15-213
“The course that gives CMU its Zip!”
Machine-Level Programming II: Control Flow
Sept. 09, 2008

Topics
- Condition Codes
  - Setting
  - Testing
- Control Flow
  - If-then-else
  - Varieties of Loops
  - Switch Statements
- x86-64 features
  - conditional move
  - different loop implementation
- x86-64 features
  - conditional move
  - different loop implementation

Processor State (IA32, Partial)

Information about currently executing program
- Temporary data
- Location of current code control point
- Location of runtime stack
- Status of recent tests

General purpose registers
- %eax
- %edx
- %ecx
- %ebx
- %esi
- %edi
- %esp
- %ebp
- %eip

Current stack top
Current stack frame
Instruction pointer
Condition codes

Condition Codes

Single Bit Registers
- CF Carry Flag
- ZF Zero Flag

Implicitly Set By Arithmetic Operations
- addl Src, Dest
  C analog: t = a + b
  (a = Src, b = Dest)
- addq 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 lea, inc, or dec instructions

Explicit Setting by Compare Instruction
- cmpl Src2,Src1
- cmpq 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 (cont.)

Explicit Setting by Test instruction

- `testl Src2, Src1`
- `testq 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: x86-64

SetX Instructions

- Set single byte based on combinations of condition codes
  - Does not alter remaining 7 bytes
- x86-64 arguments
  - x in %rdi
  - y in %rsi

Body (same for both)

```c
long lgt (long x, long y) {
    return x > y;
}
```

Note: 32-bit instructions set high order 32 bits to 0

Body

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

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^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 absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
    int result;
    if (x<=y) goto Else;
    result = x-y;

    Exit:
    return result;

    Else:
    result = y-x;
    goto Exit;
}
```

General Conditional Expression Translation

<table>
<thead>
<tr>
<th>C Code</th>
<th>Goto Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>val = Test ? Then-Expr : Else-Expr;</td>
<td>val = x&gt;y ? x-y : y-x;</td>
</tr>
</tbody>
</table>

- Test is expression returning integer
  - 0 interpreted as false
  - ≠0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one
Conditionals: x86-64

```
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

Conditional move instruction
- `cmovC src, dest`
- Move value from src to dest if condition `C` holds
- More efficient than conditional branching
  » Simple & predictable control flow

General Form with Conditional Move

```
val = Test ? Then-Expr ? Else-Expr;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn’t hold

```
val = Then-Expr;
vale = Else-Expr;
val = vale if !Test;
```

Limitations of Conditional Move

```
val = Then-Expr;
vale = Else-Expr;
val = vale if !Test;
```

Limitations:
- Then-Expr or Else-Expr has side effect
- Then-Expr or Else-Expr requires significant computation

```
int xgty = 0, xltey = 0;
```

```
int absdiff_se(int x, int y)
{
    int result;
    if (x > y) {
        xgty++; result = x-y;
    } else {
        xltey++; result = y-x;
    }
    return result;
}
```

Implementing Loops

IA32
- All loops translated into form based on “do-while”

x86-64
- Also make use of “jump to middle”

Why the Difference
- IA32 compiler developed for machine where all operations costly
- x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead
“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:
    push %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
```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
Alternative “While” Loop Translation

C Code
```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```
- Historically used by GCC
- 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
    goto loop;
done:
```

New Style “While” Loop Translation

C Code
```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x = x-1;
    }
    return result;
}
```
- Recent technique for GCC
  - Both IA32 & x86-64
- First iteration jumps over body computation within loop

Goto Version
```c
int fact_while goto3(int x) {
    int result = 1;
    goto middle;
    loop:
        result *= x;
        x = x-1;
    middle:
        if (x > 1)
            goto loop;
    return result;
}
```

Jump-to-Middle While Translation

C Code
```c
while (Test) {
    Body
}
```

Goto Version
```c
goto middle;
loop:
    Body
    goto loop;
middle:
    if (Test)
        goto loop;
```
Jump-to-Middle Example

```c
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x--;
    }
    return result;
}
```

Most common strategy for recent IA32 & x86-64 code generation

```
# x in %edx, result in %eax
jmp L34        # goto Middle
L35:               # Loop:
imull %edx, %eax # result *= x
decl %edx        # x--
L34:               # Middle:
cmpi $1, %edx # x:1
jg L35           # if >, goto Loop
```

