CSCI-UA.0201

Computer Systems Organization

Machine Level – Control

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
Control
Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (%rax, ...)
  - Location of runtime stack (%rsp, %rbp)
  - 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>

### Instruction pointer

- %rip

### Condition codes

- CF  
- ZF  
- SF  
- OF
Setting Condition Codes Implicitly

• Can be implicitly set by arithmetic operations

Example: `addq Src, Dest (t = a+b)`

**CF (Carry flag) set** if carry out from most significant (31-st) bit (unsigned overflow)

**ZF (Zero flag) set** if \( t == 0 \)

**SF (Sign flag) set** if \( t < 0 \) (as signed)

**OF (Overflow flag) set** if signed overflow

\[ (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0) \]

• Condition codes not set by `lea` instruction!
Setting Condition Codes Explicitly

- Can also be explicitly set

```
cmpl b,a
```
set condition codes based on computing $a-b$ without storing the result in any 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 $(a-b)$ results in signed overflow
Reading Condition Codes

- **setX dest**

Sets the **lower byte** of *dest* based on combinations of condition codes and does not alter remaining 7 bytes. Destination can also be memory location.

<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>_sets1</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>

These instructions are usually used after a comparison.
Recall: x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%al</th>
<th>%r8</th>
<th>%r8b</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%bl</td>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%rdi</td>
<td>%dil</td>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>

– Can reference low-order byte
Example

```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>%eax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```
cmpq     %rsi, %rdi    # Compare x:y
setg     %al           # Set when >
movzbl   %al, %eax     # Zero rest of %eax
ret       |
```
What do we do with condition codes?

1. Setting a single byte to 0 or 1 based on some combination of the condition codes.

2. Conditionally jump to other parts of the program.

3. Conditionally transfer data.
# 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>
Indirect jump

jmp * Operand

Unconditional jump

Can be:

- register
- Memory address using any of the addressing modes we saw.
Example

```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>
What do we do with condition codes?

1. Setting a single byte to 0 or 1 based on some combination of the condition codes.
2. Conditionally jump to other parts of the program.
3. Conditionally transfer data.
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
```
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

Computations with side effects

\[
\text{val} = x > 0 \ ? \ x *= 7 : x += 3;
\]

- Both values get computed
- Must be side-effect free
What we have seen so far ...

• The arithmetic and logic operations can be applied to data of size 1(b), 2(w), 4(l), and 8(q) bytes.

• Condition codes are needed in order to implement conditional branches/jumps.

• The compiler uses the different condition codes and different jump formats to implement the different control structures we have in high-level languages: for-loop, do-while, while, switch, if-then-else, etc.
How does the compiler translate loops?

• do-while
• while
• for
“Do-While” Loop Compilation

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 1s in argument `x`
- Use conditional branch to either continue looping or to exit loop
“Do-While” Loop Compilation

<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>

Goto Version

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

Assembly Version

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

C Code

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

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```
General “While” Translation #1

• “Jump-to-middle” translation

While version

```
while (Test)
    Body
```

Goto Version

```
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

```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

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:
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;
}
```

• Compare to do-while version of function
• Initial conditional guards entrance to loop