Machine-Level Programming II
Control Flow
Feb. 3, 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
  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 && b>0 && t<0) || (a<0 && b<0 && t>0)
  Not Set by leal instruction

Setting Condition Codes (cont.)

Explicit Setting by Compare Instruction
  cmpl Src2, Src1
    • cmpl 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
    • 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
  testl Src2, Src1
    • Sets condition codes based on value of Src1 & Src2
      - Useful to have one of the operands be a mask
    • 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>setsns</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) &amp; ZF</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 `andl 0xFF, %eax` to finish job

```plaintext
%eax %ebx %ecx %edx
%al %ah %dl %dh
%cl %ch %bl %bh
```

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

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>js</td>
<td>~(SF^OF)&amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jns</td>
<td>~SF^OF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>EF</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

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

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

Conditional Branch Example (Cont.)

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

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

```
if (ok)
    goto done;
else
    rval = x;
```

```
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:
```

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;
}
```

Do-While“ Loop Compilation

Registers

%edx x
%eax 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
```

General “Do-While” Translation

C Code

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

Goto Version

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

“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;
}
```

- Body can be any C statement
  - Typically compound statement:
    ```c
    {
    Statement;
    Statement;
    Statement;
    }
    ```

- Test is expression returning integer
  - 0 interpreted as false
  - ≠0 interpreted as true

- Use backward branch to continue looping
- Only take branch when "while" condition holds

- Do-While“ Loop Example
- C Code
- Goto Version
- Do-While“ Loop Compilation
- General “Do-While” Translation
- “While“ Loop Example #1
- First Goto Version

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails
C Code
int fact_while(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}

Second Goto Version
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;
}

Actual "While" Loop Translation
- Uses same inner loop as do-while version
- Guards loop entry with extra test

"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;
}

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:

ipwr Computation
int ipwr(int x, unsigned p)
{
    int result = 1;
    while (p) {
        if (p & 0x1)
            result *= x;
        x = x*x;
        p = p>>1;
    }
    return result;
}

<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>
"While" → "Do-While" → "Goto"

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

```c
int result = 1;
if (!p) goto done;
```

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

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

```
Example #2 Compilation
```

```assembly
pushl %ebp
movl %esp,%ebp
movl $1,%eax
movl 8(%ebp),%ecx
movl 12(%ebp),%edx
testl %edx,%edx
je L36
L37:
testb $1,%dl
je L38
imull %ecx,%eax
L38:
imull %ecx,%ecx
shrl $1,%edx
jne L37
```

```
Goto Version
```

```assembly
int result = 1;
if (!p)
goto done;
```

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

```
For" Loop Example
```

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

```
"For" → "While"
```

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

```
For Version
```

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

```
While Version
```

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

```
Do-While Version
```

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

```
Goto Version
```

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

```
While Version
```

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

```
Goto Version
```

```c
do {
    Body
    Update ;
} while (Test)
done:
```

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

```
Registers
%ecx x
%edx p
%eax result
```

```
Example #2 Compilation
```

```
Goto Version
```

```
pushl %ebp
movl %esp,%ebp
movl $1,%eax
movl 8(%ebp),%ecx
movl 12(%ebp),%edx
testl %edx,%edx
je L36
L37:
testb $1,%dl
je L38
imull %ecx,%eax
L38:
imull %ecx,%ecx
shrl $1,%edx
jne L37
```

```
Goto Version
```

```
pushl %ebp
movl %esp,%ebp
movl $1,%eax
movl 8(%ebp),%ecx
movl 12(%ebp),%edx
testl %edx,%edx
je L36
L37:
testb $1,%dl
je L38
imull %ecx,%eax
L38:
imull %ecx,%ecx
shrl $1,%edx
jne L37
```

```
Goto Version
```

```
pushl %ebp
movl %esp,%ebp
movl $1,%eax
movl 8(%ebp),%ecx
movl 12(%ebp),%edx
testl %edx,%edx
je L36
L37:
testb $1,%dl
je L38
imull %ecx,%eax
L38:
imull %ecx,%ecx
shrl $1,%edx
jne L37
```

```
Goto Version
```

```
pushl %ebp
movl %esp,%ebp
movl $1,%eax
movl 8(%ebp),%ecx
movl 12(%ebp),%edx
testl %edx,%edx
je L36
L37:
testb $1,%dl
je L38
imull %ecx,%eax
L38:
imull %ecx,%ecx
shrl $1,%edx
jne L37
```

