Machine-Level Programming II: Control

CSci 2021: Machine Architecture and Organization
September 26th-28th, 2018

Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O'Hallaron

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

Processor State (x86-64, Partial)

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

- Registers
  - %rax
  - %rbx
  - %rcx
  - %rdx
  - %rsi
  - %rdi
  - %r8
  - ... (more registers)

- Current stack top

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: `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; i.e., copy of sign bit)
- 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: 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
  - (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (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

- CF set when a&b == 0
- SF set when a&b < 0
Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte to 0 or 1 based on condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>SF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~SF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>setls</td>
<td>SF</td>
<td>Nonnegative</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 &gt;)</td>
</tr>
<tr>
<td>setae</td>
<td>~CF</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe</td>
<td>CF</td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>

- **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;
}
```

Exercise Break: More Conditions

- Every condition can be negated by putting “n” in the mnemonic, for “not”
  - We skipped some of these conditions in the previous table, because they were equivalent to others
  - Which other conditions are these equivalent to?

1. setng: not greater than
2. setnbe: not below or equal

Equivalents of More Conditions

- Intuition: cover three cases: <, =, >
- **setng not greater than (signed)**
  - If not greater, than either less than or equal: setle
  - Check conditions:
    - `~(SF^OF) & ~ZF = ~(SF^OF) & ~ZF = (SF^OF) | ZF`

- **setnbe not below or equal (unsigned)**
  - If not below or equal, must be above: seta
  - Check conditions:
    - `(CF | ZF) = ~CF & ~ZF`

Logistics announcement: HA2, midterm 1

- Hands-on assignment 2, on data operations, is available now
  - Continuation of this week’s lab, but with different and more challenging puzzles; an individual assignment
- HA2 due date moved to Friday, October 12th
  - HA3 will also probably move a few days as well
  - I still recommend you start working on HA2 before the midterm
  - HA2 is challenging in different ways than HA1
- Midterm 1 is Monday, October 8th
  - Covers C basics, data representation, and machine code up through today’s lecture
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 Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
</tr>
<tr>
<td>jg</td>
<td>~SF^OF &amp; ~ZF</td>
</tr>
<tr>
<td>jge</td>
<td>~SF^OF</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
</tr>
</tbody>
</table>

Conditional Branch

```c
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)

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

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

Using Conditional Moves

- **Conditional Move Instructions**

<table>
<thead>
<tr>
<th>C Code</th>
<th>Goto Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val = Test ? Then_Expr : Else_Expr;</code></td>
<td><code>ntest = !Test; if (!ntest) goto Else; val = Then_Expr; goto Done; Else: val = Else_Expr; Done: ...</code></td>
</tr>
</tbody>
</table>

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

```
long absdiff (long x, long y)
{
    long result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    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>%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  # eval = y-x
    cmpq %rsi, %rdx  # x:y
    cmovle %rdx, %rax  # if <=, result = eval
    ret
```

Bad Cases for Conditional Move

Expensive Computations

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

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

Risk Computations

- Both values get computed
- May have undesirable effects
- Computations with side effects

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

Today

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

“Do-While” Loop Example

C Code

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

Goto Version

```
l/op 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

Goto Version

```
l/op pcount goto (unsigned long x) {
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
        return result;
}
```

```
    movl  $0, %eax  # result = 0
    L2:
        # loop:
        movq %rdi, %rdx  # t = x & 0x1
        addq %rdx, %rax  # result += t
        shrq %rdi  # x >>= 1
    jne  L2  # if (x) goto loop
    rep: ret  # synonym of “ret”
```

General “Do-While” Translation

C Code

```
do Body (Test);

    while (Test)
    {
        Statement1;
        Statement2;
        ...
        Statementn;
    }
```

Goto Version

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

- Body:

  - Statement1;
  - Statement2;
  - ...
  - Statementn;

- ```ret```

Register Use(s)

- %rdi: Argument x
- %rax: Result
**General “While” Translation #1**
- “Jump-to-middle” translation
- Used with –Og

**While version**
```
while (Test)
  Body
```

**Goto Version**
```
goto test;
loop:
  Body
  test:
    if (Test) goto loop;
  done:
```

**While Loop Example #1**
- Compare to do-while version of function
- Initial goto starts loop at test

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

**Jump to Middle**
```
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;
}
```

**General “While” Translation #2**
- “Do-while” conversion
- Used with –O1

**While version**
```
while (Test)
  Body
```

**Do-While Version**
```
if (!Test)
  goto done;
loop:
  Body
  if (Test) goto loop;
  done:
```

**Goto Version**
```
if (!Test)
  goto done;
loop:
  Body
  if (Test) goto loop;
  done:
```

**While Loop Example #2**
- Compare to do-while version of function
- Initial conditional guards entrance to loop

**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;
}
```

**Do-While Version**
```
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 Form**
```
for (Init; Test; Update)
  Body
```

**“For” Loop → While Loop**

**For Version**
```
for (Init; Test; Update)
  Body
```

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

Init
1 = 0

Test
i <= WSIZE

Update
i++

Body
unsigned bit = (x >> i) & 0x1;
result += bit;
}
return result;

“For” Loop Do-While Conversion

C Code

Goto Version

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

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

Switch Statement Example

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

Switch Statement Example

Setup:

<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>

What range of values go to LB?

Note that w not initialized here
Switch Statement Example

```c
long switch_eg(long x, long y, long z) {
    long w = 1;
    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:      // .L7
        case 6:      // .L7
            w -= z;
            break;
        default:     // .L8
            w = 2;
    }
    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

```c
.L4:
    .quad 0x0   # x = 0
    .quad 0x0   # x = 1
    .quad 0x0   # x = 2
    .quad 0x0   # x = 3
    .quad 0x0   # x = 4
    .quad 0x0   # x = 5
    .quad 0x0   # x = 6
```

### Code Blocks (x == 1)

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

Handling Fall-Through

```c
long w = 1;
switch(x) {
    . . .
    case 2:      // .L5
        w = y/z;
        /* Fall Through */
    case 3:      // .L9
        w += z;
        break;
    . . .
}
```

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

```c
long w = 1;
switch(x) {
    . . .
    case 2:      // .L5
        w = y/z;
        /* Fall Through */
    case 3:      // .L9
        w += z;
        break;
    . . .
}
```

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>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```

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

switch(x) {
    . . .
    case 5: // .L7
        movl $1, %eax # w = 1
        subq %rdx, %rax # w -= z
        ret
    .L7: # Case 5,6
        movl $2, %eax # w = 2
        ret
    case 6: // .L7
        w = z;
        break;
    .L8: # Default:
        movl $2, %eax # w = 2
        ret
    default: // .L8
        w = z;
        break;
}

Exercise Break: switch Bounds

- Every jump table needs to check that the index is in bounds
  - For each of these code patterns, what indexes are allowed?

```
cmpq $5, %rax
ja .Ldefault
jmp *.%L1(,%rax,8)
andq $7, %rax
jmp *.%L2(,%rax,8)
movzlq %8(%rbp), %eax
jmp *.%L3(,%rax,8)
```

https://chimein.cla.umn.edu/course/view/2021

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-else-if-else)

Summary

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

- Next Time
  - Stack
  - Call / return
  - Procedure call discipline