Condition Codes

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

Implicity Set by Arithmetic Operations
- `add1 Src, Dest`
  - 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
- `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)`
### 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)</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></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

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

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

**Set Up**
```
pushl %ebp
movl %esp, %ebp
```

**Body**
```
movl 8(%ebp), %edx
movl 12(%ebp), %eax
```

**Finish**
```
movl %ebp, %esp
popl %ebp
ret
```
Conditional Branch Example (Cont.)

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

movl $(%ebp),%edx # edx = x
movl %edx,%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**

```
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;
}
```

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

“Do-While” Loop Compilation

**Goto Version**

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

**Assembly**

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

**Registers**

- %edx x
- %eax result

General “Do-While” Translation

**C Code**

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

**Goto Version**

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

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

Second Goto Version
```c
int fact_while_goto2 (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;
Body
while (Test);
done:
```

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

“For” Loop Example

```c
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: \( x^p = z_0 \cdot (z_2^2 \cdot \ldots \cdot (z_{2^{n-1}}^2 \cdot \ldots \cdot (z_{2^{n-2}}^2 \cdot \ldots \cdot (z_2^2 \cdot \ldots \cdot z_2^2) \cdot \ldots) \cdot \ldots) \) \( \cdot \ldots \cdot \cdot \) \n- Complexity \( O(\log p) \)

Example
\[
3^{10} = 3^1 \cdot 3^8 = 3^1 \cdot (3^2)^4 = 3^1 \cdot (3^{2})^2
\]
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>59049</td>
<td>43046721</td>
<td>0</td>
</tr>
</tbody>
</table>

“For” Loop Example

General Form

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

<table>
<thead>
<tr>
<th>Init</th>
<th>Test</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = 1</td>
<td>p != 0</td>
<td>p = p &gt;&gt; 1</td>
</tr>
</tbody>
</table>

Body

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

“For”→“While”

For Version

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

```
Body
```

Do-While Version

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

done:

```
Goto Version

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

```
loop:
    goto loop;
```

```
Update
```

```
result = 1;
```

```
p = p >> 1;
```

```
Goto Version

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

```
loop:
    goto loop;
```

```
Update
```

```
p = p >> 1;
```

```
done:
```

“For” Loop Compilation

Goto Version

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

```
loop:
    goto loop;
```

```
Update
```

```
p = p >> 1;
```

```
done:
```

```
result = 1;
```

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

```
result = 1;
```

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

```
p = p >> 1;
```

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

```
done:
```

```
delete:
```

```
result = 1;
```

```
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
  - Usually should also specify "default:" case

Jump Table Structure

Switch Form

Jump Targets

Jump Table Structure

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**

<table>
<thead>
<tr>
<th>.section .rodata</th>
<th>.align 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>.long .L51 #Op = 0</td>
<td></td>
</tr>
<tr>
<td>.long .L52 #Op = 1</td>
<td></td>
</tr>
<tr>
<td>.long .L53 #Op = 2</td>
<td></td>
</tr>
<tr>
<td>.long .L54 #Op = 3</td>
<td></td>
</tr>
<tr>
<td>.long .L55 #Op = 4</td>
<td></td>
</tr>
<tr>
<td>.long .L56 #Op = 5</td>
<td></td>
</tr>
</tbody>
</table>

**Enumerated Values**

- ADD 0
- MUL 1
- MINUS 2
- DIV 3
- MOD 4
- BAD 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

<table>
<thead>
<tr>
<th>.L49:</th>
<th># Done:</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl tebp,tebp # Finish</td>
<td></td>
</tr>
<tr>
<td>popl tebp # Finish</td>
<td></td>
</tr>
<tr>
<td>ret # Finish</td>
<td></td>
</tr>
</tbody>
</table>

---

### 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

```
0x8048718 <unparse_symbol>:
0x8048718: 55  pushl  %ebp
0x8048719: 89 e5  movl  %esp, %ebp
0x804871b: 8b 45 08  movl  0x8(%ebp), %eax
0x804871e: 83 f8 05  cmpl $0x5,%eax
0x8048721: 77 39  ja  0x804875c <unparse_symbol+0x44>
0x8048723: 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

E.g., 30870408 really means 0x08048730

Disassembled Targets

- Not practical to use jump table
- Would require 1000 entries
- Obvious translation into if-then-else would have max. of 9 tests

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 Structure

- Organizes cases as binary tree
- Logarithmic performance

Compare to possible case values
- Jumps different places depending on outcomes

movl $144, %eax
jmp L19
L20:
movl $222, %eax
jmp L19
L21:
movl $333, %eax
jmp L19
L22:
movl $444, %eax
jmp L19

movl $1, %eax
jmp L19
L4:
movl $111, %eax
jmp L17
L5:
movl $222, %eax
jmp L16
L6:
movl $333, %eax
jmp L15
L7:
movl $444, %eax
jmp L14
L8:
movl $555, %eax
jmp L13
L9:
movl $666, %eax
jmp L12
L10:
movl $777, %eax
jmp L11
L11:
movl $888, %eax
jmp L10
L12:
movl $999, %eax
jmp L9
L13:
movl $101010, %eax
jmp L8
L14:
movl $111111, %eax
jmp L7
L15:
movl $121212, %eax
jmp L6
L16:
movl $131313, %eax
jmp L5
L17:
movl $141414, %eax
jmp L4

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
- Use general registers to store condition information
- Special comparison instructions
- E.g., on ARM:
  - Set register $1 to 1 when Register $31 < 1
  - Set register $1 to 0 when

Standard Techniques
- All loops converted to do-while form
- Large switch statements use jump tables
- CISC machines generally have condition code registers