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

Machine-Level Programming II
Control Flow
Sept. 9, 1999

Topics

• Condition Codes
  – Setting
  – Testing

• Control Flow
  – If-then-else
  – Varieties of Loops
  – Switch Statements
Condition Codes

Single Bit Registers

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>Carry Flag</td>
</tr>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>SF</td>
<td>Sign Flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag</td>
</tr>
</tbody>
</table>

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

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>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>ZF</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

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

```assembly
movl 12(%ebp),%eax  # eax = y
cmpl %eax,8(%ebp)   # Compare x : eax
setg %al            # al = x > y
andl $255,%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>ZF</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;
}
```

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

Set Up

Body

Finish
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:
```

Skipped when x \leq y
“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;
    loop:
        result *= x;
        x = x-1;
        if (x > 1) goto loop;
    return result;
}
```

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

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;
}
```

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

Second Goto Version

```c
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;
}
```
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:
```
“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;
}

Algorithm

• Exploit property that \( p = p_0 + 2p_1 + 4p_2 + \ldots + 2^{n-1}p_{n-1} \)

• Gives: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}^2)^2)^2)^2 \)

\[
\begin{align*}
    z_i = 1 & \quad \text{when } p_1 = 0 \\
    z_i = x & \quad \text{when } p_1 = 1
\end{align*}
\]

• Complexity \( O(\log p) \)

Example

\( 3^{10} \)

\[
\begin{align*}
    &= 3^2 \times 3^8 \\
    &= 3^2 \times ((3^2)^2)^2
\end{align*}
\]
ipwr Computation

```c
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”

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

```
int result = 1;
if (!p) goto done;
do {
    if (p & 0x1)
        result *= x;
    x = x*x;
    p = p>>1;
} while (p);
done:
```

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

• Also converted conditional update into test and branch around update code
Example #2 Compilation

Goto Version

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

```asm
pushl %ebp          # Setup
movl %esp,%ebp     # Setup
movl $1,%eax       # eax = 1
movl 8(%ebp),%ecx  # ecx = x
movl 12(%ebp),%edx # edx = p
testl %edx,%edx    # Test p
    je L36          # If 0, goto done
L37:                 # Loop:
testb $1,%dl       # Test p & 0x1
    je L38          # If 0, goto skip
imull %ecx,%eax    # result *= x
L38:                 # Skip:
imull %ecx,%ecx    # x *= x
    shrl $1,%edx    # p >>= 1
    jne L37         # if p goto Loop
L36:                 # Done:
movl %ebp,%esp     # Finish
popl %ebp          # Finish
ret                 # Finish
```

Registers

- `%ecx` x
- `%edx` p
- `%eax` result
"For" Loop Example

General Form

\[
\text{for (Init; Test; Update )}
\]

\[
\text{Body}
\]

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

**Init**

result = 1

**Test**

p != 0

**Update**

p = p >> 1

**Body**

```
“For” → “While”

For Version

\[
\text{for (Init; Test; Update)} \\
\quad \text{Body}
\]

While Version

\[
\text{Init;} \\
\quad \text{while (Test)} \{ \\
\quad \quad \text{Body} \\
\quad \quad \text{Update ;} \\
\quad \}\n\]

Do-While Version

\[
\text{Init;} \\
\quad \text{if (!Test)} \\
\quad \quad \text{goto done;} \\
\quad \text{do } \{ \\
\quad \quad \text{Body} \\
\quad \quad \text{Update ;} \\
\quad \}\text{while (Test)} \\
\text{done:}
\]

Goto Version

\[
\text{Init;} \\
\quad \text{if (!Test)} \\
\quad \quad \text{goto done;} \\
\quad \text{loop:} \\
\quad \quad \text{Body} \\
\quad \quad \text{Update ;} \\
\quad \quad \text{if (Test)} \\
\quad \quad \quad \text{goto loop;} \\
\quad \}\text{done:}
\]
“For” Loop Compilation

Goto Version

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

Test
result = 1
p != 0

Update
p = p >> 1

Body
{ 
    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

```c
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:
    Targ0
    Targ1
    Targ2
    ...
    Targn-1
```

Jump Targets

```
Targ0:
    Code Block 0
Targ1:
    Code Block 1
Targ2:
    Code Block 2
Targn-1:
    Code Block n-1
```

