Machine-Level Representation: Control

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

Outline

• Control: Condition codes
• Conditional branches
• Loops
• Switch Statements

Review: Assembly/Machine Code View

Programmer-Visible State

- PC: Program counter
  - Address of next instruction
  - Called %rip in x86-64
- Register file
  - Heavily used program data
  - Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

Processor State (x86-64, Partial)

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

Condition Codes (Implicit Setting)

- Single bit registers
  - CF: Carry Flag (for unsigned)
  - SF: Sign Flag (for signed)
- Implicitly set (think of it as side effect) by arithmetic operations
  - Example: addq Src, Dest ← t = a+b
  - CF set if carry is set from most significant bit (unsigned overflow)
  - SF set if t < 0 (as signed)
  - OF set if two’s-complement (signed) overflow
  - Not set by leaq instruction

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 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
  - Not set by leaq instruction
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`

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

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><code>ZF</code></td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~<code>ZF</code></td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>setl</td>
<td><code>SF</code></td>
<td>Negative</td>
</tr>
<tr>
<td>setn</td>
<td>~<code>SF</code></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td><code>SF</code>&amp;<del><code>OF</code>&amp;</del><code>ZF</code></td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td><code>SF</code>&amp;~<code>OF</code></td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td><code>SF</code>&amp;<code>OF</code></td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td><code>SF</code>&amp;<code>OF</code>&amp;<code>ZF</code></td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td><del><code>CF</code>&amp;</del><code>ZF</code></td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td><code>CF</code> &amp; <code>ZF</code></td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

x86-64 Integer Registers

- `rax` `%ax`
- `rbx` `%bx`
- `rcx` `%cx`
- `rdx` `%dx`
- `rsi` `%si`
- `rdi` `%di`
- `rsp` `%sp`
- `rbp` `%bp`

- Can reference low-order byte

x86-64 Integer Registers (Cont.)

- Can reference low-order byte

Announcement 2/12/2016

- Homework assignment #2 will be issued today, due Wednesday 2/24/2016 before the class
- Bomb Lab has been issued on Wednesday 2/10/2016, due 11:55pm Friday 2/26/2016

Outline

- 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>X</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>i+</td>
<td>0</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>i+</td>
<td>1</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>i</td>
<td>0</td>
<td>Negative</td>
</tr>
<tr>
<td>i+</td>
<td>1</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>j</td>
<td>(SF^OF) 0</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>j+</td>
<td>(SF^OF) 1</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>j+</td>
<td>(SF^OF) 0</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>j</td>
<td>(SF^OF) 1</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>i+</td>
<td>0</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Expressing with Goto Code

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

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

Conditional Branch Example (Old Style)

- **Generation**
  - gcc –Og –S –fno-if-conversion control.c

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

General Conditional Expression Translation (Using Branches)

<table>
<thead>
<tr>
<th>C Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val = Test ? Then_Expr : Else.Expr;</code></td>
</tr>
</tbody>
</table>

```
Val = x>y ? x-y : y-x;
```

Goto Version

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

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

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

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

Conditional Move Example

- **C Code**
  - `val = Test ? Then_Expr : Else.Expr;`

```
Goto Version
result = Then_Expr;
val = Else.Expr;
nt = !Test;
if (!nt) result = val;
return result;
```

```
absdiff:  
movq tedi, trax # x
subq tedi, trax # result = x-y
movq tedi, tdi  # y
subq tedi, tdx  # result = y-x
cmpq tedi, tdi # if <=, result = eval
ret
```
Bad Cases for Conditional Move

Expensive Computations

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

- 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

Outline

- 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
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

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

General “Do-While” Translation

C Code

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

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

Goto Version

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

General “While” Translation #1

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

```
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;
}
```

• Compare to do-while version of function
• Initial goto starts loop at test

Jump to Middle

General “While” Translation #2

While version

```c
while (Test) {
    Body
}
```

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

Do-While Version

```c
if (!Test) {
    goto done;
}
loop:
    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;
}
```

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

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;
}
```

“For” Loop Example #1

```c
#define WSIZE \8
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 Form

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

General Form

- `Init`
- `Test`
- `Update`
- `Body`

For Version

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

While Version

```c
while (Test) {
    Body
}
```

For-While Conversion

```c
long pcount_for_while (unsigned long x) {
    int i;
    long result = 0;
    while (i < WSIZE) {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
        i++;
    }
    return result;
}
```
Outline

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

Switch Statement Example

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

Jump Table Structure

<table>
<thead>
<tr>
<th>Switch Form</th>
<th>Jump Table</th>
<th>Jump Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch(x) {</td>
<td>jtab: Targ0: Code Block 0</td>
<td></td>
</tr>
<tr>
<td>case val_0:</td>
<td>Targ1: Code Block 1</td>
<td></td>
</tr>
<tr>
<td>case val_1:</td>
<td>Targ2: Code Block 2</td>
<td></td>
</tr>
<tr>
<td>case val_n-1:</td>
<td>Targn-1: Code Block n-1</td>
<td></td>
</tr>
</tbody>
</table>

Setup:
```
switch_mg: 
    movq %rdx, %rcx
    cmpq $6, %rdi   # x:6
    ja .L8           # Use default
    jmp *._L4(.rdi, 8) # goto *jTab[x]
```

Switch Statement Example

```c
long switch_eg (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case val_0:
            Block 0
        case val_1:
            Block 1
        . . .
        case val_n-1:
            Block n-1
    }
    return w;
}
```

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 ≤ x ≤ 6
Jump Table

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

Carnegie Mellon

```
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;
}
```

Handling Fall-Through

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

Carnegie Mellon

```
Code Blocks (x == 1)

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

Code Blocks (x == 2, x == 3)

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

Code Blocks (x == 5, x == 6, default)

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

Summarizing

- 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)
Overview

• What We Have Learned
  • Control: Condition codes
  • Conditional branches & conditional moves
  • Loops
  • Switch statements
• What Comes Next
  • Stack
  • Call / return
  • Procedure call discipline