unsigned comparisons)
  
  **ZF set** if `a == b`
  
  **SF set** if `(a-b) < 0` (as signed)

  **OF set** if two’s-complement (signed) overflow
  
  `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)"
Condition Codes (Explicit Setting: Test)

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

  - sets condition codes based on value of \( \text{Src1} \ & \ \text{Src2} \)
  - 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>

Supplied by CMU.
Reading Condition Codes (Cont.)

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

- **Uses one of 8 addressable byte registers**
  - does not alter remaining 7 bytes
  - typically use movzbl to finish job

```c
int gt (int x, int y)
{
    return x > y;
}
```

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
</table>

**Body**

```c
cmp %esi, %edi  # compare x : y
setg %al       # %al = x > y
movzbl %al, %eax # zero rest of %eax/%rax
```
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>¬ZF &amp; CF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
## Quiz 1

What would be an appropriate description if the condition is ~CF?
- a) above or equal (unsigned)
- b) not less (signed)
- c) incomparable

<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</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 ^ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Supplied by CMU, but converted to x86-64.
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.
Supplied by CMU.

Note that, as shown in the goto version, both \( \text{then_expr} \) and \( \text{else_expr} \) are computed before the test is performed.
Conditional Move Example: x86-64

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

```assembly
absdiff:
    movl  %edi, %eax
    subl  %esi, %eax  # result = x-y
    movl  %esi, %edx
    subl  %edi, %edx  # tval = y-x
    cmpl  %esi, %edi  # compare x:y
    cmovle %edx, %eax  # if <=, result = tval
    ret
```

Supplied by CMU.
Bad Cases for Conditional Move

Expensive Computations

```plaintext
val = Test(x) ? Hard1(x) : Hard2(x);
```

- both values get computed
- only makes sense when computations are very simple

Risky Computations

```plaintext
val = p ? *p : 0;
```

- both values get computed
- may have undesirable effects

Computations with side effects

```plaintext
val = x > 0 ? x*=7 : x+=3;
```

- both values get computed
- must be side-effect free

Supplied by CMU.
“Do-While” Loop Example

**C Code**

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

**Goto Version**

```
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.
Supplied by CMU.

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

<table>
<thead>
<tr>
<th>C Code</th>
<th>Goto Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>do</code></td>
<td><code>loop:</code></td>
</tr>
<tr>
<td>`    Body</td>
<td><code>    Body</code></td>
</tr>
<tr>
<td><code>    while (Test);</code></td>
<td><code>if (Test)</code></td>
</tr>
<tr>
<td><code>)</code></td>
<td><code>  goto loop</code></td>
</tr>
</tbody>
</table>

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

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

---

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 pcound_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

for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}

Init
i = 0

Test
i < WSIZE

Update
i++

Body
{ unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
"For" Loop $\rightarrow$ 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:
```
“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) ! Test
        goto done;
    loop:
        unsigned mask = 1 << i;
        result += (x & mask) != 0;
    }  
    Update
    if (i < WSIZE) Test
        goto loop;
        done:
    return result;
}
```

Supplied by CMU.
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 = z;
   }
   return w;
   }

Switch-Statement Example

- Multiple case labels
  - here: 5 & 6
- Fall-through cases
  - here: 2
- Missing cases
  - here: 4

Supplied by CMU.
Jump-Table Structure

Switch Form

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

Jump Table

- `jtаб`:
  - `Targ0`
  - `Targ1`
  - `Targ2`
  - `Targn-1`

Jump Targets

- `Targ0`: Code Block 0
- `Targ1`: Code Block 1
- `Targ2`: Code Block 2
- `Targn-1`: Code Block n-1

Approximate Translation

```c
target = JTab[x];
goto *target;
```
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.
Supplied by CMU, but converted to x86-64.
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

```
.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
```

Supplied by CMU, but converted to x86-64.
Supplied by CMU, but converted to x86-64.
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;
    ....
}
```

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

    case 3:
        w = 1;

    merge:
        w += z;
```

Supplied by CMU, but converted to x86-64.
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 is stored 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.
The diagram illustrates the `idivq` instruction in computer systems. It involves two registers: `rdx` and `rax`. The `idivq` instruction takes a `rcx` register as a divisor. The result consists of a `remainder` stored in `rdx` and a `quotient` stored in `rax`. The diagram shows the division process and the flow of data between these registers.
```
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]
```

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`).
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”.

```
0x4005c0: 0x0000000000000000 0x0000000000000000
0x4005d0: 0x0000000000000000 0x0000000000000000
0x4005e0: 0x0000000000000000 0x0000000000000000
0x4005f0: 0x0000000000000000 0x0000000000000000
```
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
0x00000000004004bc <switch_eg+16>:      mov   %rsi,%rax
0x00000000004004b0 <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>:      div   %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
0x00000000004004e0 <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 2

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

```
movl $0, %eax
.L2:
    movl %eax, a(%rax,4)
    addq $1, %rax
    cmpq $10, %rax
    jne .L2
    ret
```

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

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

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

a b c