Approx. Translation

```
target = JTab[op];
go to *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

<table>
<thead>
<tr>
<th>Value</th>
<th>Code</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>

Setup:

pushl %ebp # Setup
movl %esp,%ebp # Setup
movl 8(%ebp),%eax # eax = op
cmpl $5,%eax # Compare op : 5
ja .L64 # If > goto done
jmp *.L72(,%eax,4) # 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
- Jump .L64
- Jump target is denoted by label .L64
- Jump *.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

.L72:
  .long .L66 #Op = 0
  .long .L67 #Op = 1
  .long .L68 #Op = 2
  .long .L69 #Op = 3
  .long .L70 #Op = 4
  .long .L71 #Op = 5

Enumerated Values

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

Targets & Completion

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

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Opcode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>804875d:89 e5</td>
<td>movl</td>
<td>%esp,%ebp</td>
</tr>
<tr>
<td>804875f:8b 45 08</td>
<td>movl</td>
<td>0x8(%ebp),%eax</td>
</tr>
<tr>
<td>8048762:83 f8 05</td>
<td>cmpl</td>
<td>$0x5,%eax</td>
</tr>
<tr>
<td>8048765:77 4e</td>
<td>ja</td>
<td>80487b5</td>
</tr>
<tr>
<td>&lt;unparse_symbol+0x59&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8048767:ff 24 85 70 87</td>
<td>jmp</td>
<td>*0x8048770(,%eax,4)</td>
</tr>
</tbody>
</table>
Object Code (cont.)

Jump Table

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>804876c:</td>
<td>04 08</td>
</tr>
<tr>
<td>804876e:</td>
<td>89 f6</td>
</tr>
<tr>
<td>8048770:</td>
<td>88 87 04 08 90</td>
</tr>
<tr>
<td>8048775:</td>
<td>87</td>
</tr>
<tr>
<td>8048776:</td>
<td>04 08</td>
</tr>
<tr>
<td>8048778:</td>
<td>98</td>
</tr>
<tr>
<td>8048779:</td>
<td>87 04 08</td>
</tr>
<tr>
<td>804877c:</td>
<td>a0 87 04 08 a8</td>
</tr>
<tr>
<td>8048781:</td>
<td>87 04 08</td>
</tr>
<tr>
<td>8048784:</td>
<td>b0 87</td>
</tr>
<tr>
<td>8048786:</td>
<td>04 08</td>
</tr>
</tbody>
</table>

movl %esi,%esi
movb %al,0x87900804(%edi)
addb $0x8,%al
cwtl
xchgl %eax,(%eax,%ecx,1)
movb 0xa8080487,%al
xchgl %eax,(%eax,%ecx,1)
movb $0x87,%al
addb $0x8,%al
Decoding Jump Table

Known
- Starts at 8048770
- 4 bytes / entry
- Little Endian byte ordering

<table>
<thead>
<tr>
<th>Address</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048770</td>
<td>8048788</td>
</tr>
<tr>
<td>8048774</td>
<td>8048790</td>
</tr>
<tr>
<td>8048778</td>
<td>8048798</td>
</tr>
<tr>
<td>804877c</td>
<td>80487a0</td>
</tr>
<tr>
<td>8048780</td>
<td>80487a8</td>
</tr>
<tr>
<td>8048784</td>
<td>80487b0</td>
</tr>
</tbody>
</table>
Alternate Decoding Technique

Use GDB

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

0x8048770 <unparse_symbol+20>:
   0x08048788
   0x08048790
   0x08048798
   0x080487a0

0x8048780 <unparse_symbol+36>:
   0x080487a8
   0x080487b0
Disassembled Targets

- No-operations (nop) inserted to align target addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Immediate</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048788</td>
<td>b8 2b</td>
<td>00 00 00 00</td>
<td>movl $0x2b, %eax</td>
</tr>
<tr>
<td>804878d</td>
<td>eb 26</td>
<td></td>
<td>jmp 80487b5 &lt;unparse_symbol+0x59&gt;</td>
</tr>
<tr>
<td>804878f</td>
<td>90</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>8048790</td>
<td>b8 2a</td>
<td>00 00 00 00</td>
<td>movl $0x2a, %eax</td>
</tr>
<tr>
<td>8048795</td>
<td>eb 1e</td>
<td></td>
<td>jmp 80487b5 &lt;unparse_symbol+0x59&gt;</td>
</tr>
<tr>
<td>8048797</td>
<td>90</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>8048798</td>
<td>b8 2d</td>
<td>00 00 00 00</td>
<td>movl $0x2d, %eax</td>
</tr>
<tr>
<td>804879d</td>
<td>eb 16</td>
<td></td>
<td>jmp 80487b5 &lt;unparse_symbol+0x59&gt;</td>
</tr>
<tr>
<td>804879f</td>
<td>90</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>80487a0</td>
<td>b8 2f</td>
<td>00 00 00 00</td>
<td>movl $0x2f, %eax</td>
</tr>
<tr>
<td>80487a5</td>
<td>eb 0e</td>
<td></td>
<td>jmp 80487b5 &lt;unparse_symbol+0x59&gt;</td>
</tr>
<tr>
<td>80487a7</td>
<td>90</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>80487a8</td>
<td>b8 25</td>
<td>00 00 00 00</td>
<td>movl $0x25, %eax</td>
</tr>
<tr>
<td>80487ad</td>
<td>eb 06</td>
<td></td>
<td>jmp 80487b5 &lt;unparse_symbol+0x59&gt;</td>
</tr>
<tr>
<td>80487af</td>
<td>90</td>
<td></td>
<td>nop</td>
</tr>
<tr>
<td>80487b0</td>
<td>b8 3f</td>
<td>00 00 00 00</td>
<td>movl $0x3f, %eax</td>
</tr>
</tbody>
</table>
Matching Disassembled Targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048770:</td>
<td>08048788</td>
</tr>
<tr>
<td>8048774:</td>
<td>08048790</td>
</tr>
<tr>
<td>8048778:</td>
<td>08048798</td>
</tr>
<tr>
<td>804877c:</td>
<td>080487a0</td>
</tr>
<tr>
<td>8048780:</td>
<td>080487a8</td>
</tr>
<tr>
<td>8048784:</td>
<td>080487b0</td>
</tr>
</tbody>
</table>

- 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
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