

Machine-Level Programming II: Arithmetic & Control

Lecture 4, March 10, 2011
Alexandre David

*Credits to Randy Bryant & Dave O'Hallaron
from Carnegie Mellon*

Today

- **Complete addressing mode, address computation (leal)**
- Arithmetic operations
- Control: Condition codes
- Conditional branches
- While loops

Complete Memory Addressing Modes

■ Most General Form

■ $D(Rb, Ri, S)$ $Mem[Reg[Rb]+S*Reg[Ri]+ D]$

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 8 integer registers
- Ri: Index register: Any, except for `%esp`
 - Unlikely you’d use `%ebp`, either
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

■ Special Cases

■ (Rb, Ri) $Mem[Reg[Rb]+Reg[Ri]]$

■ $D(Rb, Ri)$ $Mem[Reg[Rb]+Reg[Ri]+D]$

■ (Rb, Ri, S) $Mem[Reg[Rb]+S*Reg[Ri]]$

Address Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8(%edx)</code>		
<code>(%edx,%ecx)</code>		
<code>(%edx,%ecx,4)</code>		
<code>0x80(,%edx,2)</code>		

Address Computation Examples

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8(%edx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%edx,%ecx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

Address Computation Instruction

■ `leal Src, Dest`

- *Src* is address mode expression
- Set *Dest* to address denoted by expression

■ Uses

- Computing addresses without a memory reference
 - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form $x + k*y$
 - $k = 1, 2, 4, \text{ or } 8$

■ Example

```
int mul12(int x)
{
    return x*12;
}
```

Converted to ASM by compiler:

```
leal (%eax,%eax,2), %eax    ;t <- x+x*2
sall $2, %eax              ;return t<<2
```

Today

- Complete addressing mode, address computation (leal)
- **Arithmetic operations**
- Control: Condition codes
- Conditional branches
- While loops

Some Arithmetic Operations

■ Two Operand Instructions:

<i>Format</i>	<i>Computation</i>	
<code>addl</code>	<code>Src, Dest</code>	<code>Dest = Dest + Src</code>
<code>subl</code>	<code>Src, Dest</code>	<code>Dest = Dest - Src</code>
<code>imull</code>	<code>Src, Dest</code>	<code>Dest = Dest * Src</code>
<code>sall</code>	<code>Src, Dest</code>	<code>Dest = Dest << Src</code> <i>Also called shll</i>
<code>sarl</code>	<code>Src, Dest</code>	<code>Dest = Dest >> Src</code> <i>Arithmetic</i>
<code>shrl</code>	<code>Src, Dest</code>	<code>Dest = Dest >> Src</code> <i>Logical</i>
<code>xorl</code>	<code>Src, Dest</code>	<code>Dest = Dest ^ Src</code>
<code>andl</code>	<code>Src, Dest</code>	<code>Dest = Dest & Src</code>
<code>orl</code>	<code>Src, Dest</code>	<code>Dest = Dest Src</code>

■ Watch out for argument order!

■ No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

■ One Operand Instructions

`incl` *Dest* $Dest = Dest + 1$

`decl` *Dest* $Dest = Dest - 1$

`negl` *Dest* $Dest = - Dest$

`notl` *Dest* $Dest = \sim Dest$

■ See book for more instructions

Arithmetic Expression Example

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

arith:

```
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %ecx
    movl    12(%ebp), %edx
    leal    (%edx,%edx,2), %eax
    sall    $4, %eax
    leal    4(%ecx,%eax), %eax
    addl    %ecx, %edx
    addl    16(%ebp), %edx
    imull   %edx, %eax
    popl    %ebp
    ret
```

