Machine-Level Programming: Control

Lecture: 7

CMPU 224 – Computer Organization
Jason Waterman
### movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction*
Complete Memory Addressing Modes

• Most General Form

\[ D(R_b, R_i, S) \quad \text{Mem}[\text{Reg}[R_b] + S \times \text{Reg}[R_i] + D] \]

• D: Constant “displacement” signed integer
• \( R_b \): Base register: Any of 16 integer registers
• \( R_i \): Index register: Any, except for \( \% \text{rsp} \)
• S: Scale: 1, 2, 4, or 8
Address Computation Instruction

• **leaq** *Src, Dst*
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression

• Uses
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x + k*y`
    - `k = 1, 2, 4, or 8`

• Example

```c
long m12(long x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```asm
leaq (%rdi,%rdi,2), %rax  # t <- x+x*2
salq $2, %rax           # return t<<2
```
Some Arithmetic Operations

- Two Operand Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>( Src, Dest ) Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>( Src, Dest ) Dest = Dest − Src</td>
</tr>
<tr>
<td>imulq</td>
<td>( Src, Dest ) Dest = Dest * Src</td>
</tr>
<tr>
<td>salq</td>
<td>( Src, Dest ) Dest = Dest &lt;&lt; Src (Also called shlq)</td>
</tr>
<tr>
<td>sarq</td>
<td>( Src, Dest ) Dest = Dest &gt;&gt; Src (Arithmetic shift)</td>
</tr>
<tr>
<td>shrq</td>
<td>( Src, Dest ) Dest = Dest &gt;&gt; Src (Logical shift)</td>
</tr>
<tr>
<td>xorq</td>
<td>( Src, Dest ) Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>( Src, Dest ) Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>( Src, Dest ) Dest = Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order, \( \text{subq} \) in particular
- No distinction between signed and unsigned int (why?)
Some Arithmetic Operations

- One Operand Instructions
  - `incq Dest` \( Dest = Dest + 1 \)
  - `decq Dest` \( Dest = Dest - 1 \)
  - `negq Dest` \( Dest = -Dest \)
  - `notq Dest` \( Dest = \neg Dest \)

- See book for more instructions
Machine Programming: Summary

• History of Intel processors and architectures
  • Evolutionary design leads to many quirks and artifacts

• C, assembly, machine code
  • New forms of visible state: program counter, registers, ...
  • Compiler must transform statements, expressions, procedures into low-level instruction sequences

• Assembly Basics: Registers, operands, move
  • The x86-64 move instructions cover wide range of data movement forms

• Arithmetic
  • C compiler will figure out different instruction combinations to carry out computation
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)

<table>
<thead>
<tr>
<th>Registers</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rip</td>
<td>%rbp</td>
<td>%rsp</td>
<td>%rip</td>
</tr>
</tbody>
</table>

Condition codes

Current stack top

Instruction pointer
Condition Codes (Implicit Setting)

• Single bit registers set by arithmetic and logic operations
  • CF Carry Flag – The most recent operation generated a carry out of the MSB, overflow for unsigned operations (unsigned)
  • ZF Zero Flag – The most recent operation yielded zero
  • SF Sign Flag – The most recent operation yielded a negative value (signed)
  • OF Overflow Flag – The most recent operation caused a two’s-complement overflow, either positive or negative (signed)

• Implicitly set (think of it as side effect) by arithmetic/logic operations based on the result of the operation
  • For logical operations, the carry and overflow flags are set to zero
  • For shift operations, CF is set to the last bit shifted out, OF is set to zero
  • INC and DEC set OF and ZF, but leave the carry flag unchanged

• Not set by leaq instruction

• Condition Codes are not accessed directly, but some instructions alter their behavior based on the value of the Condition Codes
Comparing Two Numbers by Subtracting

• By subtracting two numbers you can compare them!
  • Example: $A - B$

• **Equality**: when $A$ and $B$ are equal, $A - B == 0$ (ZF)

• **Not Equal**: When $A - B != 0$ (~ZF)

• **Greater than**: when $A > B$, $A - B == $ Positive number and not zero (~SF & ~ZF)

• **Greater than or equal**: when $A >= B$, $A - B == $ Positive number or zero (~SF | ZF)

• **Less than**: when $A < B$, $A - B == $ Negative number (SF)

• **Less than or equal**: when $A <= B$, $A - B == $ Negative number or zero (SF | ZF)
Setting Condition Codes Explicitly with Compare

• **Compare Instruction:** \texttt{cmp } \( S_1, S_2 \)
  
  • Similar to the \texttt{sub} instruction
  
  • Sets the condition codes according to the differences of their two operands (\( S_2 - S_1 \)) but without setting the destination operand
  
  • Used to compare two numbers
  
  • Example: \texttt{cmpq b,a}\  \textcolor{red}{\textbf{swapped}}
  
  Read as: \( a \ comparator b \) (also as \( a:b \))

• Operands are reversed for a compare
  
  • Why? AT&T vs Intel assembler syntax
  
  • In Intel syntax operands are reversed
  
  • We use AT&T style syntax, so remember to switch the order of operands for compare
Reading Condition Codes (SetX instructions)

- **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>Instruction</th>
<th>Synonym</th>
<th>Effect</th>
<th>Set condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete D</td>
<td>setz</td>
<td>D ← ZF</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>setne D</td>
<td>setnz</td>
<td>D ← ~ZF</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>sets D</td>
<td></td>
<td>D ← SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns D</td>
<td></td>
<td>D ← ~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg D</td>
<td>setnle</td>
<td>D ← ~(SF ^ OF) &amp; ~ZF</td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td>setge D</td>
<td>setnl</td>
<td>D ← ~(SF ^ OF)</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>setl D</td>
<td>setnge</td>
<td>D ← SF ^ OF</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>setle D</td>
<td>setng</td>
<td>D ← (SF ^ OF)</td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>seta D</td>
<td>setnbe</td>
<td>D ← ~CF &amp; ~ZF</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>setae D</td>
<td>setnb</td>
<td>D ← ~CF</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>setb D</td>
<td>setnae</td>
<td>D ← CF</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>setbe D</td>
<td>setna</td>
<td>D ← CF</td>
<td>Below or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
Reading Condition Codes (Cont.)

