Assembly II: Control Flow

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
IA-32 Processor State

Temporary data

- %eax
- %edx
- %ecx
- %ebx
- %esi
- %edi

General purpose registers

Location of runtime stack

- %esp
- %ebp

Current stack top

Current stack frame

Location of current code control point

- %eip

Instruction pointer

Status of recent tests

- CF
- ZF
- SF
- OF

Condition codes (EFLAGS)
Setting Condition Codes (1)

- Single bit registers
  - CF (Carry), SF (Sign), ZF (Zero), OF (Overflow)

- Implicitly set by arithmetic operations
  - Example: `addl Src, Dest` (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 && b>0 && t<0) || (a<0 && b<0 && t>0)

- Not set by leal, incl, or decl instruction
Setting Condition Codes (2)

- **Explicitly setting by compare instruction**
  - Example: `cmpl b, a`
  - Computes \((a - b)\) without saving the result
  - CF set if carry out from most significant bit
    - Used for unsigned comparisons
  - ZF set if \(a == b\)
  - SF set if \((a - b) < 0\)
  - OF set if two’s complement overflow
    - \((a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)\)
Explicitly setting by test instruction

- Example: `testl b, a`
- Sets condition codes based on value of `a` and `b`
  - Useful to have one of the operands be a mask
- Computes `a & b` without setting destination
- ZF set when `a & b == 0`
- SF set when `a & b < 0`
- CF and OF are cleared to 0
## 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 &gt;)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF ^ OF)</td>
<td>Greater or Equal (Signed &gt;=)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF ^ OF)</td>
<td>Less (Signed &lt;)</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 &gt;)</td>
</tr>
<tr>
<td>jae</td>
<td>~CF</td>
<td>Above or Equal (Unsigned &gt;=)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (Unsigned &lt;)</td>
</tr>
<tr>
<td>jbe</td>
<td>CF</td>
<td>ZF</td>
</tr>
</tbody>
</table>
### 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>$R_8 \leftarrow ZF$</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>$R_8 \leftarrow \sim ZF$</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>$R_8 \leftarrow SF$</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>$R_8 \leftarrow \sim SF$</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>$R_8 \leftarrow \sim(SF \land OF) \land ZF$</td>
<td>Greater (Signed $&gt;$)</td>
</tr>
<tr>
<td>setge</td>
<td>$R_8 \leftarrow \sim(SF \land OF)$</td>
<td>Greater or Equal (Signed $\geq$)</td>
</tr>
<tr>
<td>setl</td>
<td>$R_8 \leftarrow (SF \land OF)$</td>
<td>Less (Signed $&lt;$)</td>
</tr>
<tr>
<td>setle</td>
<td>$R_8 \leftarrow (SF \land OF) \lor ZF$</td>
<td>Less or Equal (Signed $\leq$)</td>
</tr>
<tr>
<td>seta</td>
<td>$R_8 \leftarrow \sim CF \land \sim ZF$</td>
<td>Above (Unsigned $&gt;$)</td>
</tr>
<tr>
<td>setae</td>
<td>$R_8 \leftarrow \sim CF$</td>
<td>Above or Equal (Unsigned $\geq$)</td>
</tr>
<tr>
<td>setb</td>
<td>$R_8 \leftarrow CF$</td>
<td>Below (Unsigned $&lt;$)</td>
</tr>
<tr>
<td>setbe</td>
<td>$R_8 \leftarrow CF \lor ZF$</td>
<td>Below or Equal (Unsigned $\leq$)</td>
</tr>
</tbody>
</table>
Reading Condition Codes (2)

- **setX instructions**
  - One of 8 addressable byte registers
    - %ah, %al, %bh, %bl, %ch, %cl, %dh, %dl
  - Does not alter remaining 3 bytes
  - Typically use *movzbl* to finish job

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

```
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
```

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>%dh</td>
<td>%dl</td>
</tr>
<tr>
<td>%ecx</td>
<td>%ch</td>
<td>%cl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bh</td>
<td>%bl</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: inverted ordering!
Conditional Branch (1)

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

```
_max:
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx
movl 12(%ebp),%eax
cmpl %eax,%edx
jle L9
movl %edx,%eax
L9:
movl %ebp,%esp
popl %ebp
ret
```