```
Goto Version
```

```
pushl %ebp
movl %esp,%ebp
movl $1,%eax
movl 8(%ebp),%ecx
movl 12(%ebp),%edx
testl %edx,%edx
je L36
L37:
testb $1,%dl
je L38
imull %ecx,%eax
L38:
imull %ecx,%ecx
shrl $1,%edx
jne L37
```
"For" Loop Compilation

Goto Version

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

Test
  p != 0
Update
  p = p >> 1
Body
  { if (p & 0x1)
    result *= x;
    x = x*x;
  }

Goto Version

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

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

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

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 '?';
  }
}

Jump Table Structure

Switch Form

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

Jump Table

jtab:
  Target0:
  Code Block 0
  Target1:
  Code Block 1
  Target2:
  Code Block 2
  Targn-1:
  Code Block n-1

Jump Targets

Approx. Translation

target = JTab[op];
goto *target;

Switch Statement Example

Branching Possibilities

typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
  op_type;
char unparse_symbol(op_type op)
{
  switch (op) {
  }
}

Enumerated Values

ADD 0
MULT 1
MINUS 2
DIV 3
MOD 4
BAD 5

pushl %ebp
movl $%ebp,%esp
movl 8(%esp),%eax
cmp 1 $5,%eax
ja .L64
jmp .L72 (%eax,4)
# Setup
# Set up
# eax = op
# Compare op : 5
# If > goto done
# goto Table[op]
Assembly Setup Explanation

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

Table Structure
- Each target requires 4 bytes
- Base address at .L72

Jumping
- jmp .L64
  - Jump target is denoted by label .L64
- jmp *.L72(,%eax,4)
  - Start of jump table denoted by label .L72
  - Register %eax holds op
  - Must scale by factor of 4 to get offset into table
  - Fetch target from effective Address .L72 + op*4

Jump Table

Table Contents
- .L66:
  - movl $43,%eax # '+'
  - jmp .L64
- .L67:
  - movl $42,%eax # '*'
  - jmp .L64
- .L68:
  - movl $45,%eax # '-'
  - jmp .L64
- .L69:
  - movl $47,%eax # '/'
  - jmp .L64
- .L70:
  - movl $37,%eax # '%'
  - jmp .L64
- .L71:
  - movl $63,%eax # '?'
  - # Fall Through to .L64

Enumerated Values
- ADD 0
- MULT 1
- MINUS 2
- DIV 3
- MOD 4
- RAD 5

Switch Statement Completion

Setup
- Label .L64 becomes address 0x80487b5
- Label .L72 becomes address 0x8048770

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

Object Code

Setup
- Label .L64 becomes address 0x80487b5
- Label .L72 becomes address 0x8048770

```
804875d: 89 e5      movl %ebp,%ebp
804875f: 8b 45 08   movl 0x8(%ebp),%eax
8048762: 83 e8 05   cmpl $0x5,%eax
8048765: 77 4e 4e   ja 80487b5
              <unparse_symbol+0x59>
8048767: ff 24 85 70 87 jmp *0x8048770(,%eax,4)
```
Object Code (cont.)

Jump Table
- Disassembler tries to interpret byte sequence as instructions
- Very strange results!

Decoding Jump Table

Known
- Starts at 0x8048770
- 4 bytes / entry
- Little Endian byte ordering

Alternate Decoding Technique
Use GDB

gdb code-examples
(gdb) x/6xw 0x8048770
  - Examine & hexadecimal format "words" (4-bytes each)
  - Use command "help x" to get format documentation

Disassembled Targets
- No-operations (nop) inserted to align target addresses
Matching Disassembled Targets

Address Entry
8048770: 08048788
8048774: 08048790
8048778: 08048798
804877c: 080487a0
8048780: 080487a8
8048784: 080487b0
8048788: b8 2b 00 00 00 movl
804878d: eb 26          jmp
804878f: 90             nop
8048790: b8 2a 00 00 00 movl
8048795: eb 1e          jmp
8048797: 90             nop
8048798: b8 2d 00 00 00 movl
804879d: eb 16          jmp
804879f: 90             nop
80487a0: b8 2f 00 00 00 movl
80487a5: eb 0e          jmp
80487a7: 90             nop
80487a8: b8 25 00 00 00 movl
80487ad: eb 06          jmp
80487af: 90             nop
80487b0: b8 3f 00 00 00 movl

Summary

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: cmpl $16,1,$1 - Sets register $1 to 1 when Register $16 < 1