Control Flow

Condition Codes

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

- Implicitly Set 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
  - `cmp1 Src2, Src1`
    - `cmp1 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)`

Setting Condition Codes (cont.)

- 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>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>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 `movzbl` to finish job

```
# Initial setup
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
```

Note inverted ordering!

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>

Conditional Branch Example

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

```
_long:
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
    # eax = y
movl 12(%ebp),%edx
    # edx = x
cmp %eax,%edx
    # Compare x : y
jle L9
mov %edx,%eax
    # Zero rest of %eax
```

```
L9:
popl %ebp
ret
```

Body

Note inverted ordering!
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:

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

C Code

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

```
Goto Version

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

```
Goto Version

int fact_goto(int x) {
    int result = 1;
    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;
    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

```
Goto Version

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

```
Goto Version

int fact_goto(int x) {
    int result = 1;
    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;
    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
“While” Loop Example #1

C Code

```
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;
    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

Actual “While” Loop Translation

<table>
<thead>
<tr>
<th>C Code</th>
<th>Second Goto Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fact_while_goto2 (int x) {</td>
<td>int fact_while_goto2 (int x)</td>
</tr>
<tr>
<td>int result = 1;</td>
<td>int result = 1;</td>
</tr>
<tr>
<td>while (x &gt; 1) {</td>
<td>if (!(x &gt; 1))</td>
</tr>
<tr>
<td>result *= x;</td>
<td>goto done;</td>
</tr>
<tr>
<td>x = x - 1;</td>
<td>result *= x;</td>
</tr>
<tr>
<td>x = x - 1;</td>
<td>x = x - 1;</td>
</tr>
<tr>
<td>goto loop;</td>
<td>if (x &gt; 1)</td>
</tr>
<tr>
<td>done:</td>
<td>goto loop;</td>
</tr>
<tr>
<td></td>
<td>done:</td>
</tr>
<tr>
<td></td>
<td>return result;</td>
</tr>
</tbody>
</table>

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

General “While” Translation

```
c 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:
```

- Algorithm
  - Exploit property that \( p = p_0 + 2p_1 + 4p_2 + ... + 2^{n-1}p_{n-1} \)
  - Gives: \( x^p = z_0 + z_1 \cdot (z_2 \cdot ... \cdot (z_{n-1} \cdot z_n) \cdot ... \cdot z_n) \cdot ... \cdot z_n \)
    - \( z_i = 1 \) when \( p_i = 0 \)
    - \( z_i = x \) when \( p_i = 1 \)
  - Complexity \( O(\log p) \)

“For” Loop Example

```
/* 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;
}
```

Example

\[
3^{10} = 3 \cdot (3^2)^3 = 3^7 \cdot (3^2)^2
\]
/* 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

<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>9</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>29</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>

### Diagrams

**“For” Loop Example**

- **General Form**
  - **Init**: result = 1
  - **Test**: p != 0
  - **Update**: p = p >> 1
  - **Body**: if (p & 0x1) result *= x; x = x*x;

**“For” Loop Compilation**

- **Goto Version**
  - **Init**: result = 1
  - **Test**: p = p >> 1
  - **Body**: if (p & 0x1) result *= x; x = x*x;

- **Do-While Version**
  - **Init**: if (!Test) goto done;
  - **Body**: 
    - if (!Test) goto done;
    - result = 1;
    - loop:
      - if (p & 0x1) result *= x;
      - x = x*x;
      - p = p >> 1;
      - if (p != 0) goto loop;
    - done:

- **While Version**
  - **Init**: if (!Test) goto done;
  - **Body**: 
    - if (!Test) goto done;
    - result = 1;
    - 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

<table>
<thead>
<tr>
<th>Switch Form</th>
<th>Jump Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch(op) {</td>
<td></td>
</tr>
</tbody>
</table>
| case val_0:
| Block 0 |
| case val_1:
| Block 1 |
| case val_n-1:
| Block n-1 |
| } |
| jtab: |
| Targ0: |
| Targ1: |
| Targ2: |
| Targn-1: |

Approx. Translation

```c
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

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 '?';
    }
}

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 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**

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

### Targets & Completion

- `movl $43,%eax # '+' jmp .L49`
- `movl $42,%eax # '*' jmp .L49`
- `movl $45,%eax # '-' jmp .L49`
- `movl $47,%eax # '/' jmp .L49`
- `movl $37,%eax # '%' jmp .L49`
- `movl $63,%eax # '?' # Fall Through to .L49`