Set Up
Body
Finish
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

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  }  Skipped when x ≤ y
L9:                        # Done:
“Do-While” Loop (1)

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 (2)

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

```asm
_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--
    cmp $1,%edx # Compare x : 1
    jg L11 # if > goto loop

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

- Registers
  - %edx  x
  - %eax  result


“Do-While” Loop (3)

- General “Do-While” translation

C Code

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

Goto Version

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

- *Body* can be any C statement
  - Typically compound statement:

```
{
  Statement_1;
  Statement_2;
  ...
  Statement_n;
}
```

- *Test* is expression returning integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
“While” Loop (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
“While” Loop (2)

C Code

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

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;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test
"While" Loop (3)

- General "While" translation

**C Code**

```c
while (Test)
    Body
```

Do-While Version

```c
if (!Test)
    goto done;
do
    Body
while (Test);
done:
```

Goto Version

```c
if (!Test)
    goto done;
loop:
    Body
    if (Test)
        goto loop;
done:
```
“For” Loop (1)

```c
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

### Algorithm

- Exploit property that \( p = p_0 + 2p_1 + 4p_2 + \ldots 2^{n-1}p_{n-1} \)
- Gives: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot ((z_{n-1}^2)^2)^2 \)
  
  \[
  z_i = \begin{cases} 
  1 & \text{ when } p_i = 0 \\
  x & \text{ when } p_i = 1
  \end{cases}
  \]
- Complexity \( O(\log p) \)

#### Example

\[
3^{10} = 3^2 \cdot 3^8 = 3^2 \cdot (3^2)^2
\]
```
int result;
for (result = 1;
    p != 0;
    p = p>>1) {
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

**General Form**

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

**Body**

```
{ if (p & 0x1)
    result *= x;
    x = x*x;
}```

**Init**

`result = 1`

**Test**

`p != 0`

**Update**

`p = p >> 1`
"For" Loop (3)

For Version

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

While Version

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

Do-While Version

```c
Init;
if (!Test)
   goto done;
do {
   Body
   Update ;
} while (Test)
done:
```

Goto Version

```c
Init;
if (!Test)
   goto do;
do {
   Body
   Update ;
} while (Test)
   if (Test)
      goto loop;
done:
```
"For" Loop (4)

Goto Version

Init;
  if (!Test)
    goto done;
loop:
  Body
  Update;
  if (Test)
    goto loop;
done:

Result = 1;
if (p == 0)
goto done;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
p = p >> 1;
if (p != 0)
goto loop;
done:

Init
result = 1

Test
p != 0

Update
p = p >> 1

Body
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
“Switch” Statement (1)

- **Implementation options**
  - Series of conditionals
    - Good if few cases
    - Slow if many
  - Jump table
    - Lookup branch target
    - Avoids conditionals
    - Possible when cases are small integer constants
  - GCC
    - Picks one based on case structure
  - Bug in example code
    - No default given

```c
typedef enum {
  ADD, MULT, MINUS, DIV,
  MOD, BAD
} op_type;

char unparsesymbol (op_type op) {
  switch (op) {
    case ADD : return '+';
    case MULT: return '*';
    case MINUS: return '-';
    case DIV: return '/';
    case MOD: return '%';
    case BAD: return '?';
  }
}
```
"Switch" Statement (2)

- Jump table structure

**Switch Form**

```c
switch(op) {
    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>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>Targn-1</th>
</tr>
</thead>
</table>

**Jump Targets**

- **Targ0:** Code Block 0
- **Targ1:** Code Block 1
- **Targn-1:** Code Block n-1

**Approx. Translation**

