The Hardware/Software Interface
CSE351 Winter 2013

x86 Programming II

Conditionals and Control Flow

- A conditional branch is sufficient to implement most control flow constructs offered in higher level languages
  - if (condition) then (...) else (...)
  - while (condition) (...)
  - do (...) while (condition)
  - for (initialization; condition; iteration) (...)

- Unconditional branches implement some related control flow constructs
  - break, continue

- In x86, we'll refer to branches as “jumps” (either conditional or unconditional)

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>SF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~SF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>ja</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>
Processor State (IA32, Partial)

- Information about currently executing program
  - Temporary data (%eax, …)
  - Location of runtime stack (%ebp, %esp)
  - Location of current code control point (%eip)
  - Status of recent tests (ZF, SF, OF)

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ecx</th>
<th>%edx</th>
<th>%ebx</th>
<th>%esi</th>
<th>%edi</th>
<th>%esp</th>
<th>%ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

General purpose registers

Current stack top

Current stack frame

Instruction pointer

Condition codes

Condition Codes (Implicit Setting)

- Single-bit registers
  - CF Carry Flag (for unsigned)
  - ZF Zero Flag
  - OF Overflow Flag (for signed)
  - Implicitly set (think of it as side effect) by arithmetic operations
    Example: addl/addrq Src, Dest \( \leftrightarrow t = a+b \)
  - CF set if carry out from most significant bit (unsigned overflow)
  - ZF set if \( t = 0 \)
  - SF set if \( t < 0 \) (as signed)
  - OF set if two's complement (signed) overflow
    \((a>0 \&\& b>0 \&\& t<0) \lor (a<0 \&\& b<0 \&\& t>=0)\)

- Not set by lea instruction (beware!)

- Full documentation (IA32): http://www.jegerlehner.ch/intel/IntelCodeTable.pdf

Condition Codes (Explicit Setting: Compare)

- Single-bit registers
  - CF Carry Flag (for unsigned)
  - SF Sign Flag (for signed)

- Explicit Setting by Compare Instruction
  - `cmp`/`cmpq` `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 \( a-b < 0 \) (as signed)
    - OF set if two's complement (signed) overflow
      \((a>0 \&\& b<0 \&\& (a-b)<0) \lor (a<0 \&\& b>0 \&\& (a-b)>0)\)

Condition Codes (Explicit Setting: Test)

- Single-bit registers
  - CF Carry Flag (for unsigned)
  - SF Sign Flag (for signed)

- Explicit Setting by Test instruction
  - `testl/testq` `Src2,Src1`
    - `test l b,a` like computing \( a \& b \) without setting destination
    - Sets condition codes based on value of `Src1` & `Src2`
    - Useful to have one of the operands be a mask
  - ZF set if \( a\&b == 0 \)
  - SF set if \( a\&b < 0 \)

- `testl %eax, %eax`
  - Sets SF and ZF, check if eax is +,0,-
Reading Condition Codes

**SetX Instructions**
- Set a single byte to 0 or 1 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>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>

**One of 8 addressable byte registers**
- Does not alter remaining 3 bytes
- Typically use movzb1 to finish job

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

Body: y at 12(%ebp), x at 8(%ebp)
```assembly
movl 12(%ebp),%eax  # eax = y
cmpl %eax,8(%ebp)   # Compare x and y
going (x > y)        # Zero rest of %eax
```

Reading Condition Codes (Cont.)

**Jumping**
- 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>SF</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 absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

`absdiff:`

```assembly
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmpl %eax, %edx
jle .L7
subl %eax, %edx
movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

Conditional Branch Example (Cont.)

```c
int absdiff(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
Else:
    result = y-x;
goto Exit;
}
```

`absdiff:`

```assembly
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmpl %eax, %edx
jle .L7
subl %eax, %edx
movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

---

Conditional Branch Example (Cont.)

```c
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;
}
```

`goto_ad:`

```assembly
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmpl %eax, %edx
jle .L7
subl %eax, %edx
movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

---

Conditional Branch Example (Cont.)

C allows "goto" as means of transferring control

- Closer to machine-level programming style
- Generally considered bad coding style
Conditional Branch Example (Cont.)

```c
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;
}
```

```
int x %edx
int y %eax
```

Conditional Branch Example (Cont.)

