Machine-Level Programming II: Control Flow

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/JDEP284

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
- `addl` `Src, Dest`
  - 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
- `cmp l Src2, Src1`
  - `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` or `b > a`
  - OF set if two's complement overflow
- `(a<0 && b<0 && (a-b)<0) || (a>0 && b>0 && (a-b)>0)`

Explicit Setting by Test Instruction
- `test l 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>

SetX Instructions (Cont.)
- One of 8 addressable byte registers
- Embedded within first 4 integer registers
- Does not alter remaining 3 bytes
- Typically use movzbl to finish job

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

```
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:
popl %ebp
ret
```

```
Skipped when x \leq y
```

Conditional Branch Example (Cont.)

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

```
movl 8(%ebp),%edx # edx = x
movl 12(%ebp),%eax # eax = y
cmpl %eax,%edx # x = y
jle L9 # if x < goto L9
movl %edx,%eax # eax = x
```

```
L9: # Done:
    return %eax;
```

```
Do-While" Loop Example

C Code
```
int fact_do

    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 Example

```
### “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
  { Statement1; Statement2; Statement3;
  }
  ```
- **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:
        result *= x;
        x = x-1;
        if (!(x > 1)) goto done;
    return result;
}
```

- **Is this code equivalent to the do-while version?**
- **Must jump out of loop if test fails**

**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;
}
```

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

### 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**

```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;
    }
    return result;
}
```

**“For” Loop Example**

```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;
    }
    return result;
}
```

**Algorithm**

- **Exploit property that**
  ```c
  p = p_0 + 2p_1 + 4p_2 + \ldots + 2^{\lceil \log_2 p_0 \rceil}p_{\lceil \log_2 p_0 \rceil}
  ```
- **Gives:**
  ```c
  x = x \cdot (x \cdot (x \cdot \ldots ))^{2^{\lceil \log_2 p \rceil}}
  ```
- **z_i = 1 when p_i = 0**
- **Complexity O(log p)**

**Example**

```c
3^3 = 3 \cdot 3 \cdot 3
   = 3^2 \cdot 3\cdot 1
   = 27
```
ipwr Computation

/* 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;
}

result  x  p
1  1  2
1  2  1
2  4  2
5  32  3
59049  65536  0

"For" Loop Example

General Form
for (Init; Test; Update )
  Body

Init    Test    Update
result = 1    p = 0    p = p >> 1
{ 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:

While Version
Init;
while (Test)
  Body
  Update;

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

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, 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 '?';
    }
}

Jump Table Structure

Switch Form

Jump Table

Jump Targets

Approx. Translation

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 '?';
    }
}

Jump Table Structure

Jump Targets
Switch Statement Example

Enumerated Values

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

Branching Possibilities

```
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]
```

Jumping

```
jmp .L49  # Jump target is denoted by label .L49
```

```
jmp *.L57(,%eax,4)  # Start of jump table denoted by label .L57
```

```
reg %eax holds op
```

```
must scale by factor of 4 to get offset into table
```

Assembly Setup Explanation

Symbolic Labels

- Labels of form .LXX translated into addresses by assembler
- Each target requires 4 bytes
- Base address at .L57

Table Structure

- Jump target is denoted by label .L49
- 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
```

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

```
.L49:
 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

```
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]
```

Jump Table

```
jmp .L49  # Jump target is denoted by label .L49
```

```
jmp *.L57(,%eax,4)  # Start of jump table denoted by label .L57
```

```
reg %eax holds op
```

```
must scale by factor of 4 to get offset into table
```

Object Code (cont.)

Jump Table

- Doesn't show up in disassembled code
- Can inspect using GDB
- Use command "help x" to get format documentation
- Examine 4-byte decimal format "words" (4-bytes each)
- Use command "hex x" to get format documentation

```
0x08048bc0 <_fini+32>:  
 0x08048730  
 0x08048737  
 0x08048740  
 0x08048747  
 0x08048750  
 0x08048757
```

```
0x0804875c <unparse_symbol+0x44>:
```
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

Contents of section .rodata:
8048bc0 30870408 37870408 40870408 47870408 0...7...@
8048bd0 50870408 57870408 46616374 28256429 P...W...Fact(%d)
8048be0 203d2025 6c640a00 43686172 203d2025 = %ld..Char = %

E.g., 30870408 really means 0x08048730

Disassembled Targets
0x8048730:b8 2b 00 00 00 movl $0x2b,%eax
0x8048735:eb 25          jmp 804875c <unparse_symbol+0x44>
0x8048737:b8 2a 00 00 00 movl $0x2a,%eax
0x804873c:eb 1e          jmp 804875c <unparse_symbol+0x44>
0x804873e:89 f6          movl %esi,%esi
0x8048740:b8 2d 00 00 00 movl $0x2d,%eax
0x8048745:eb 15          jmp 804875c <unparse_symbol+0x44>
0x8048747:b8 2f 00 00 00 movl $0x2f,%eax
0x804874c:eb 0e          jmp 804875c <unparse_symbol+0x44>
0x804874e:89 f6          movl %esi,%esi
0x8048750:b8 25 00 00 00 movl $0x25,%eax
0x8048755:eb 05          jmp 804875c <unparse_symbol+0x44>
0x8048757:b8 3f 00 00 00 movl $0x3f,%eax

Matching Disassembled Targets

Sparse Switch Example
/* 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
movl E(%ebp),%eax # get x
cmpl $444,%eax   # x:444
je L5
jg L7
cmp $111,%eax   # x:111
je L1
jg L3
testl %eax,%eax # x:0
je L4
jmp L4
L5:
  movl $2,%eax
  jmp L9
L7:
  movl $3,%eax
  jmp L9
L9:
  movl $4,%eax
  jmp L9

Sparse Switch Code Structure
- Organizes cases as binary tree
- Logarithmic performance

Disassembled Targets
- movl basi,basi does nothing
- Inserted to align instructions for better cache performance
### Summarizing

<table>
<thead>
<tr>
<th>Control</th>
<th>Standard Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>All loops converted to do-while form</td>
</tr>
<tr>
<td></td>
<td>Large switch statements use jump tables</td>
</tr>
<tr>
<td>Assembler Control</td>
<td>CISC machines generally have condition code registers</td>
</tr>
<tr>
<td>Compiler</td>
<td>Conditions in RISC</td>
</tr>
<tr>
<td></td>
<td>Use general registers to store condition information</td>
</tr>
<tr>
<td></td>
<td>Special comparison instructions</td>
</tr>
<tr>
<td></td>
<td>E.g., on Alpha:</td>
</tr>
<tr>
<td></td>
<td><code>cmp $16,1,$1</code></td>
</tr>
<tr>
<td></td>
<td>Sets register $1 to 1 when Register $16 == 1</td>
</tr>
</tbody>
</table>

- if-then-else
- do-while
- while
- switch

- jump
- Conditional jump

- Must generate assembly code to implement more complex control