“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: $x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot ((z_{n-1}^2)^2)^{2^{n-1}}$
- $z_1 = 1$ when $p_1 = 0$
- $z_i = x$ when $p_i = 1$
- Complexity $O(\log p)$

```
Example
$3^{10} = 3^2 \cdot 3^8$
$= 3^2 \cdot (3^7)^2$
```

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;
}
```

“For” Loop Example

```
result x p
1 3 10
1 9 5
9 81 2
9 6561 1
531441 43046721 0
```

General Form

- **Init**
  - `result = 1`
- **Test**
  - `p != 0`
- **Update**
  - `p = p >> 1`

```
Body
{ if (p & 0x1)
    result *= x;
    x = x*x;
}
```
Switch Statements

Implementation Options
- Series of conditionals
  - Organize in tree structure
  - Logarithmic performance
- Jump Table
  - Lookup branch target
  - Constant time
  - Possible when cases are small integer constants
- GCC
  - Picks one based on case structure

Switch Statement Example

Features
- Multiple case labels
- Fall through cases
- Missing cases

Switch Statement Example (IA32)

Setup:
```assembly
switch_eg:
pushl %ebp # Setup
movl %esp, %ebp # Setup
pushl %ebx # Setup
movl $1, %ebx # w = 1
movl 8(%ebp), %edx # edx = x
movl 16(%ebp), %ecx # ecx = z
cmpl $6, %edx # x:6
ja .L61 # if > goto default
jmp *.L62(%edx,4) # goto JTab[x]
```

Jump Table Structure

Switch Form
```
switch(x) {
  case val_0: Block 0
  case val_1: Block 1
  ... 
  case val_n-1: Block n-1
}
```

Jump Table
```
jtab:
  Targ0: Code Block 0
  Targ1: Code Block 1
  ... 
  Targn-1: Code Block n-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[x];
goto *target;
```
Assembly Setup Explanation

Table Structure
- Each target requires 4 bytes
- Base address at .L62

Jumping
- jmp .L61
- Jump target is denoted by label .L61
- jmp *.L62(%edx,4)
- Start of jump table denoted by label .L62
- Register %edx holds x
- Must scale by factor of 4 to get offset into table
- Fetch target from effective Address .L61 + x*4
  - Only for 0 ≤ x ≤ 6

Jump Table

Table Contents

```assembly
.section .rodata
.align 4
.L62:
    .long .L61 # x = 0
    .long .L62 # x = 1
    .long .L57 # x = 2
    .long .L58 # x = 3
    .long .L61 # x = 4
    .long .L60 # x = 5
    .long .L60 # x = 6
```

switch(x) {
  case 1:      // .L56
    w = y*z;
    break;
  case 2:      // .L57
    w = y/z;
    /* Fall Through */
  case 3:      // .L58
    w += z;
    break;
  case 5:
  case 6:      // .L60
    w -= z;
    break;
  default:     // .L61
    w = 2;
}

Code Blocks (Partial)

```
switch(x) {
  ...
  case 2:      // .L57
    w = y/z;
    leave
    ret
    case 3:      // .L58
    w += z;
    break;
    ...
    default:     // .L61
    w = 2;
}
```

```
.L61: // Default case
    movl $2, %ebx # w = 2
    movl %ebx, %eax # Return w
    popl %ebx
    leave
    ret

.L57: // Case 2:
    movl 12(%ebp), %ebx # w = yimull %ecx, %ebx # w*= z
    movl %ebx, %eax # Return w
    popl %ebx
    leave
    ret

.L58: // Case 3:
    addl %ecx, %ebx # w+= z
    movl %ebx, %eax # Return w
    popl %ebx
    leave
    ret
```

Code Blocks (Rest)

```
switch(x) {
  ...
  case 5:
  case 6:      // .L60
    w -= z;
    break;
    ...
}
```

```
.L60: // Cases 5&6:
    subl %ecx, %ebx # w -= z
    movl %ebx, %eax # Return w
    popl %ebx
    leave
    ret

