The Hardware/Software Interface
CSE351 Winter 2013

x86 Programming II
Today’s Topics: control flow

- Condition codes
- Conditional and unconditional branches
- Loops
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; iterative) {...}

- 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>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>
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 (CF, ZF, SF, OF)

General purpose registers

- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp

Current stack top
- %esp
Current stack frame
- %ebp

Instruction pointer
- %eip

Condition codes
- CF
- ZF
- SF
- OF
Condition Codes (Implicit Setting)

- **Single-bit registers**
  - \( CF \)  Carry Flag (for unsigned)  \( SF \)  Sign Flag (for signed)
  - \( ZF \)  Zero Flag  \( OF \)  Overflow Flag (for signed)

- **Implicitly set (think of it as side effect) by arithmetic operations**
  - Example: \( \text{addl}/\text{addq} \ Src, Dest \leftrightarrow t = a+b \)
  - \( CF \) 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
    \( (a>0 \ \&\& \ b>0 \ \&\& \ t<0) \ || \ (a<0 \ \&\& \ b<0 \ \&\& \ t>=0) \)

- **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)
  - **ZF** Zero Flag
  - **SF** Sign Flag (for signed)
  - **OF** Overflow Flag (for signed)

- **Explicit Setting by Compare Instruction**
  - `cmp`/`cmpq` $Src2,Src1$
  - `cmp` $b,a$ 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
    - $(a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b>0 \land (a-b)>0)$
Condition Codes (Explicit Setting: Test)

- **Single-bit registers**
  - **CF** Carry Flag (for unsigned)
  - **ZF** Zero Flag
  - **SF** Sign Flag (for signed)
  - **OF** Overflow Flag (for signed)

- **Explicit Setting by Test instruction**
  - `testl/testq Src2,Src1`
  - `testl 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>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 to 0 or 1 based on combination of condition codes

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

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

**Body:** y at 12(%ebp), x at 8(%ebp)

```
movl 12(%ebp), %eax
cmpl %eax, 8(%ebp)
setg %al
movzbl %al, %eax
```

What does each of these instructions do?
Reading Condition Codes (Cont.)

- **SetX Instructions:**
  Set single byte to 0 or 1 based on combination of condition codes

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

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

**Body:** y at 12(%ebp), x at 8(%ebp)

- `movl 12(%ebp),%eax`  # eax = y
- `cmpl %eax,8(%ebp)`  # Compare x and y → (x − y)
- `setg %al`  # al = x > y
- `movzbl %al,%eax`  # Zero rest of %eax
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 absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
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
```

Body1
```
Setup

Finish
```

Body2
Conditional Branch Example (Cont.)

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

```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;
}
```

- 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;
}
```

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

**Variables in the Assembly Code:**
- `int x %edx`
- `int y %eax`
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;
}
```

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

int x %edx
int y %eax
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;
}
```

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

.L7:
    subl %edx, %eax
    jmp .L8

.L8:
    leave
    ret
```

```
int x %edx
int y %eax
```
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;
}
```

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

```c
int x %edx
int y %eax
```
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;
}
```

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

.L7:
    subl %edx, %eax
    movl %edx, %eax
    ret

.L8:
    leave
    ret
```

```
typedef:
    int x
    %edx
    int y
    %eax
```
**General Conditional Expression Translation**

**C Code**

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

**Goto Version**

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

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

```c
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 control flow)
- But overhead: both branches are evaluated
PC Relative Addressing

- PC relative branches are relocatable
- Absolute branches are not
## Compiling Loops

**C/Java code:**

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

**Machine code:**

```
loopTop:  cmpl  $0, %eax
            je   loopDone
            <loop body code>
            jmp  loopTop
loopDone:
```

- **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?**
“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
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx
    .L11:
        imull %edx,%eax
        decl %edx
        cmpl $1,%edx
        jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers:

- `%edx`: x
- `%eax`: result

Translation?
“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 8(%ebp),%edx

.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11

    movl %ebp,%esp
    popl %ebp
    ret
```

Registers:

%edx  x
%eax  result
General “Do-While” Translation

C Code

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

- **Body:**  
  ```c
  { 
    Statement_1; 
    Statement_2; 
    ... 
    Statement_n; 
  }
  ```

- **Test** returns integer
  - $0$ interpreted as false
  - $\neq 0$ interpreted as true

Goto Version

```
loop:
    Body
    if (Test)
    goto loop
```
“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
int fact_while(int x) {
    int result = 1;
    while (x > 1) {
        result *= x;
        x--;
    }
    return result;
}

# x in %edx, result in %eax
    jmp .L34       # goto Middle
.L35:           # Loop:
    imull %edx, %eax # result *= x
    decl %edx    # x--
.L34:           # Middle:
    cmpl $1, %edx # x:1
    jg .L35       # if >, goto Loop
“For” Loop Example: Square-and-Multiply

/* 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;
}

Algorithm

- Exploit bit representation: \( p = p_0 + 2p_1 + 2^2p_2 + \ldots + 2^{n-1}p_{n-1} \)
- Gives: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}^2)^2)^2)\ldots)^2 \)
  
    \[ z_i = 1 \text{ when } p_i = 0 \]

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

- 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 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>before 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=1010₂</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5= 10₁₂</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>81</td>
<td>2= 1₀₂</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6561</td>
<td>1= 1₂</td>
</tr>
<tr>
<td>5</td>
<td>59049</td>
<td>43046721</td>
<td>0₂</td>
</tr>
</tbody>
</table>
“For” Loop Example

```c
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
```

<table>
<thead>
<tr>
<th>Init</th>
<th>Test</th>
<th>Update</th>
<th>Body</th>
</tr>
</thead>
</table>
| result = 1 | p != 0    | p = p >> 1 | {
|            |           |            |   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 ;
}
```

Goto Version

```c
Init;
goto middle;
loop:
    Body
    Update ;
middle:
    if (Test)
        goto loop;
done:
```
For-Loop: Compilation

For Version

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

Goto Version

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

```
result = 1;
goto middle;
loop:
    if (p & 0x1)
        result *= x;
    x = x*x;
p = p >> 1;
middle:
    if (p != 0)
        goto loop;
done:
```
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
  - sall $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 setx 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
  if (!Test)
      goto done;
  do
      Body
  while (Test);
  done:
  ```

  **Goto Version**
  ```c
  if (!Test)
      goto done;
  loop:
      Body
      if (Test) goto loop;
  done:
  goto middle;
  loop:
      Body
      middle:
      if (Test) goto loop;
  ```
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