x86-64 Programming II: Control

Brad Karp
UCL Computer Science

CS 3007
16th January 2018

(lecture notes derived from material from Phil Gibbons, Randy Bryant, and Dave O’Hallaron)
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements
Recall: ISA = Assembly/Machine Code View

Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction

- **Register file**
  - Heavily used program data

- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

**Memory**

- Byte-addressable array
- Code and user data
- Stack to support procedures
Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (%rax, ... )
  - Location of runtime stack (%rsp)
  - Location of current code control point (%rip, ... )
  - Status of recent tests (CF, ZF, SF, OF)

### Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%r8</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

Current stack top

**Instruction pointer**

**Condition codes**

| CF | ZF | SF | OF |
Aside/Review: Two’s Complement Representation

- Computers manipulate signed and unsigned integers

- **Unsigned integers**
  - range represented with k bits: \([0 – 2^{k-1}]\)
  - representation: same as usual binary integer
  - e.g., 3 bits: \([0 – 7], [000, 001, ..., 110, 111]\)

- **Signed integers in two’s complement representation**
  - range represented with k bits: \([-2^{(k-1)}, 2^{(k-1)}-1]\)
  - representation:
    - most significant bit is *sign bit* (0 “positive,” 1 “negative”)
    - treat most significant bit as negative in value; other bits positive in value
  - e.g., 3 bits: \([-4, 3], [100, 101, 110, 111, 000, 001, 010, 011]\)

- **Pitfall:** same bits mean different things in unsigned and two’s complement, when MSB is 1!
Condition Codes (Implicit Setting)

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

- Implicitly set (as side effect) of arithmetic operations

  Example: \( \text{addq Src, Dest} \leftrightarrow t = a + b \)

  - **CF set** if carry/borrow out from most significant bit (unsigned overflow)
  - **ZF set** if \( t = 0 \)
  - **SF set** if \( t < 0 \) (\( t \) interpreted as signed by CPU; may not be in C!)
  - **OF set** if two’s-complement (signed) overflow
    \( (a>0 \land b>0 \land t<0) \lor (a<0 \land b<0 \land t>0) \)

- **Not set by leaq instruction**
For unsigned arithmetic, CF set indicates overflow.
For unsigned arithmetic, CF set indicates overflow
For unsigned arithmetic, CF set indicates overflow.
For signed arithmetic, SF indicates when result is a negative number
OF set when

\[ y \ldots + y \ldots = z \ldots \]

\[ z = \sim y \]

For signed arithmetic, OF set indicates overflow.
ZF set when

000000000000...00000000000
Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpq Src2, Src1`
  - `cmpq b,a` like computing `a−b` without setting destination
  
  - **CF set** if carry/borrow 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
    
    \[(a>0 \&\& b<0 \&\& (a−b)<0) \|\| (a<0 \&\& b>0 \&\& (a−b)>0)\]

[784x19]13
[701x571]Carnegie Mellon
Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - `testq Src2, Src1`
    - `testq b,a` like computing `a&b` without setting destination
  - Sets condition codes based on value of `Src1 & Src2`
  - Useful to have one of the operands be a mask
    - **ZF set** when `a&b == 0`
    - **SF set** when `a&b < 0`
Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - `testq Src2, Src1`
    - `testq b,a` like computing `a&b` without setting destination
  - Sets condition codes based on value of `Src1 & Src2`
  - Useful to have one of the operands be a mask

- **ZF set** when `a&b == 0`
- **SF set** when `a&b < 0`

Common idiom:

```
testq %rax,%rax
```
Reading Condition Codes

- **setX Instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

<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) ~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>settle</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>
Reading Condition Codes

setX Instructions

- Set low-order byte of destination to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes

<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>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>Less or Equal (Signed)</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>

N.B. condition codes typically read implicitly in other instructions, not with setX instructions
## x86-64 Integer Registers

| %rax | %al |
| %rbx | %bl |
| %rcx | %cl |
| %rdx | %dl |
| %rsi | %sil |
| %rdi | %dil |
| %rsp | %spl |
| %rbp | %bpl |

| %r8  | %r8b |
| %r9  | %r9b |
| %r10 | %r10b |
| %r11 | %r11b |
| %r12 | %r12b |
| %r13 | %r13b |
| %r14 | %r14b |
| %r15 | %r15b |

