Machine-Level Programming II  
Control Flow  
Sept. 14, 2000

Topics
- Condition Codes
  - Setting
  - Testing
- Control Flow
  - If-then-else
  - Varieties of Loops
  - Switch Statements

Condition Codes

Single Bit Registers
- CF  Carry Flag
- ZF  Zero Flag
- SF  Sign Flag
- OF  Overflow Flag

Implicit Setting By Arithmetic Operations

\[
\text{addl } \text{Src, Dest} \\
\text{C analog: } t = a+b
\]
- 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
  \[
  (a>0 && b>0 && t<0) || (a<0 && b<0 && t>0)
  \]
Not Set by leal instruction

Setting Condition Codes (cont.)

Explicit Setting by Compare Instruction

\[
\text{cmp} l  \text{Src2, Src1}
\]
- 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
  \[
  (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
  \]

Explicit Setting by Test instruction

\[
\text{testl } \text{Src2, Src1}
\]
- Sets condition codes based on value of \( \text{Src1} & \text{Src2} \)
  - Useful to have one of the operands be a mask
- \text{testl b, a} like computing \( a\&b \) without setting destination
- ZF set when \( a\&b == 0 \)
- SF set when \( a\&b < 0 \)

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>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>
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 andl 0xFF,%eax to finish job

Jumping

jX Instructions
- Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Conditional Branch Example

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

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

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

Conditional Branch Example (Cont.)

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

L9:
```

```
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:
```

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

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
```

General “Do-While” Translation

C Code
```c
do
    Body
while (Test);
```

Goto Version
```c
loop: 
    Body
    if (Test)
        goto loop
```

- **Body** can be any C statement
  - Typically compound statement:
    ```c
    {
        Statement_1;
        Statement_2;
        ...
        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;
    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

General “While” Translation

C Code

```c
while (Test)
{
    Body
}
```

Do-While Version

```c
if (!Test)
    goto done;
```

Goto Version

```c
if (!Test)
    goto done;
```

“While” Loop Example #2

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

Algorithm

- Exploit property that \( p = p_n + 2p_{n-1} + 4p_{n-2} + \ldots + 2^{n-1}p_1 \)
- Gives: \( x^n = z_0 \cdot (z_2 \cdot \ldots \cdot (z_{n-2} \cdot \ldots \cdot z_2 \cdot z_1)^2)^2 \)
- Complexity \( O(\log p) \)

<table>
<thead>
<tr>
<th>result</th>
<th>x</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>6561</td>
<td>1</td>
</tr>
<tr>
<td>531441</td>
<td>43046721</td>
<td>0</td>
</tr>
</tbody>
</table>

Example

\( 3^{10} = 3^2 \cdot 3^8 = 3^2 \cdot ((3^2)^2)^2 \)
**“While” → “Do-While” → “Goto”**

```
int result = 1;
while (p) {
    if (p & 0x1)
        result *= x;
    x = x*x;
p = p>>1;
}
```

```
int result = 1;
if (!p)
goto done;
do {
    if (p & 0x1)
        result *= x;
    x = x*x;
p = p>>1;
    if (p)
goto loop;
done:
```

• Also converted conditional update into test and branch around update code

**Example #2 Compilation**

**Goto Version**

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

**Registers**

- %ecx x
- %edx p
- %eax result

**“For” Loop Example**

```
int result;
for (result = 1;
p != 0;
p = p >> 1) {
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

**General Form**

- **Init**: `result = 1`
- **Test**: `p != 0`
- **Update**: `p = p >> 1`
- **Body**

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

**“For” → “While”**

**Do-While Version**

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

**“For” Loop Example**

```
int result;
for (result = 1;
p != 0;
p = p >> 1) {
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```

**“For Version**

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

**While Version**

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

**Do-While Version**

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

**Goto Version**

```
Init:
if (!Test)
goto done;
done:
```
**“For” Loop Compilation**

**Goto Version**

```
Init;
if (!Test)
goto done;
loop: Body
Test
if (Test) goto loop;
Update
p = p >> 1
```

**Body**

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

```
if (!Test) goto done;
```

```
Init;
```

```
if (p != 0) goto done;
```

```
do:
```

```
result = 1;
```

```
x = x*x;
```

```
if (p != 0) goto loop;
```