.L56: // Case 1:
    movl 12(%ebp), %ebx # w = y
    imull %ecx, %ebx # w*= z
    movl %ebx, %eax # Return w
    popl %ebx
    leave
    ret
```
x86-64 Switch Implementation

- Same general idea, adapted to 64-bit code
- Table entries 64 bits (pointers)
- Cases use revised code

Jump Table

```asm
.section .rodata
.align 8
.L62:
.quad .L55  # x = 0
.quad .L50  # x = 1
.quad .L51  # x = 2
.quad .L52  # x = 3
.quad .L55  # x = 4
.quad .L54  # x = 5
.quad .L54  # x = 6

.L50: // Case 1:
    movq  %rsi, %r8  # w = y
    imulq  %rdx, %r8  # w *= z
    movq  %r8, %rax  # Return w
    ret
```

IA32 Object Code

Setup

- Label .L61 becomes address 0x8048630
- Label .L62 becomes address 0x80488dc

Assembly Code

```
switch(x) {
    case 1:      // .L50
        w = y*z;
        break;
        ...}
```

Disassembled Object Code

```
08048610 <switch_eg>:
    ...     
08048622:  77 0c                   ja 0x8048610
08048624:  ff 24 95 dc 88 04 08    jmp *0x80488dc(%edx,4)
```

IA32 Object Code (cont.)

Jump Table

- Doesn’t show up in disassembled code
- Can inspect using GDB
  ```bash
gdb asm-cntl
  (gdb) x/7xw 0x80488dc
  ```
  - Examine `7` hexadecimal format “words” (4-bytes each)
  - Use command “help x” to get format documentation

Disassembled Targets

```
08048630:  bb 02 00 00 00  mov $0x2,%ebx
08048635:  89 d8                   mov %ebx,%eax
08048637:  5b                      pop %ebx
08048638:  c9                      leave
08048639:  c3                      ret
0804863a:  8b 45 0c                mov 0xc(%ebp),%eax
0804863d:  99                      cltd
0804863e:  f7 f9                   idiv %ecx
08048640:  89 c3                   mov %eax,%ebx
08048642:  01 cb add %ecx,%ebx
08048644:  89 d8                   mov %ebx,%eax
08048646:  5b                      pop %ebx
08048647:  c9                      leave
08048648:  c3                      ret
08048649:  29 cb sub %ecx,%ebx
0804864b:  89 d8                   mov %ebx,%eax
0804864d:  5b                      pop %ebx
0804864e:  c9                      leave
0804864f:  c3                      ret
08048650:  8b 5d 0c                mov 0xc(%ebp),%ebx
08048653:  0f af d9                imul %ecx,%ebx
08048656:  89 d8                   mov %ebx,%eax
08048658:  5b                      pop %ebx
08048659:  c9                      leave
0804865a:  c3                      ret
```
Matching Disassembled Targets

x86-64 Object Code

Setup
- Label .L61 becomes address 0x0000000000400716
- Label .L62 becomes address 0x0000000000400990

Assembly Code

```
switch_eg:
    ...  
    ja .L55 # if > goto default
    jmp *.L56(,%rdi,8) # goto JTab[x]
```

Disassembled Object Code

```
0000000000400700 <switch_eg>:
    ...  
    40070d:  77 07                 ja 400716
    40070f:  ff 24 fd 90 09 40 00  jmpq *0x400990(,%rdi,8)
```

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;
    }
}
```

x86-64 Object Code (cont.)

Jump Table
- Can inspect using GDB
gdb asm-cntl
(gdb) x/7xg 0x400990
  
- Examine 7 hexadecimal format “giant words” (8-bytes each)
- Use command “help x” to get format documentation
0x400990:
  0x0000000000400716
  0x0000000000400739
  0x0000000000400720
  0x000000000040072b
  0x0000000000400716
  0x0000000000400732
  0x0000000000400732
Sparse Switch Code (IA32)

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

```
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, for
- switch

Assembler Control
- Conditional jump
- Conditional move
- Indirect jump

Compiler
- Must generate assembly code to implement more complex control

Standard Techniques
- IA32 loops converted to do-while form
- x86-64 loops use jump-to-middle
- Large switch statements use jump tables

Conditions in CISC
- CISC machines generally have condition code registers