```
target = JTab[op];
goto *target;
```
“Switch” Statement (3)

Branching Possibilities

typedef enum {
    ADD, MULT, MINUS, DIV, MOD, BAD
} op_type;

char unparse_symbol(op_type op) {
    switch (op) {
        • • •
    }
}

unparse_symbol:
    pushl %ebp                  # Setup
    movl %esp,%ebp              # Setup
    movl 8(%ebp),%eax           # eax = op
    cmpl $5,%eax                # Compare op : 5
    ja .L49                     # If > goto done
    jmp *.L57(,%eax,4)          # goto Table[op]

Enumerated Values

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
</tr>
<tr>
<td>MULT</td>
<td>1</td>
</tr>
<tr>
<td>MINUS</td>
<td>2</td>
</tr>
<tr>
<td>DIV</td>
<td>3</td>
</tr>
<tr>
<td>MOD</td>
<td>4</td>
</tr>
<tr>
<td>BAD</td>
<td>5</td>
</tr>
</tbody>
</table>

Setup:
“Switch” Statement (4)

- **Symbolic labels**
  - Labels of form `.LXX` translated into addresses by assembler

- **Table structure**
  - Each target requires 4 bytes, Base address at `.L57`

- **Jumping**
  - `jmp .L49`
    - Jump target is denoted by label `.L49`
  - `jmp *.L57(,%eax,4)`
    - Start of jump table denoted by label `.L57`
    - Register `%eax` holds `op`
    - Must scale by factor of 4 to get offset into table
    - Fetch target from effective address `.L57 + op * 4`
“Switch” Statement (5)

Table Contents

```
.section .rodata
.align 4
.L57:
.long .L51 #Op = 0
.long .L52 #Op = 1
.long .L53 #Op = 2
.long .L54 #Op = 3
.long .L55 #Op = 4
.long .L56 #Op = 5
```

Enumerated Values

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0</td>
</tr>
<tr>
<td>MULT</td>
<td>1</td>
</tr>
<tr>
<td>MINUS</td>
<td>2</td>
</tr>
<tr>
<td>DIV</td>
<td>3</td>
</tr>
<tr>
<td>MOD</td>
<td>4</td>
</tr>
<tr>
<td>BAD</td>
<td>5</td>
</tr>
</tbody>
</table>

Targets & Completion

```
.L51:
  movl $43,%eax # '+'
  jmp .L49

.L52:
  movl $42,%eax # '*'
  jmp .L49

.L53:
  movl $45,%eax # '-'
  jmp .L49

.L54:
  movl $47,%eax # '/'
  jmp .L49

.L55:
  movl $37,%eax # '%'
  jmp .L49

.L56:
  movl $63,%eax # '?'
  # Fall Through to .L49
```
“Switch” Statement (6)

- Switch statement completion

```assembly
.L49:
    movl %ebp,%esp     # Done:
    popl %ebp         # Finish
    ret               # Finish
```

- What value returned when op is invalid?
  - Register %eax set to op at beginning of procedure
  - This becomes the return value

- Advantage of jump table
  - Can do k-way branch in O(1) operations
“Switch” Statement (7)

- **Sparse switch example**
  - Not practical to use jump table
    - Would require 1000 entries
  - Obvious translation into if-then-else would have max. of 9 tests

```c
/* Return x/111 if x is multiple && <= 999.
   Return -1 otherwise */
int div111(int x) {
    switch(x) {
    case 0: return 0;
    case 111: return 1;
    case 222: return 2;
    case 333: return 3;
    case 444: return 4;
    case 555: return 5;
    case 666: return 6;
    case 777: return 7;
    case 888: return 8;
    case 999: return 9;
    default: return -1;
    }
}
```
"Switch" Statement (8)

- Compares x to possible case values
- Jumps different places depending on outcomes

```
movl 8(%ebp),%eax # get x
cmpl $444,%eax    # x:444
  je L8
  jg L16
  cmp $111,%eax    # x:111
  je L5
  jg L17
  testl %eax,%eax  # x:0
  je L4
  jmp L14

L5:
  movl $1,%eax
  jmp L19

L6:
  movl $2,%eax
  jmp L19

L7:
  movl $3,%eax
  jmp L19

L8:
  movl $4,%eax
  jmp L19

...```

...
“Switch” Statement (9)

- Sparse switch code structure
  - Organizes cases as binary tree
  - Logarithmic performance

![Binary Tree Diagram]

- Parse switch code structure
  - Organizes cases as binary tree
  - Logarithmic performance
Summary

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

- **Assembler control**
  - Jump
  - Conditional jump
  - Indirect 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: $\text{cmple} \quad 16, 1, 1$
    - Sets register $1$ to $1$ when $16 \leq 1$