Machine-Level Representation: Control

CSCI 2021: Machine Architecture and Organization

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

With Slides from Bryant, O’Hallaron
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

- **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

| \%rax | \%r8  |
| \%rbx | \%r9  |
| \%rcx | \%r10 |
| \%rdx | \%r11 |
| \%r8  | \%r12 |
| \%r9  | \%r13 |
| \%r10 | \%r14 |
| \%r11 | \%r15 |

\%rsp
\%rbp
\%rip

Instruction pointer

Condition codes

Current stack top

4
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)
  - OF set if two’s-complement (signed) overflow
    \[(a>0 \&\& b>0 \&\& t<0) \mid \mid (a<0 \&\& b<0 \&\& t>=0)\]

- 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
  
  \[(a > 0 \land b < 0 \land (a - b) < 0) \lor (a < 0 \land b > 0 \land (a - b) > 0)\]
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`
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) &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>
# x86-64 Integer Registers

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

- Can reference **low-order byte**
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>

```assembly
cmpq   %rsi, %rdi   # Compare x:y
setg   %al           # Set when >
movzbl %al, %eax     # Zero rest of %rax
ret
```
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>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

```
gcc -Og -S -fno-if-conversion control.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       #L4 is a label
  movq    %rdi, %rax
  subq    %rsi, %rax
  ret     #return %rax
.L4:
  # x <= y
  movq    %rsi, %rax
  subq    %rdi, %rax
  ret     #return %rax
```

<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 \texttt{goto} statement

• Jump to position designated by a \texttt{label}

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

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

C Code

```
val = Test ? Then_Expr : Else_Expr;

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

Goto Version

```
nest = !Test;
if (nent) 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:
    
    ```c
    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;
}

void 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
```

<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>
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
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
long 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:    
       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
```

23
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

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

\[
\text{while (Test)} \\
\text{Body}
\]

Do-While Version

\[
\text{if (!Test)} \\
goto \text{done;} \\
do \\
\text{Body} \\
\text{while(Test)}; \\
done:
\]

Goto Version

\[
\text{if (!Test)} \\
goto \text{done;} \\
\text{loop:} \\
\text{Body} \\
\text{if (Test)} \\
goto \text{loop;} \\
done:
\]

• “Do-while” conversion
• Used with –O1
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
“For” Loop Form

General Form

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

Init

```c
i = 0
```

Test

```c
i < WSIZE
```

Update

```c
i++
```

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 → While Loop

For Version

```plaintext
for (Init; Test; Update )
  Body
```

While Version

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

Init

\[ i = 0 \]

Test

\[ i < \text{WSIZE} \]

Update

\[ i++ \]

Body

\[
\begin{align*}
\text{long pcount_for_while} & \quad (\text{unsigned long } x) \\
\{ & \\
\text{size_t } i; & \\
\text{long } \text{result} = 0; & \\
i = 0; & \\
\text{while } (i < \text{WSIZE}) & \\
\{ & \\
\text{unsigned bit} = & \\
(x \gg i) \& 0x1; & \\
\text{result} += \text{bit}; & \\
i++; & \\
\} & \\
\text{return } \text{result}; & \\
\}
\end{align*}
\]
Outline

• Control: Condition codes
• Conditional branches
• Loops
• Switch Statements
Switch Statement - Example

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

<table>
<thead>
<tr>
<th>jtab:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ0</td>
</tr>
<tr>
<td>Targ1</td>
</tr>
<tr>
<td>Targ2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Targn-1</td>
</tr>
</tbody>
</table>

Jump Targets

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

Translation (Extended C)

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

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

**Setup:**

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

<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 takes default?

Note that w not initialized here
Switch Statement Example

`long switch_eg(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:

```
switch_eg:
    movq  %rdx, %rcx
    cmpq  $6, %rdi      # x:6
    ja    .L8           # Use default
    jmp   *.L4(%rdi,8)   # goto *JTab[x]
```
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

```
.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)

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>
long w = 1;
    ...
switch(x) {
    ...
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    ...
}
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  #convert to oct word
  idivq %rcx  # y/z
  jmp   .L6    # goto merge
.L6:                  # Case 3
  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>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value 41</td>
</tr>
</tbody>
</table>
Code Blocks (x == 5, x == 6, default)

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

.L7:       # Case 5,6
    movl  $1, %eax  # w = 1
    subq  %rdx, %rax # w -= z
    ret

.L8:       # Default:
    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>
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