### Switch Statement Completion

```
.L49: # Done:  
    movl %ebp,%esp # Finish  
    popl %ebp # Finish  
    ret # 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 `k`-way branch in $O(1)$ operations

### Object Code

**Setup**

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

```c
do something
```

**Object Code (cont.)**

**Jump Table**

- Doesn’t show up in disassembled code
- Can inspect using GDB

```
gdb code-examples  
(gdb) x/6xw 0x8048bc0  
```

**Examine** hexadecimal format “words” (4-bytes each)

- Use command `help x` to get format documentation

```
0x8048bc0 <_fini+32>:  
0x08048730  
0x08048737  
0x08048740  
0x08048747  
0x08048750  
0x08048757
```
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:
8048bc0 30870408 30780408 40870408 40780408 0...7...8...G...
8048bd0 30870408 30780408 46616374 28256429 P...W...Fact(%d)
8048be0 20320205 6c640a00 43686172 20320205 = %ld..Char = %

E.g., 30870408 really means 0x08048730

Disassembled Targets

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048730</td>
<td>movl $0x2b,%eax</td>
<td>08048730</td>
</tr>
<tr>
<td>0x8048735</td>
<td>jmp 0804875c &lt;unparse_symbol+0x44&gt;</td>
<td>08048735</td>
</tr>
<tr>
<td>0x8048737</td>
<td>jmp 0804875c &lt;unparse_symbol+0x44&gt;</td>
<td>08048737</td>
</tr>
<tr>
<td>0x8048738</td>
<td>movl %esi,%esi</td>
<td>08048738</td>
</tr>
<tr>
<td>0x8048740</td>
<td>jmp 0804875c &lt;unparse_symbol+0x44&gt;</td>
<td>08048740</td>
</tr>
<tr>
<td>0x8048742</td>
<td>jmp 0804875c &lt;unparse_symbol+0x44&gt;</td>
<td>08048742</td>
</tr>
<tr>
<td>0x8048745</td>
<td>jmp 0804875c &lt;unparse_symbol+0x44&gt;</td>
<td>08048745</td>
</tr>
<tr>
<td>0x8048747</td>
<td>jmp 0804875c &lt;unparse_symbol+0x44&gt;</td>
<td>08048747</td>
</tr>
<tr>
<td>0x8048750</td>
<td>jmp 0804875c &lt;unparse_symbol+0x44&gt;</td>
<td>08048750</td>
</tr>
<tr>
<td>0x8048755</td>
<td>jmp 0804875c &lt;unparse_symbol+0x44&gt;</td>
<td>08048755</td>
</tr>
<tr>
<td>0x8048757</td>
<td>movl $0x3f,%eax</td>
<td>08048757</td>
</tr>
</tbody>
</table>

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

Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8048730</td>
<td>movl 0x2b</td>
<td>08048730</td>
</tr>
<tr>
<td>0x8048735</td>
<td>jmp 0804875c</td>
<td>08048735</td>
</tr>
<tr>
<td>0x8048737</td>
<td>movl 0x2a</td>
<td>08048737</td>
</tr>
<tr>
<td>0x8048738</td>
<td>jmp 0804875c</td>
<td>08048738</td>
</tr>
<tr>
<td>0x8048740</td>
<td>movl 0x2d</td>
<td>08048740</td>
</tr>
<tr>
<td>0x8048742</td>
<td>jmp 0804875c</td>
<td>08048742</td>
</tr>
<tr>
<td>0x8048745</td>
<td>movl 0x2f</td>
<td>08048745</td>
</tr>
<tr>
<td>0x8048747</td>
<td>jmp 0804875c</td>
<td>08048747</td>
</tr>
<tr>
<td>0x8048750</td>
<td>movl 0x3f</td>
<td>08048750</td>
</tr>
<tr>
<td>0x8048755</td>
<td>jmp 0804875c</td>
<td>08048755</td>
</tr>
<tr>
<td>0x8048757</td>
<td>movl 0x3f</td>
<td>08048757</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
Disclaimer

- Parts (most in this lecture) of the slides were developed by the course text authors: Dave O’Hallaron and Randy Bryant. The slides are intended for the sole purpose of instruction of computer organization at the University of Rochester. All copyrighted materials belong to their original owner(s).