15-213
“The course that gives CMU its Zip!”

Machine-Level Programming II
Control Flow
Sept. 13, 2001

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
    \( (a>0 \land b>0 \land t<0) \lor (a<0 \land b<0 \land t>=0) \)
  Not Set by leal instruction

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
    \( (a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b<0 \land (a-b)>0) \)

Setting Condition Codes (cont.)

Explicit Setting by Test Instruction (cont.)
- testl Src2, Src1
  - Sets condition codes based on value of \( Src1 \land Src2 \)
    - 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>setsn</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>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>

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>Less or Equal (Signed)</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>

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

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

Conditional Branch Example

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

Body
```
 pushed %ebp
 movl %esp, %ebp
 movl 8(%ebp), %eax # Compare x : eax ←
 movl 12(%ebp),%eax # eax = y
 cmp %eax, %edx
 jle L9 # al = x > y
 movl %edx, %eax # Zero rest of %eax
```

Note inverted ordering!

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

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

“Do-While” Loop Example

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

```c
Goto Version
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

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

```
_registers
%eax x
%edx x
%eax result
```

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

General “Do-While” Translation

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

```
_Goto Version
loop:
    Body
if (Test) goto loop;
```

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

- Test is expression returning integer
  = 0 interpreted as false
  ≠0 interpreted as true
“While” Loop Example #1

First Goto Version

```c
int fact_while_goto(int x)
{
    int result = 1;
    loop:
        if (!x) 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

Second Goto Version

```c
int fact_while_goto2(int x)
{
    int result = 1;
    loop:
        if (!x) goto done;
        result *= x;
        x = x-1;
    goto loop;
    done:
    return result;
}
```

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

General “While” Translation

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

Do-While Version

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

Goto Version

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

“For” Loop Example

```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: $x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot ((z_{n-1}^2)^2)^{2 \ldots n-1}$
  - $z_i = 1$ when $p_i = 0$
  - $z_i = x$ when $p_i = 1$
- Complexity $O(\log p)$

Example

$3^{10} = 2^6 \cdot 3^8$
$= 2^6 \cdot ((3^2)^2)^2$
ipwr Computation

```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;
}
```

<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>
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 '?';
  }
}

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

Jump Table

Jump Targets

Approx. Translation

target = JTab[op];
goto *target;

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
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
Jump Table

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>

Jump 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
```

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 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

Object Code

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

```
08048718 <unparse_symbol>:
  8048718: 55 pushl %ebp
  8048719: 89 e5 movl %esp,%ebp
  804871b: 8b 45 08 movl 0x8(%ebp),%eax
  804871e: 83 f8 05 cmpl $0x5,%eax
  8048721: 77 39 ja 804875c <unparse_symbol+0x44>
  8048723: ff 24 85 c0 8b jmp *0x8048bc0,(%eax,4)
```

Object Code (cont.)

Jump Table
- Doesn't show up in disassembled code
- Can inspect using GDB

```
gdb code-examples
(gdb) x/6xw 0x8048bc0
   - Examine 6 hexadecimal format "words" (4-bytes each)
   - Use command "help x" to get format documentation

0x8048bc0 _fini+32>:
  0x8048730 0x8048737 0x8048740 0x8048747 0x8048750 0x8048757
```
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

    objdump code-examples -s --section=.rodata
• Show everything in indicated segment.

Hard to read
• Jump table entries shown with reversed byte ordering

Contents of section .rodata:
8048b0c 30870408 37870408 40870408 47870408 0...7...@...G...
8048bdc 50870408 57870408 46616374 28256429 P...W...Fact(%d)
8048bee 203d2025 6c640a00 43686172 203d2025 = %ld..Char = % ...

• E.g., 30870408 really means 0x08048730

Disassembled Targets

Matching Disassembled Targets

Entry
0x08048730
0x08048737
0x08048740
0x08048747
0x08048750
0x08048757

8048730:b8 2b 00 00 00 movl $0x2b,%eax
8048735:eb 25 jmp 804875c <unparse_symbol+0x44>
8048737:b8 2a 00 00 00 movl $0xa2,%eax
804873c:eb 1e jmp 804875c <unparse_symbol+0x44>
804873e:89 f6 movl %esi,%esi
8048740:b8 2d 00 00 00 movl $0x2d,%eax
8048745:eb 15 jmp 804875c <unparse_symbol+0x44>
8048747:b8 2f 00 00 00 movl $0x2f,%eax
804874c:eb 0e jmp 804875c <unparse_symbol+0x44>
804874e:89 f6 movl %esi,%esi
8048750:b8 25 00 00 00 movl $0x25,%eax
8048755:eb 05 jmp 804875c <unparse_symbol+0x44>
8048757:b8 3f 00 00 00 movl $0x3f,%eax

• movl %esi,%esi does nothing
• Inserted to align instructions for better cache performance

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

Assembler Control
• jump
• Conditional jump

Compiler
• Must generate assembly code to implement more complex control

Summarizing

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

...