• SetX Instructions:
  • Set single byte based on combination of condition codes; descriptions apply after a cmpq instruction – **remember to reverse your operands!**
  
• 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>

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

```c
int gt (long x, long y) {
  return x > y;
}
```
Test Instruction: \texttt{test S_1, S_2}

- Similar to the \texttt{and} instruction
- Sets the ZF and the SF based on \((S_2 \& S_1)\) but without setting the destination operand
- Often the same operand repeated (\texttt{testq \%rax, \%rax}) to check if the value is zero, positive, or negative
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Jump condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp Label</td>
<td>1</td>
<td></td>
<td>Direct jump</td>
</tr>
<tr>
<td>jmp *Operand</td>
<td>1</td>
<td></td>
<td>Indirect jump</td>
</tr>
<tr>
<td>je Label</td>
<td>jz</td>
<td>ZF</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>jne Label</td>
<td>jnz</td>
<td>~ZF</td>
<td>Not Equal / not zero</td>
</tr>
<tr>
<td>js Label</td>
<td>SF</td>
<td></td>
<td>Negative</td>
</tr>
<tr>
<td>jns Label</td>
<td>~SF</td>
<td></td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg Label</td>
<td>jnle</td>
<td>~(SF ^ OF) &amp; ~ZF</td>
<td>Greater (signed &gt; )</td>
</tr>
<tr>
<td>jge Label</td>
<td>jnl</td>
<td>~(SF ^ OF)</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>jl Label</td>
<td>jnge</td>
<td>SF ^ OF</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>jle Label</td>
<td>jng</td>
<td>(SF ^ OF)</td>
<td>Less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>ja Label</td>
<td>jnbe</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>jae Label</td>
<td>jnb</td>
<td>~CF</td>
<td>Above or equal (unsigned &gt;=)</td>
</tr>
<tr>
<td>jb Label</td>
<td>jnae</td>
<td>CF</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>jbe Label</td>
<td>jna</td>
<td>CF</td>
<td>Less or equal (unsigned &lt;=)</td>
</tr>
</tbody>
</table>
Expressing with test and 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;
    return result;
Else:
    result = y - x;
    return result;
}
```
Conditional Branch Example (Old Style)

- Generation

```bash
linux> gcc -Og -S -fno-if-conversion control.c
```

```c
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x - y;
    return result;
Else:
    result = y - x;
    return result;
}
```

### absdiff:
- `cmpq %rsi, %rdi` # x:y
- `jle .L4 Label` # Less Than or Equal
- `movq %rdi, %rax` %rsi, %rax
- `subq %rsi, %rax` %rsi, %rax
- `ret .L4:` # x <= y
- `movq %rsi, %rax` %rsi, %rax
- `subq %rdi, %rax` %rdi, %rax
- `ret` %rax

### 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>
Conditional Branch Example (Old Style)

• Generation

```bash
linux> 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;
}
```

Note: the jump condition is the logical not of the if condition!

```
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 Moves

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Move Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmove S,R</td>
<td>cmovz</td>
<td>ZF</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>cmovne S,R</td>
<td>cmovnz</td>
<td>~ZF</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>cmovs S,R</td>
<td></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>cmovns S,R</td>
<td></td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>cmovg S,R</td>
<td>cmovnle</td>
<td>~ (SF ^ OF) &amp; ~ZF</td>
<td>Greater (signed &gt;)</td>
</tr>
<tr>
<td>cmovge S,R</td>
<td>cmovnl</td>
<td>~ (SF ^ OF)</td>
<td>Greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>cmovl S,R</td>
<td>cmovnge</td>
<td>SF ^ OF</td>
<td>Less (signed &lt;)</td>
</tr>
<tr>
<td>cmovle S,R</td>
<td>cmovng</td>
<td>(SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>cmova S,R</td>
<td>cmovnbe</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>cmovae S,R</td>
<td>cmovnb</td>
<td>~CF</td>
<td>Above or equal (Unsigned &gt;=)</td>
</tr>
<tr>
<td>cmovbe S,R</td>
<td>cmovnae</td>
<td>CF</td>
<td>Below (unsigned &lt;)</td>
</tr>
<tr>
<td>cmove S,R</td>
<td>cmovna</td>
<td>CF</td>
<td>ZF</td>
</tr>
</tbody>
</table>
General Conditional Expression Translation (Using Branches)

C Code (Ternary Operator)

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

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

Goto Version

```
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:
    ```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 /* x <= y */
        result = y-x;
    return result;
}
```

### absdiff:

- `movq %rdi, %rax # x`
- `subq %rsi, %rax # result = x-y`
- `movq %rsi, %rdx # y`
- `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

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

Computations with side effects

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

- Both values get computed
- Must be side-effect free