- Can reference low-order byte
Reading Condition Codes (Cont’d)

- **SetX Instructions:**
  - Set single byte based on combination of condition codes

- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use `movzbl` to finish job
    - 32-bit instructions also set upper 32 bits to 0

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```assembly
cmpq %rsi, %rdi # Compare x:y
setg %al       # Set when >
movzbl %al, %eax # Zero rest of %rax
ret
```
Reading Condition Codes (Cont’d)

- **SetX Instructions:**
  - Set single byte based on combination of condition codes

- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use `movzbl` to finish job
    - 32-bit instructions also set upper 32 bits to 0

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```asm
cmpq    %rsi, %rdi       # Compare x:y
setg    %al              # Set when >
movzbl  %al, %eax        # Zero rest of %rax
ret
```
Reading Condition Codes (Cont.)

- **SetX Instructions:**
  - Set single byte based on combination of condition codes

- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use `movzbl` to finish job
    - 32-bit instructions also set upper 32 bits to 0

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```c
cmpq    %rsi, %rdi  # Compare x:y
setg    %al        # Set when >
movzbl  %al, %eax  # Zero rest of %rax
ret     
```
Reading Condition Codes (Cont.)

Beware weirdness `movzbl` (and others)

```
setg %al          # Set when >
movzbl %al, %eax  # Zero rest of %rax
ret
```

Use(s)
- Argument `x`
- Argument `y`
- Return value

```
movzbl %al, %eax  # Zero rest of %rax
```
Reading Condition Codes (Cont.)

Beware weirdness `movzbl` (and others)

`movzbl %al, %eax`

Zapped to all 0's

```
setg %al          # Set when >
movzbl %al, %eax  # Zero rest of %rax
ret
```
Reading Condition Codes (Cont.)

Beware weirdness `movzbl` (and others)

```
movzbl %al, %eax
```

Zapped to all 0’s

```
setg %al  # Set when >
movzbl %al, %eax  # Zero rest of %rax
ret
```
Today

- Control: Condition codes
- **Conditional branches**
- Loops
- **Switch Statements**
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>Less or Equal (Signed)</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>
Conditional Branch Example (Old Style)

### Generation

shark> gcc -Og -S -fno-if-conversion control.c

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

```
absdiff:
    cmpq %rsi, %rdi      # x:y
    jle .L4
    movq %rdi, %rax
    subq %rsi, %rax
    ret
.L4:             # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Conditional Branch Example (Old Style)

Generation

shark> gcc -Og -S -fno-if-conversion

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

absdiff:
  cmpq %rsi, %rdi       # x:y
  jle .L4
  movq %rdi, %rax
  subq %rsi, %rax
  ret
.L4:       # x <= y
  movq %rsi, %rax
  subq %rdi, %rax
  ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Expressing with Goto Code

- C allows `goto` statement
- Jump to position designated by label

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

```c
long absdiff_j
 (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```
General Conditional Expression Translation (Using Branches)

C Code

```c
val = Test ? Then_Expr : Else_Expr;
```

```c
val = x>y ? x-y : y-x;
```

Goto Version

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

- Create separate code regions for then & else expressions
- Execute appropriate one
Using Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    - if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe

- **Why?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```c
val = Test
? Then_Expr
: Else_Expr;
```

Goto Version

```c
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```
Conditional Move Example

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

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

### absdiff:

- `movq %rdi, %rax` # x
- `subq %rsi, %rax` # result = x-y
- `movq %rsi, %rdx`
- `subq %rdi, %rdx` # eval = y-x
- `cmpq %rsi, %rdi` # x:y
- `cmovle %rdx, %rax` # if <=, result = eval
- `ret`
Conditional Move Example

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

When is this bad?

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```
absdiff:
    movq   %rdi, %rax  # x
    subq   %rsi, %rax  # result = x-y
    movq   %rsi, %rdx
    subq   %rdi, %rdx  # eval = y-x
    cmpq   %rsi, %rdi  # x:y
    cmovle %rdx, %rax  # if <=, result = eval
    ret    |
```
Bad Cases for Conditional Move