```asm
absdiff:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmpl %eax, %edx
jle .L7
subl %eax, %edx
movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

```
int x %edx
int y %eax
```

General Conditional Expression Translation

**C Code**

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

**Goto Version**

```c
int = Test;
    if (int) goto Else;
    val = Then-Expr;
    Done:
        ...
    Else:
        val = Else-Expr;
        goto Done;
```

- Test is expression returning integer
- !=0 interpreted as true
- !=0 interpreted as false
- Create separate code regions for then & else expressions
- Execute appropriate one
- How might you make this more efficient?
### Conditionals: x86-64

- **Conditional move instruction**
  - `cmovC src, dest`  
  - Move value from `src` to `dest` if condition `C` holds  
  - More efficient than conditional branching (simple control flow)  
  - But overhead: both branches are evaluated

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

- **PC Relative Addressing**
  - PC relative branches are relocatable
  - Absolute branches are not

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

```c
0x100     cmp  r2, r3    0x1000
0x102     je   0x70     0x1002
0x104     …       …      0x1004
…         …       …
0x172     add  r3, r4  0x1072
```

- **Compiling Loops**
  - How to compile other loops should be straightforward
    - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop
  - How would `for(i=0; i<100; i++)` be implemented?

```c
while (sum != 0) {
    <loop body>
}
```

```
loopTop:     cmpl $0, %eax
            je  loopDone
            jmp  loopTop

loopDone:
```

```
```

- **“Do-While” Loop Example**
  - Use backward branch to continue looping
  - Only take branch when “while” condition holds

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

```c
int fact_goto(int x) {
    int result = 1;
    loop:
        result *= x;
        x = x - 1;
    if (x > 1) goto loop;
    return result;
}
```
### “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
    movl %esp,%ebp
    movl $1,%eax
    movl $1,%edx
.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```

**General “Do-While” Translation**

- **C Code**
  ```c
do
    Body
    while (Test);

    Body: {
        Statement;
        Statement;
        ... 
        Statement;
    }

    Test returns integer
    = 0 interpreted as false
    ≠ 0 interpreted as true
  ```

- **Goto Version**
  ```c
loop:
    Body
    if (Test) goto loop
  ```

- **Assembly**
  ```
  Translation?
  ```

### “While” Loop Translation

**C Code**

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

**Goto Version**

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

- **Used by GCC for both IA32 & x86-64**
- **First iteration jumps over body computation within loop straight to test**
**“While” Loop Example**

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

**“For” Loop Example: Square-and-Multiply**

```c
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

**ipwr Computation**

```c
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x*x;
    }
    return result;
}
```

**“For” Loop Example**

```c
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned int 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>Iteration</th>
<th>result</th>
<th>x=3</th>
<th>p=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>10=1010</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5=101</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>81</td>
<td>2=10</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6561</td>
<td>1=1</td>
</tr>
<tr>
<td>5</td>
<td>59049</td>
<td>43046721</td>
<td>0</td>
</tr>
</tbody>
</table>
“For” → “While”

For Version

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

Goto Version

```c
Init;
    goto middle;
loop:
    Body
    Update;
    middle:
    if (Test)
    goto loop;
    done:
```

While Version

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

Quick Review

- Complete memory addressing mode
  - (%eax), 17(%eax), 2(%ebx, %ecx, 8), ...

- Arithmetic operations that do set condition codes
  - subl %eax, %ecx    # ecx = ecx + eax
  - sal $4,%edx      # edx = edx << 4
  - addl 16(%ebp),%ecx # ecx = ecx + Mem[16+ebp]
  - imull %ecx,%eax    # eax = eax * ecx

- Arithmetic operations that do NOT set condition codes
  - leal 4(%edx,%eax),%eax # eax = 4 + edx + eax

Quick Review

- x86-64 vs. IA32
  - Integer registers: 16 x 64-bit vs. 8 x 32-bit
  - movq, addq, ... vs. movl, addl, ...
    - movq -> “move quad word” or 4*16-bits
  - x86-64: better support for passing function arguments in registers

- Control
  - Condition code registers
    - Set as side effect or by cmp, test
    - Used:
      - Read out by setex instructions (setg, setle, ...)
      - Or by conditional jumps (jle .L4, je .L10, ...)
      - Or by conditional moves (cmovle %edx, %eax)
Quick Review

- **Do-While loop**
  - C Code
    ```c
    do
      Body
    while (Test);
    ```
  - Goto Version
    ```c
    loop:
      Body
      if (!Test)
        goto loop;
    ```
  
- **While-Do loop**
  - While version
    ```c
    while (Test)
      Body
    ```
  - Do-While Version
    ```c
    do
      Body
    while (!Test):
      done:
    ```
  
  **OR**
  ```c
  while (Test)
    Body
  ```

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**
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (see text)

- **Conditions in CISC**
  - CISC machines generally have condition code registers