**Jump Table Structure**

```
switch(op) {
case 0: Block 0
case 1: Block 1
... case n-1: Block n-1
}
```

**Jump Table**

```
jtab:
Targ0:  Code Block 0
Targ1:  Code Block 1
Targ2:  Code Block 2
```

**Jump Targets**

```
switch(op) {
case 0: goto Targ 0
case 1: goto Targ 1
... case n-1: goto Targ n-1
}
```

**Switch Statement Example**

```
void unparse_symbol(op_type op) {
    switch (op) {
        case ADD : return '+';
        case MULT : return '*' ;
        case MINUS: return '-' ;
        case DIV : return '/' ;
        case MOD : return '%' ;
        case BAD : return '?' ;
    }
}
```

**Switch Statements**

**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

**Enumerated Values**

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

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

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

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

**Setup**

```
unparse_symbol:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    cmpl $5,%eax
    ja .L49
    jmp *.L57(%eax,4)
    .L49:
    ret
```

```
switch(op) {
    case 0: goto Targ 0
    case 1: goto Targ 1
    ... case n-1: goto Targ n-1
}
```

```
setup:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    cmpl $5,%eax
    ja .L49
    jmp *.L57(%eax,4)
    .L49:
    ret
```

```
typedef enum
{ADD, MULT, MINUS, DIV, MOD, BAD}
op_type;
```
Assembly Setup Explanation

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`
- 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 Completion

Puzzle
- What value returned when `op` is invalid?

Answer
- Register `%eax` set to `op` at beginning of procedure
- This becomes the returned value

Advantage of Jump Table
- Can do `k`-way branch in `O(1)` operations

Jump Table

Table Contents

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.long</td>
<td>.long</td>
<td>.long</td>
<td>.long</td>
<td>.long</td>
<td>.long</td>
<td>.long</td>
</tr>
<tr>
<td>.long</td>
<td>.long</td>
<td>.long</td>
<td>.long</td>
<td>.long</td>
<td>.long</td>
<td>.long</td>
</tr>
</tbody>
</table>

Enumerated Values

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

Object Code

Setup
- Label `.L49` becomes address 0x804875c
- Label `.L57` becomes address 0x8048bc0

08048718 <unparse_symbol>:
- `pushl %ebp`
- `movl %esp,%ebp`
- `movl 0x8(%ebp),%eax`
- `cmp $0x5,%eax`
- `ja 804875c <unparse_symbol+0x44>`
- `jmp *0x8048bc0(,%eax,4)`
### Object Code (cont.)

#### Jump Table
- Doesn't show up in disassembled code
- Can inspect using GDB
  
  \[
  \text{gdb code-examples} \\
  \text{(gdb) x/6wx 0x8048bc0} \\
  \text{– Examine § hexadecimal format “words” (4-bytes each)} \\
  \text{– Use command “help x” to get format documentation}
  \]

### Extracting Jump Table from Binary

#### Jump Table Stored in Read Only Data Segment (.rodata)
- Various fixed values needed by your code

**Can examine with objdump**

\[
\text{objdump code-examples -s --section=.rodata} \\
\text{• Show everything in indicated segment.}
\]

**Hard to read**
- Jump table entries shown with reversed byte ordering

### Disassembled Targets
- No-operations \((\text{movl } %esi, %esi)\) inserted to align target addresses

#### Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Entry</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048730</td>
<td>(\text{movl} 0x2b)</td>
</tr>
<tr>
<td>0x08048735</td>
<td>(\text{jmp} 0x804875c)</td>
</tr>
<tr>
<td>0x08048737</td>
<td>(\text{movl} 0x2a)</td>
</tr>
<tr>
<td>0x0804873c</td>
<td>(\text{jmp} 0x804875c)</td>
</tr>
<tr>
<td>0x0804873e</td>
<td>(\text{movl} %esi, %esi)</td>
</tr>
<tr>
<td>0x08048740</td>
<td>(\text{jmp} 0x804875c)</td>
</tr>
<tr>
<td>0x08048745</td>
<td>(\text{movl} 0x2d)</td>
</tr>
<tr>
<td>0x08048747</td>
<td>(\text{jmp} 0x804875c)</td>
</tr>
<tr>
<td>0x0804874c</td>
<td>(\text{movl} %esi, %esi)</td>
</tr>
<tr>
<td>0x0804874e</td>
<td>(\text{movl} 0x2f)</td>
</tr>
<tr>
<td>0x08048750</td>
<td>(\text{jmp} 0x804875c)</td>
</tr>
<tr>
<td>0x08048755</td>
<td>(\text{movl} 0x3f)</td>
</tr>
<tr>
<td>0x08048757</td>
<td>(\text{movl} 0x3f)</td>
</tr>
</tbody>
</table>
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
  ```