Expensive Computations

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

- Both values get computed
- Only makes sense when computations are very simple
Bad Cases for Conditional Move

Expensive Computations

\[ \text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) \ : \ \text{Hard2}(x) ; \]

- Both values get computed
- Only makes sense when computations are very simple

Bad Performance
Bad Cases for Conditional Move

Expensive Computations

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

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

val = p ? *p : 0;

- Both values get computed
- May have undesirable effects
Bad Cases for Conditional Move

Expensive Computations

\[
\text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \ \text{Hard2}(x);
\]

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

\[
\text{val} = p \ ? \ *p : 0;
\]

- Both values get computed
- May have undesirable effects

Bad Performance
Unsafe
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
Bad Cases for Conditional Move

Expensive Computations

\[ \text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \ \text{Hard2}(x); \]
- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

\[ \text{val} = p \ ? \ \ast p : 0; \]
- Both values get computed
- May have undesirable effects

Computations with side effects

\[ \text{val} = x > 0 \ ? \ x*=7 : x+=3; \]
- Both values get computed
- Must be side-effect free
Exercise

<table>
<thead>
<tr>
<th>xorq</th>
<th>%rax, %rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>subq</td>
<td>$1, %rax</td>
</tr>
<tr>
<td>cmpq</td>
<td>$2, %rax</td>
</tr>
<tr>
<td>setl</td>
<td>%al</td>
</tr>
<tr>
<td>movzblq</td>
<td>%al, %eax</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%rax</th>
<th>SF</th>
<th>CF</th>
<th>OF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x0000 0000 0000 0001</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: setl and movzblq do not modify condition codes
Exercise

cmpq $b, $a like computing $a - $b without setting dest

- **CF set** if carry/borrow 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

<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>setsns</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>Less or Equal (Signed)</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>

<table>
<thead>
<tr>
<th>%rax</th>
<th>SF</th>
<th>CF</th>
<th>OF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td>xorq %rax, %rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subq $1, %rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmpq $2, %rax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>setl %al</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>movzblq %al, %eax</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: **setl** and **movzblq** do not modify condition codes
Exercise

cmpq b, a like computing a - b without setting dest

■ CF set if carry/borrow 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

<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>Less or Equal (Signed)</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>

<table>
<thead>
<tr>
<th>%rax</th>
<th>SF</th>
<th>CF</th>
<th>OF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000 0000 0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: setl and movzblq do not modify condition codes
Exercise

cmpq b,a like computing a-b without setting dest

- CF set if carry/borrow 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

<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>setsns</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>Less or Equal (Signed)</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>

<table>
<thead>
<tr>
<th>%rax</th>
<th>SF</th>
<th>CF</th>
<th>OF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000 0000 0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0xFFFF FFFF FFFF FFFF</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: setl and movzblq do not modify condition codes
Exercise

cmpq b,a like computing a-b without setting dest

■ **CF set**  if carry/borrow 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

<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>setns</td>
<td>SF</td>
<td>Negative</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>Less or Equal (Signed)</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>

<table>
<thead>
<tr>
<th>%rax</th>
<th>SF</th>
<th>CF</th>
<th>OF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000 0000 0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0xFFFF FFFF FFFF FFFF</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0xFFFF FFFF FFFF FFFF</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Note:** setl and movzblq do not modify condition codes
Exercise

cmpq b, a like computing a - b without setting dest

- **CF set** if carry/borrow 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

<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>Less or Equal (Signed)</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>

<table>
<thead>
<tr>
<th>%rax</th>
<th>SF</th>
<th>CF</th>
<th>OF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0000 0000 0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0xFFFF FFFF FFFF FFFF</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0xFFFF FFFF FFFF FFFF</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0xFFFF FFFF FFFF FF01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: setl and movzblq do not modify condition codes
Exercise

cmpq b, a like computing a − b without setting dest

- **CF set** if carry/borrow 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

<table>
<thead>
<tr>
<th>cmpq</th>
<th>$2, %rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>setl</td>
<td>%al</td>
</tr>
<tr>
<td>movzblq</td>
<td>%al, %eax</td>
</tr>
</tbody>
</table>

