Machine-Level Programming II: Control Flow

Today
- Condition codes
- Control flow structures

Next time
- Procedures
Condition codes

- Single bit registers
  - CF Carry Flag
  - SF Sign Flag
  - ZF Zero Flag
  - OF Overflow Flag

- Implicitly set by arithmetic operations
  - addl source,destination
  - 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
      - \((a > 0 \&\& b > 0 \&\& t < 0) \mid \mid (a < 0 \&\& b < 0 \&\& t \geq 0)\)

- Not set by leal instruction
Setting condition codes

- 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 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b) >0)
    ```
Setting condition codes

- Explicit setting by test instruction
  
  \( \text{testl } \text{Src}2, \text{Src}1 \)
  
  - Sets condition codes based on value of \( \text{Src}1 \) \& \( \text{Src}2 \)
    
    • Useful to have one of the operands be a mask
    
    - \( \text{testl } b, a \) like computing \( a \& b \) without setting destination
      
      - \( \text{ZF set when } a \& b == 0 \)
      
      - \( \text{SF set when } a \& b < 0 \)
Reading condition codes

- **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

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

```assembly
    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!
## 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>settle</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>
Checkpoint
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**

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

```assembly
_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

```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

```assembly
movl 8(%ebp),%edx  # edx = x
movl 12(%ebp),%eax  # eax = y
cmp %eax,%edx  # x : y
jle L9  # if <= goto L9
movl %edx,%eax  # eax = x
L9:            # Done:
             # Skipped when x ≤ 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;
}
```

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
```

- Registers
  - %edx x
  - %eax result
General “Do-While” translation

**C Code**

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

**Goto Version**

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

- **Body** can be any C statement
  - Typically compound statement:
    ```c
    {
      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:
Checkpoint
"For" loop example

Algorithm

- Exploit property that \( p = p_0 + 2p_1 + 4p_2 + \ldots + 2^{n-1}p_{n-1} \)
- Gives: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}^2)^2)\ldots)^2 \)

\[ z_i = 1 \quad \text{when} \quad p_i = 0 \]
\[ z_i = x \quad \text{when} \quad p_i = 1 \]

- Complexity \( O(\log p) \)

Example

\[ 3^{10} = 3^2 \cdot 3^8 \]
\[ = 3^2 \cdot ((3^2)^2)^2 \]
/** 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>
<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>


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

**General Form**

```
for (Init; Test; Update)

Body
```

---

**Init**

result = 1

**Test**

p != 0

**Update**

p = p >> 1

**Body**

```
{
    if (p & 0x1)
        result *= x;
    x = x*x;
}
```
“For” → “While”

For Version

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

Do-While Version

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

While Version

```plaintext
Init;  
  while (Test) {  
    Body  
    Update;  
  }
```

Goto Version

```plaintext
Init;  
  if (!Test)  
    goto done;  
  loop:  
    Body  
    Update;  
    if (Test)  
      goto loop;  
done:
```
"For" loop compilation

Goto Version

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

Init
result = 1

Test
p != 0

Update
p = p >> 1

Body

result = 1;
if (p == 0)
    goto done;
loop:
    if (p & 0x1)
        result *= x;
    x = x*x;
    p = p >> 1;
    if (p != 0)
        goto loop;
done:
Switch statements

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 val_0:
        Block 0
        
    case val_1:
        Block 1
        
    case val_n-1:
        Block n-1
    }
```

Jump table

<table>
<thead>
<tr>
<th>jtab:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ0</td>
</tr>
<tr>
<td>Targ1</td>
</tr>
<tr>
<td>Targ2</td>
</tr>
<tr>
<td>Targn-1</td>
</tr>
</tbody>
</table>

Jump targets

<table>
<thead>
<tr>
<th>Targ0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Block 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Targ1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Block 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Targ2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Block 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Targn-1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Block n-1</td>
</tr>
</tbody>
</table>

Approx. translation

```
target = JTab[op];
goto *target;
```
Switch statement example

Branching possibilities

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

char unparse_symbol(op_type op)
{
    switch (op) {
        • • •
    }
}

setup:

unparse_symbol:
    pushl %ebp           # Setup
    movl %esp,%ebp      # Setup
    movl 8(%ebp),%eax   # eax = op
    cmpl $5,%eax        # Compare op : 5
    ja .L49             # If > goto done
    jmp *.L57(,%eax,4)  # goto Table[op]

Enumerated values

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

```assembly
.sect .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

<table>
<thead>
<tr>
<th>Symbol</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>

### Targets & completion

```assembly
.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

```
.L49:   # Done:
    movl %ebp,%esp  # Finish
    popl %ebp      # Finish
    ret            # Finish
```

Monday, October 10, 2011
Object code

- Setup
  - Label \texttt{.L49} becomes address \texttt{0x804875c}
  - Label \texttt{.L57} becomes address \texttt{0x8048bc0}

\begin{verbatim}
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)
\end{verbatim}
Object code

- 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>:
  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 (otool on Mac’s)
  
  ```
  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

