Machine-Level Programming II:
Control Flow
Sept. 21, 2009

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

addr  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
  \((a > 0 \land b > 0 \land t < 0) \lor (a < 0 \land b < 0 \land t \geq 0)\)

Not Set by load instruction
Explicit Setting by Compare Instruction

\texttt{cmp}l \textit{Src2,Src1}

- \texttt{cmp}l \texttt{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
  \((a > 0 \&\& b < 0 \&\& (a-b) < 0) \text{ || } (a < 0 \&\& b > 0 \&\& (a-b) > 0)\)
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>settle</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

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

Body

```
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>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;
}
```

```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 (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

```assembly
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;
}
```

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

<table>
<thead>
<tr>
<th>C Code</th>
<th>Goto Version</th>
</tr>
</thead>
</table>
| ```
  do 
  Body 
  while (Test) ;
``` | ```
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

```c
while (Test)
    Body
```

Do-While Version

```c
if (!Test)
    goto done;
do
    Body
    while(Test);
done:
```

Goto Version

```c
if (!Test)
    goto done;
loop:
    Body
    if (Test)
        goto loop;
done:
```
"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*x;
    }
    return result;
}
```

Algorithm

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

Example

\[
3^{10} = 3^2 \cdot 3^8 = 3^2 \cdot ((3^2)^2)^2
\]
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;
}

<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>
**“For” Loop Example**

```
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**

```c
for (Init; Test; Update) {
    Body
}
```

**While Version**

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

**Do-While Version**

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

**Goto Version**

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

cchar 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

```java
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>
<th>Targ0</th>
<th>Targ1</th>
<th>Targ2</th>
<th>...</th>
<th>Targn-1</th>
</tr>
</thead>
</table>

Jump Targets

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targn-1: Code Block n-1

Approx. Translation

```java
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) {
    • • •
  }
}

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
.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

```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

.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

```
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
  8048723:  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>:
  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

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

- E.g., 30870408 really means 0x08048730
Disassembled Targets

8048730: b8 2b 00 00 00 movl $0x2b,%eax
8048735: eb 25 jmp 804875c

<unparse_symbol+0x44>
8048737: b8 2a 00 00 00 movl $0x2a,%eax
804873c: eb 1e jmp 804875c

<unparse_symbol+0x44>
804873e: 89 f6 movl %esi,%esi
8048740: b8 2d 00 00 00 movl $0x2d,%eax
8048745: eb 15 jmp 804875c

<unparse_symbol+0x44>
8048747: b8 2f 00 00 00 movl $0x2f,%eax
804874c: eb 0e jmp 804875c

<unparse_symbol+0x44>
804874e: 89 f6 movl %esi,%esi
8048750: b8 25 00 00 00 movl $0x25,%eax
8048755: eb 05 jmp 804875c

<unparse_symbol+0x44>
8048757: b8 3f 00 00 00 movl $0x3f,%eax

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

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

 Movl 8048730:b8 2b 00 00 00
 Movl 8048735:eb 25
 Movl 8048737:b8 2a 00 00 00
 Movl 804873c:eb 1e
 Movl 804873e:89 f6
 Movl 8048740:b8 2d 00 00 00
 Movl 8048745:eb 15
 Movl 8048747:b8 2f 00 00 00
 Movl 804874c:eb 0e
 Movl 804874e:89 f6
 Movl 8048750:b8 25 00 00 00
 Movl 8048755:eb 05
 Movl 8048757:b8 3f 00 00 00

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