CF set if carry/borrow 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

<table>
<thead>
<tr>
<th>xorq</th>
<th>%rax, %rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>subq</td>
<td>$1, %rax</td>
</tr>
</tbody>
</table>

Note: setl and movzblq do not modify condition codes

<table>
<thead>
<tr>
<th>Comp</th>
<th>SF</th>
<th>CF</th>
<th>OF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td>setl</td>
<td>0x0000 0000 0000 0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>setl</td>
<td>0xFFFF FFFF FFFF FFFF</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>setl</td>
<td>0xFFFF FFFF FFFF FFFF</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>setl</td>
<td>0xFFFF FFFF FFFF FF01</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>setl</td>
<td>0x0000 0000 0000 0001</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

SetX | Condition | Description
---|------------|----------------|
sete | ZF | Equal / Zero
setne | ~ZF | Not Equal / Not Zero
sets | SF | Negative
setsn | ~SF | Nonnegative
setg | ~(SF^OF) & ~ZF | Greater (Signed)
setge | ~(SF^OF) | Greater or Equal (Signed)
setl | (SF^OF) | Less (Signed)
setle | (SF^OF) | Less or Equal (Signed)
seta | ~CF & ~ZF | Above (unsigned)
setb | CF | Below (unsigned)
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements
“Do-While” Loop Example

C Code

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

Goto Version

```c
long pcount_goto
    (unsigned long x) {
    long 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 to either continue looping or to exit loop
“Do-While” Loop Compilation

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

```
movl $0, %eax  # result = 0
 .L2:          # loop:
    movq %rdi, %rdx
    andl $1, %edx   # t = x & 0x1
    addq %rdx, %rax  # result += t
    shrq %rdi       # x >>= 1
    jne .L2        # if(x) goto loop
    rep; ret
```
General “Do-While” Translation

C Code

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

Goto Version

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

**Body:**

```
{ 
   Statement_1;
   Statement_2;
   ...
   Statement_n;
}
```
General “While” Translation #1

- “Jump-to-middle” translation
- Used with -Og

While version

```c
while (Test)
  Body
```

Goto Version

```c
goto test;
loop:
  Body
test:
    if (Test)
      goto loop;
done:
```
While Loop Example #1

C Code

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

Jump-to-Middle Version

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

- Compare to do-while version of function
- Initial goto starts loop at test
General “While” Translation #2

- “Do-while” conversion
- Used with –O1

**While version**

```c
while (Test)  
  Body
```

**Do-While Version**

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

**Goto Version**

```c
if (!Test)  
goto done;  
loop:  
  Body
  if (Test)  
goto loop;  
done:
```
While Loop Example #2

C Code

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

Do-While Version

```c
long pcount_goto_dw(unsigned long x) {
    long result = 0;
    if (!x) goto done;
    loop:
        result += x & 0x1;
        x >>= 1;
    if (x) goto loop;
    done:
        return result;
}
```
While Loop Example #2

C Code

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

Do-While Version

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

- Initial conditional guards entrance to loop
- Compare to do-while version of function
  - Removes jump to middle. When is this good or bad?
"For" Loop Form

General Form

\[
\text{for (Init; Test; Update)} \\
\quad \text{Body}
\]

```c
#define WSIZE 8*sizeof(int)
long pcount_for
    (unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```
“For” Loop $\rightarrow$ While Loop

For Version

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

While Version

```
Init;
while (Test) {
  Body
  Update;
}
```
For-While Conversion

Init
\[ i = 0 \]

Test
\[ i < \text{FSIZE} \]

Update
\[ i++ \]

Body
\[
\begin{align*}
\text{unsigned bit} &= (x >> i) \& 0x1; \\
\text{result} &= \text{result} + \text{bit}; \\
\end{align*}
\]

long pcount_for_while
(\text{unsigned long} \ x)
\{
\text{size_t} \ i;
\text{long} \ \text{result} = 0;
i = 0;
\text{while} (i < \text{FSIZE})
\{
\begin{align*}
\text{unsigned bit} &= (x >> i) \& 0x1; \\
\text{result} &= \text{result} + \text{bit}; \\
i &= i + 1;
\end{align*}
\}\text{return} \ \text{result};
\}
"For" Loop Do-While Conversion