**Contents of section .rodata:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048bc0</td>
<td>30870408 37870408 40870408 47870408 0...7...@...G...</td>
</tr>
<tr>
<td>8048bd0</td>
<td>50870408 57870408 46616374 28256429 P...W...Fact(%d)</td>
</tr>
<tr>
<td>8048be0</td>
<td>203d2025 6c640a00 43686172 203d2025 = %ld..Char = %</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Disassembled targets

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048730:</td>
<td><code>b8 2b 00 00 00</code></td>
<td><code>movl $0x2b, %eax</code></td>
</tr>
<tr>
<td>8048735:</td>
<td><code>eb 25</code></td>
<td><code>jmp 804875c &lt;unparse_symbol+0x44&gt;</code></td>
</tr>
<tr>
<td>8048737:</td>
<td><code>b8 2a 00 00 00</code></td>
<td><code>movl $0x2a, %eax</code></td>
</tr>
<tr>
<td>804873c:</td>
<td><code>eb 1e</code></td>
<td><code>jmp 804875c &lt;unparse_symbol+0x44&gt;</code></td>
</tr>
<tr>
<td>804873e:</td>
<td><code>89 f6</code></td>
<td><code>movl %esi, %esi</code></td>
</tr>
<tr>
<td>8048740:</td>
<td><code>b8 2d 00 00 00</code></td>
<td><code>movl $0x2d, %eax</code></td>
</tr>
<tr>
<td>8048745:</td>
<td><code>eb 15</code></td>
<td><code>jmp 804875c &lt;unparse_symbol+0x44&gt;</code></td>
</tr>
<tr>
<td>8048747:</td>
<td><code>b8 2f 00 00 00</code></td>
<td><code>movl $0x2f, %eax</code></td>
</tr>
<tr>
<td>804874c:</td>
<td><code>eb 0e</code></td>
<td><code>jmp 804875c &lt;unparse_symbol+0x44&gt;</code></td>
</tr>
<tr>
<td>804874e:</td>
<td><code>89 f6</code></td>
<td><code>movl %esi, %esi</code></td>
</tr>
<tr>
<td>8048750:</td>
<td><code>b8 25 00 00 00</code></td>
<td><code>movl $0x25, %eax</code></td>
</tr>
<tr>
<td>8048755:</td>
<td><code>eb 05</code></td>
<td><code>jmp 804875c &lt;unparse_symbol+0x44&gt;</code></td>
</tr>
<tr>
<td>8048757:</td>
<td><code>b8 3f 00 00 00</code></td>
<td><code>movl $0x3f, %eax</code></td>
</tr>
</tbody>
</table>
Matching disassembled targets

Entry

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

<table>
<thead>
<tr>
<th>Entry</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048730</td>
<td>b8 2b 00 00 00              movl</td>
</tr>
<tr>
<td>0x08048735</td>
<td>eb 25                         jmp</td>
</tr>
<tr>
<td>0x08048737</td>
<td>b8 2a 00 00 00              movl</td>
</tr>
<tr>
<td>0x0804873c</td>
<td>eb 1e                         jmp</td>
</tr>
<tr>
<td>0x08048740</td>
<td>89 f6                        movl</td>
</tr>
<tr>
<td>0x08048745</td>
<td>eb 15                         jmp</td>
</tr>
<tr>
<td>0x08048747</td>
<td>b8 2d 00 00 00              movl</td>
</tr>
<tr>
<td>0x0804874c</td>
<td>eb 0e                         jmp</td>
</tr>
<tr>
<td>0x08048750</td>
<td>89 f6                        movl</td>
</tr>
<tr>
<td>0x08048755</td>
<td>eb 25                         jmp</td>
</tr>
<tr>
<td>0x08048757</td>
<td>b8 3f 00 00 00              movl</td>
</tr>
</tbody>
</table>
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

```c
/* Return x/111 if x is multiple && <= 999.
   -1 otherwise */
int div111(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 code

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

```assembly
movl 8(%ebp),%eax  # get x
cmpl $444,%eax    # x:444
je L8
jg L16

movl $111,%eax    # x:111
je L5
jg L17

movl $1,%eax     # x:0
je L4
jmp L14

. . .

L5:
    movl $1,%eax
    jmp L19

L6:
    movl $2,%eax
    jmp L19

L7:
    movl $3,%eax
    jmp L19

L8:
    movl $4,%eax
    jmp L19

. . .
```
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 → do-while form
  - Large switch statements use jump tables
- **Conditions in CISC**
  - Machines generally have condition code registers
- **Conditions in RISC**
  - Use general registers
  - Special comparison instructions