Giving credit where credit is due

Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
I have modified them and added new slides.

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

Condition Codes

Single Bit Registers

- CF Carry Flag
- SF Sign Flag
- ZF Zero Flag
- OF Overflow Flag

Implicitly Set By Arithmetic Operations

- \(\text{add} \ Src_{Dest}\)
  - 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
    \((a>0 \&\& b>0 \&\& (a-b)<0) \mid (a<0 \&\& b<0 \&\& (a-b)>0)\)

Not Set by \text{lea} \ instruction

Setting Condition Codes (cont.)

Explicit Setting by Compare Instruction

- \(\text{cmp}l \ Src_{2},Src_{1}\)
  - \(\text{cmp}l \ b,a\) 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
    \((a>0 \&\& b<0 \&\& (a-b)>0) \mid (a<0 \&\& b>0 \&\& (a-b)<0)\)

Setting Condition Codes (cont.)

Explicit Setting by Test instruction

- \(\text{test}l \ Src_{2},Src_{1}\)
  - Sets condition codes based on value of \(Src_{1} \&\& Src_{2}\)
    - Useful to have one of the operands be a mask
  - \(\text{test}l \ b,a\) 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>setn</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setge</td>
<td>~SF&amp;OF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>~SF&amp;OF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>setbe</td>
<td>~SF&amp;OF</td>
<td>Below (unsigned)</td>
</tr>
<tr>
<td>setbh</td>
<td>CF</td>
<td>Above (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 movzb1 to finish job

<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>setn</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&amp;OF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>SF&amp;OF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>~SF&amp;OF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>~SF&amp;OF</td>
<td>Less or Equal (Signed)</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>jn</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&amp;OF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>SF&amp;OF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>~SF&amp;OF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>~SF&amp;OF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Conditional Branch Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl 12(%ebp),%eax</td>
<td># edx = x</td>
</tr>
<tr>
<td>movl 8(%ebp),%edx</td>
<td># ebx = y</td>
</tr>
<tr>
<td>jle L9</td>
<td># if &lt;= goto L9</td>
</tr>
<tr>
<td>movl %edx, %eax</td>
<td># max = x</td>
</tr>
<tr>
<td>L9</td>
<td># Done:</td>
</tr>
</tbody>
</table>

Conditional Branch Example (Cont.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int goto_max(int x, int y) {</td>
<td></td>
</tr>
<tr>
<td>int rval = y;</td>
<td></td>
</tr>
<tr>
<td>int ok = (x &lt;= y);</td>
<td></td>
</tr>
<tr>
<td>if (ok)</td>
<td>goto done;</td>
</tr>
<tr>
<td>rval = x;</td>
<td>done:</td>
</tr>
<tr>
<td>return rval;</td>
<td>}</td>
</tr>
</tbody>
</table>

“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;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    return result;
}
```

Assembly

```
L11:
    movl %edx,%eax  # x
    dec %edx        # x--
    cmp $1,%edx    # Compare x : 1
    jg L11         # if > goto loop
    movl %eax,%edx # edx = x
    movl %esp,%ebp # Setup
decl %edx        # x - -
    imull %edx,%eax # result *= x
    movl 8(%ebp),%edx # edx = x
    movl $1,%eax   # eax = 1
    jg L11         # if > goto loop
    ret            # Finish
```

Registers

- %edx x
- %eax result

“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

```
int fact_while_goto
(int x)
{
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    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;
}
```

General “While” Translation

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

Do-While Version

```
if (!Test)
    done;
else
    Body
    while (Test);
```

Goto Version

```
if (!Test)
    goto loop;
else
    Body
    if (Test)
        goto loop;
    done;
```

“For” Loop Example

```
/* Compute x raised to nonnegative power p */
int ipow_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;
}
```

```
3^8 = 2^{2^3} = 2^6 = 64
```

Algorithm

- Exploit property that $p = p_0 + 2p_1 + 4p_2 + \ldots + 2^{n-1}p_{n-1}$
- Given: $v = \ldots z_t \cdot z_{t-1} \cdot \ldots \cdot z_1 \cdot z_0$
- $z_i = 1$ when $p_i = 1$
- Complexity $O(\log p)$
ipwr Computation

```c
/* Compute x raised to nonnegative power p */
int ipwr(int x, unsigned p) {
    int result;
    for (result = 1; p != 0; p = p >> 1) {
        if (p & 0x1)
            result *= 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>2</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>31441</td>
<td>31</td>
<td>0</td>
</tr>
</tbody>
</table>
```

“For” Loop Example

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

“Switch” Statements

- 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

Jump Table Structure

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

“Switch” Form

```
switch(op) {
    case val_0:
        Block 0
        • • •
        case val_n-1:
        Block n-1
}
```

Jump Targets

```
Jump Targets
Targ0: Code Block 0
Targ1: Code Block 1
Targ2: Code Block 2
Targn-1: Code Block n-1
```
Switch Statement Example

Branching Possibilities

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

char unparse_symbol[op_type op]
{
    switch (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 .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

<table>
<thead>
<tr>
<th>Targets &amp; Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L51: movi $43, eax # 4'</td>
</tr>
<tr>
<td>jmp .L49</td>
</tr>
<tr>
<td>.L52: movi $42, eax # 3'</td>
</tr>
<tr>
<td>jmp .L49</td>
</tr>
<tr>
<td>.L53: movi $45, eax # 5'</td>
</tr>
<tr>
<td>jmp .L49</td>
</tr>
<tr>
<td>.L54: movi $47, eax # 7'</td>
</tr>
<tr>
<td>jmp .L49</td>
</tr>
<tr>
<td>.L55: movi $37, eax # 5'</td>
</tr>
<tr>
<td>jmp .L49</td>
</tr>
<tr>
<td>.L56: movi $36, eax # 6'</td>
</tr>
<tr>
<td># Fall Through to .L49</td>
</tr>
</tbody>
</table>

Switch Statement Completion

| .L49:   movi 0, esp # Finish |
| popl  esp # Finish |

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 n-way branch in O(1) operations

Object Code

Setup
- Label .L49 becomes address 0x804875c
- Label .L57 becomes address 0x8048b00

Object Code (cont.)

Jump Table
- Doesn't show up in disassembled code
- Can inspect using GDB
  - examine $x/word 0x8048bc0
  - Use command "help x" to get format documentation

0xb8<__finish+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:
0x08048730 0x08048740 0x08048742
0x08048750 0x08048752 0x08048754
0x08048757 0x08048758 0x0804875a
0x0804875c 0x0804875d

E.g., 0x08048700 really means 0x08048730

Matching Disassembled Targets

Disassembled Targets

Sparse Switch Code

Sparse Switch Code Structure

Disassembled Targets

/* Return x/111 if x is multiple
** 4 <= 999. -1 otherwise */
sam diviii(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;  
    }  
}

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

Sparse Switch Code

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

Sparse Switch Code Structure

- Organizes cases as binary tree
- Logarithmic performance
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:
    - cmp $16, 1, $1
      - Sets register $1 to 1 when Register $16 == 1