C Code

long pcount_for (unsigned long x) {
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
    }
    return result;
}

Goto Version

long pcount_for_goto_dw (unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (!((i < WSIZE))
        goto done;
    loop:
    {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
    }
    i++;
    if (i < WSIZE)
        goto loop;
    done:
    return result;
}

- Initial test can be optimized away
"For" Loop Do-While Conversion

C Code

```c
long pcount_for_for (unsigned long x) {
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Goto Version

```c
long pcount_for_goto_dw (unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (!((i < WSIZE))
        goto done;
    loop:
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    i++;
    if (i < WSIZE)
        goto loop;
    done:
    return result;
}
```

- Initial test can be optimized away
Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements
long my_switch
  (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

- \( \text{jtab:} \)
  - 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 \( n-1 \)

Translation (Extended C)

```c
goto *JTab[x];
```
Switch Statement Example

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

**Setup**

my_switch:
    movq %rdx, %rcx
    cmpq $6, %rdi  # x:6
    ja .L8
    jmp *.L4(,%rdi,8)

*What range of values takes default?*

Note that w not initialized here

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Switch Statement Example

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

### Jump table

```
.section .rodata
.align 8
.L4:
.quad .L8  # x = 0
.quad .L3  # x = 1
.quad .L5  # x = 2
.quad .L9  # x = 3
.quad .L8  # x = 4
.quad .L7  # x = 5
.quad .L7  # x = 6
```

**Setup**

```asm
my_switch:
    movq  %rdx, %rcx
    cmpq  $6, %rdi      # x:6
    ja    .L8          # use default
    jmp   *.L4(%rdi,8)  # goto *Jtab[x]
```

*Indirect jump*
Assembly Setup Explanation

- **Table Structure**
  - Each target requires 8 bytes
  - Base address at `.L4`

- **Jumping**
  - **Direct**: `jmp .L8`
  - Jump target is denoted by label `.L8`
  - **Indirect**: `jmp * .L4 (,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address `.L4 + x*8`
    - Only for $0 \leq x \leq 6$

Jump table

```assembly
.section .rodata
.align 8
.L4:
    .quad .L8  # x = 0
    .quad .L3  # x = 1
    .quad .L5  # x = 2
    .quad .L9  # x = 3
    .quad .L8  # x = 4
    .quad .L7  # x = 5
    .quad .L7  # x = 6
```
Jump Table

Jump table

```
.switch(x) {
  case 1:      // .L3
    w = y*z;
    break;
  case 2:      // .L5
    w = y/z;
    /* Fall Through */
  case 3:      // .L9
    w += z;
    break;
  case 5:
  case 6:      // .L7
    w -= z;
    break;
  default:     // .L8
    w = 2;
}
```
Code Blocks (x == 1)

```c
switch(x) {
    case 1: // .L3
        w = y*z;
        break;
    ...
}
```

.L3:
```
    movq %rsi, %rax # y
    imulq %rdx, %rax # y*z
    ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
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;
```
Code Blocks (x == 2, x == 3)

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

```
.L5:                  # Case 2
    movq  %rsi, %rax  
    cqto
    # sign extend
    # rax to rdx:rax
    idivq %rcx        # y/z
    jmp  .L6         # goto merge
.L9:                  # Case 3
    movl $1, %eax    # w = 1
    .L6:              # merge:
    addq %rcx, %rax  # w += z
    ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rcx</td>
<td>z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Code Blocks (x == 5, x == 6, default)

```c
switch(x) {
    . . .
    case 5: // .L7
    case 6: // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

### .L7:
- `movl $1, %eax` # w = 1
- `subq %rdx, %rax` # w -= z
- `ret`

### .L8:
- `movl $2, %eax` # 2
- `ret`

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Control Idioms: Summary

- **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)
Roadmap

- **Today**
  - Control: Condition codes
  - Conditional branches & conditional moves
  - Loops
  - Switch statements

- **Up Next**
  - Stack
  - Call / return
  - Procedure call discipline