Machine-Level Programming III: Control Flow

Topics

- Condition Codes
  - Setting
  - Testing
- Control Flow
  - If-then-else
  - Varieties of Loops
Controlling program execution

We can now generate programs that execute linear sequences of instructions

- Access registers and storage
- Perform computations

But - what about loops, if-then-else, etc.?

Need ISA support for:

- Comparing and testing data values
- Directing program control
  - Jump to some instruction that isn’t just the next sequential one
  - Do so based on some condition that has been tested
Condition Codes

Single Bit Registers

<table>
<thead>
<tr>
<th>CF</th>
<th>Carry Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>Sign Flag</td>
</tr>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag</td>
</tr>
</tbody>
</table>

Implicitly Set By Arithmetic Operations

`addl Src, Dest`

C analog: \( t = a + b \)

- CF set if carry out from most significant bit
  - Used to detect unsigned overflow
- ZF set if \( t == 0 \)
- SF set if \( t < 0 \)
- OF set if two’s complement overflow
  \( (a > 0 \land b > 0 \land t < 0) \lor (a < 0 \land b < 0 \land t \geq 0) \)

*Not Set by leal instruction*
Explicit Setting by Compare Instruction

\texttt{cmp} \quad \textit{Src2,Src1}

- \texttt{cmp} \ b,a \ like \ computing \ \texttt{a-b} \ without \ setting \ destination
- CF set if carry out from most significant bit
  - Used for unsigned comparisons
- ZF set if \texttt{a == b}
- SF set if \texttt{(a-b) < 0}
- OF set if two’s complement overflow
  \((a>0 \ \&\& \ b<0 \ \&\& (a-b)<0) \ \| \ (a<0 \ \&\& \ b>0 \ \&\& (a-b)>0)\)
Explicit Setting by Test instruction

\texttt{testl \ Src2,Src1}

- Sets condition codes based on value of \texttt{Src1 & Src2}
  - Useful to have one of the operands be a mask
- \texttt{testl b,a} like computing \texttt{a\&b} without setting destination
- ZF set when \texttt{a\&b == 0}
- SF set when \texttt{a\&b < 0}
## Reading Condition Codes

### SetX Instructions
- Set single byte based on combinations of condition codes

<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>settle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</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>
### Reading Condition Codes (Cont.)

#### SetX Instructions

- Set single byte based on combinations of condition codes
- One of 8 addressable byte registers
  - Embedded within first 4 integer registers
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

```c
int gt (int x, int y) {
    return x > y;
}
```

### Body

```Assembly
movl 12(%ebp),%eax  # eax = y
cmpl %eax,8(%ebp)   # Compare x : y
setg %al            # al = x > y
movzbl %al,%eax     # Zero rest of %eax
```

#### Note
- Inverted ordering!
## 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>
Conditional Branch Example

```c
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

```assembly
_max:
pushl %ebp
movl %esp,%ebp

movl 8(%ebp),%edx
movl 12(%ebp),%eax

cmpeq %eax,%edx
jle L9

movl %edx,%eax

L9:

movl %ebp,%esp
popl %ebp
ret
```

Set Up

Body

Finish
Conditional Branch Example (Cont.)

```c
int goto_max(int x, int y)
{
    int rval = y;
    int ok = (x <= y);
    if (ok)
        goto done;
    rval = x;

    done:
    return rval;
}
```

- C allows “goto” as means of transferring control
  - Closer to machine-level programming style
- Generally considered bad coding style

```assembly
movl 8(%ebp),%edx  # edx = x
movl 12(%ebp),%eax  # eax = y
cmpl %eax,%edx     # x : y
jle L9               # if <= goto L9
movl %edx,%eax      # eax = x

L9:                     # Done:
```

Skipped when \( x \leq y \)
“Do-While” Loop Example

C Code

```c
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```c
int fact_goto(int x)
{
  int result = 1;
  loop:
    result *= x;
    x = x-1;
    if (x > 1)
      goto loop;
  return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds
“Do-While” Loop Compilation

Goto Version

```c
int fact_goto (int x)
{
    int result = 1;
    loop:
        result *= x;
        x = x-1;
    if (x > 1)
        goto loop;
    return result;
}
```

Assembly

```assembly
_fact_goto:
    pushl %ebp              # Setup
    movl %esp,%ebp          # Setup
    movl $1,%eax            # eax = 1
    movl 8(%ebp),%edx       # edx = x

L11:
    imull %edx,%eax         # result *= x
    decl %edx               # x--
    cmpl $1,%edx            # Compare x : 1
    jg L11                  # if > goto loop
    movl %ebp,%esp          # Finish
    popl %ebp               # Finish
    ret                     # Finish
```

Registers

- `%edx` x
- `%eax` result
General “Do-While” Translation

C Code

\[
do \\
\begin{array}{c}
Body \\
\text{while (Test);} \\
\end{array}
\]

Goto Version

\[
\text{loop:} \\
\begin{array}{c}
\text{Body} \\
\text{if (Test)} \\
goto \text{loop} \\
\end{array}
\]

- **Body** can be any C statement
  - Typically compound statement:
    \[
    \{ \\
    \quad \text{Statement}_1; \\
    \quad \text{Statement}_2; \\
    \quad \ldots \\
    \quad \text{Statement}_n; \\
    \}
    \]
- **Test** is expression returning integer
  - = 0 interpreted as false
  - ≠0 interpreted as true
“While” Loop Example #1

C Code

```c
int fact_while
    (int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

First Goto Version

```c
int fact_while_goto
    (int x)
{
    int result = 1;
    loop:
    if (!(x > 1))
        goto done;
    result *= x;
    x = x-1;
    goto loop;
    done:
    return result;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails
Actual “While” Loop Translation

C Code

```c
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```

- Uses same inner loop as do-while version
- Guards loop entry with extra test

Second Goto Version

```c
int fact_while_goto2(int x)
{
    int result = 1;
    if (!(x > 1))
        goto done;
    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;
    done:
        return result;
}
```
General “While” Translation

C Code

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:
**Summarizing**

**C Control**
- if-then-else
- do-while
- while
- switch

**Assembler Control**
- jump
- Conditional jump

**Compiler**
- Must generate assembly code to implement more complex control

**Standard Techniques**
- All loops converted to do-while form
- Large switch statements use jump tables

**Conditions in CISC**
- CISC machines generally have condition code registers

**Conditions in RISC**
- Use general registers to store condition information
- Special comparison instructions
  - E.g., on Alpha:
    ```
    cmple $16,1,$1
    ```
    - Sets register $1 to 1 when Register $16 <= 1
Summary

Instruction support for control flow

- Test/Compare instructions modify condition codes
- Branch/Jump instructions can conditionally execute based on condition code
- ...and set program counter (%eip) point to some instruction elsewhere in the program

Next time

- More loop examples
- Switch statements and jump tables