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)**

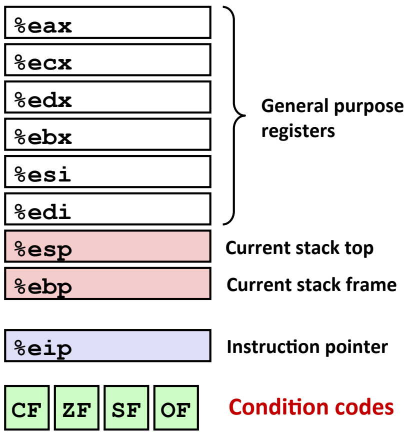
Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

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

Current stack top

Current stack frame

Instruction pointer

Condition codes

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: `addl/addq` *Src,Dest* ↔ `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)	`SF`	Sign Flag (for signed)
`ZF`	Zero Flag	`OF`	Overflow Flag (for signed)

- **Explicit Setting by Compare Instruction**

  `cmpl/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` (as signed)
  - OF set if two's complement (signed) overflow
    `(a>0 && b<0 && (a-b)<0) || (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)
`ZF`	Zero Flag	`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

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

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

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

`%eax`	`%ah`	`%al`
`%ecx`	`%ch`	`%cl`
`%edx`	`%dh`	`%dl`
`%ebx`	`%bh`	`%bl`
`%esi`		
`%edi`		
`%esp`		
`%ebp`		

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

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

`%eax`	`%ah`	`%al`
`%ecx`	`%ch`	`%cl`
`%edx`	`%dh`	`%dl`
`%ebx`	`%bh`	`%bl`
`%esi`		
`%edi`		
`%esp`		
`%ebp`		

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

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Conditional Branch Example

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

```
absdiff:
   pushl  %ebp             } Setup
   movl   %esp, %ebp
   movl   8(%ebp), %edx
   movl   12(%ebp), %eax
   cmpl   %eax, %edx       } Body1
   jle    .L7
   subl   %eax, %edx
   movl   %edx, %eax
.L8:
   leave                   } Finish
   ret
.L7:
   subl   %edx, %eax       } Body2
   jmp    .L8
```

Conditional Branch Example (Cont.)

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

```
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.)

```
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
```

```
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
```

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;
}
```

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

```
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
```

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;
}
```

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

```
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
```

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;
}
```

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

```
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
```

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;
}
```

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

```
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
```

General Conditional Expression Translation

C Code

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

```
if (Test)
  val = Then-Expr;
else
  val = Else-Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  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?

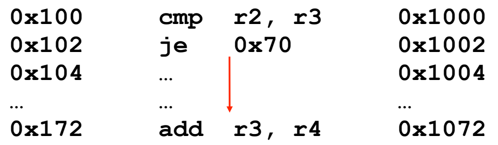
Conditionals: x86-64

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

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax  # eax = x
  movl   %esi, %edx  # edx = y
  subl   %esi, %eax  # eax = x-y
  subl   %edi, %edx  # edx = y-x
  cmpl   %esi, %edi  # x:y
  cmovle %edx, %eax  # eax=edx if <=
  ret
```

- **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



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

- **PC relative branches are <u>relocatable</u>**
- **Absolute branches are not**

Compiling Loops

C/Java code:

```
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

```
int fact_do(int x)
{
  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 Compilation

Registers:

```
%edx     x
%eax     result
```

Goto Version

```
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
```

Translation?

“Do-While” Loop Compilation

Registers:

```
%edx     x
%eax     result
```

Goto Version

```
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
```

General “Do-While” Translation

C Code

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

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- ***Body:*** {
  - *Statement*$_1$;
  - *Statement*$_2$;
  - …
  - *Statement*$_n$;

  }

- *Test* returns integer

  = 0 interpreted as false

  ≠ 0 interpreted as true

“While” Loop Translation

C Code

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

Goto Version

```
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

```
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: $p = p_0 + 2p_1 + 2^2 p_2 + \ldots\ 2^{n-1} p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot \underbrace{(\ldots((z_{n-1}{}^2)^2)\ldots)^2}_{n\text{–}1 \text{ times}}$

    $z_i = 1$ when $p_i = 0$

    $z_i = x$ when $p_i = 1$
  - Complexity $O(\log p)$

Example

$3^{10} = 3^2 * 3^8$

$= 3^2 * ((3^2)^2)^2$

`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;
}
```

before iteration	result	x=3	p=10
1	1	3	10=1010$_2$
2	1	9	5= 101$_2$
3	9	81	2= 10$_2$
4	9	6561	1= 1$_2$
5	59049	43046721	0$_2$

"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

```
for  (Init;  Test;  Update )

      Body
```

While Version

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

Goto Version

```
   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:
```

```
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

```
  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

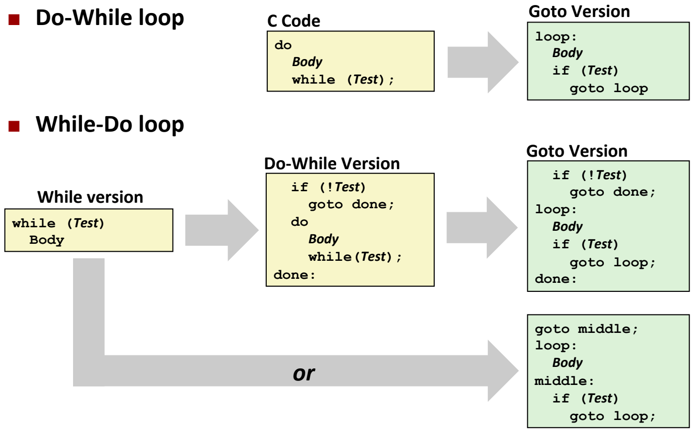
%rax	%eax	%r8	%r8d
%rbx	%edx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%ebx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- **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`**)

CF ZF SF OF

Quick Review

- Do-While loop
- While-Do loop



C Code

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

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

While version

```
while (Test)
  Body
```

Do-While Version

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

Goto Version

```
  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