} Set Up

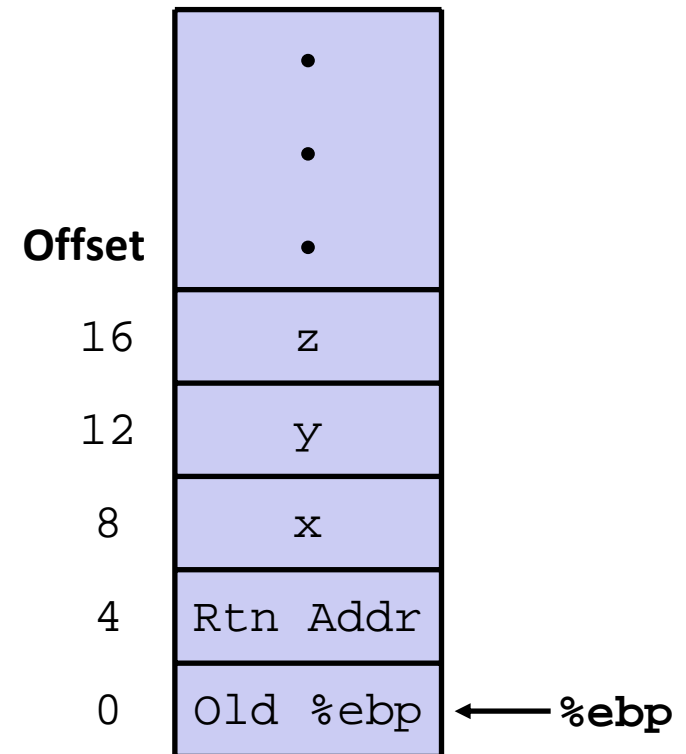
} Body

} Finish

Understanding arith

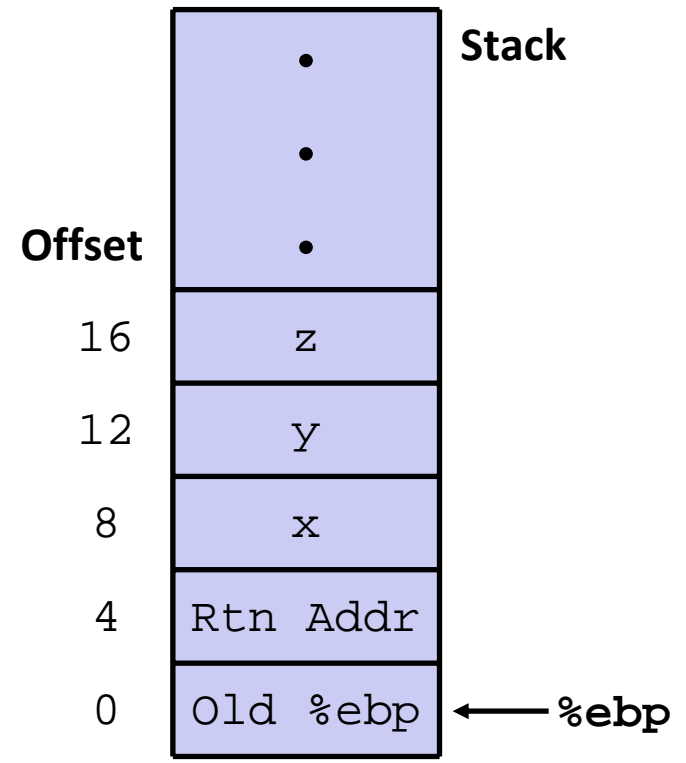
```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
movl    8(%ebp), %ecx
movl    12(%ebp), %edx
leal    (%edx,%edx,2), %eax
sall    $4, %eax
leal    4(%ecx,%eax), %eax
addl    %ecx, %edx
addl    16(%ebp), %edx
imull   %edx, %eax
```



Understanding arith

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```



```
movl    8(%ebp), %ecx      # ecx = x
movl    12(%ebp), %edx     # edx = y
leal    (%edx,%edx,2), %eax # eax = y*3
sall    $4, %eax          # eax *= 16 (t4)
leal    4(%ecx,%eax), %eax # eax = t4 +x+4 (t5)
addl    %ecx, %edx        # edx = x+y (t1)
addl    16(%ebp), %edx    # edx += z (t2)
imull   %edx, %eax        # eax = t2 * t5 (rval)
```

Observations about `arith`

```
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
 - $(x+y+z) * (x+4+48*y)$

```
movl    8(%ebp), %ecx      # ecx = x
movl    12(%ebp), %edx     # edx = y
leal    (%edx,%edx,2), %eax # eax = y*3
sall    $4, %eax          # eax *= 16 (t4)
leal    4(%ecx,%eax), %eax # eax = t4 +x+4 (t5)
addl    %ecx, %edx        # edx = x+y (t1)
addl    16(%ebp), %edx    # edx += z (t2)
imull   %edx, %eax        # eax = t2 * t5 (rval)
```

Another Example

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

logical:

```
    pushl %ebp
    movl %esp,%ebp
} Set Up

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
} Body

    popl %ebp
    ret
} Finish
```

```
    movl 12(%ebp),%eax    # eax = y
    xorl 8(%ebp),%eax    # eax = x^y      (t1)
    sarl $17,%eax        # eax = t1>>17   (t2)
    andl $8185,%eax      # eax = t2 & mask (rval)
```

Another Example

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

logical:

```
    pushl %ebp
    movl %esp,%ebp
```

} Set Up

```
    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
```

} Body

```
    popl %ebp
    ret
```

} Finish

<code>movl 12(%ebp),%eax</code>	<code># eax = y</code>
<code>xorl 8(%ebp),%eax</code>	<code># eax = x^y (t1)</code>
<code>sarl \$17,%eax</code>	<code># eax = t1>>17 (t2)</code>
<code>andl \$8185,%eax</code>	<code># eax = t2 & mask (rval)</code>

Another Example

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

logical:

```
    pushl %ebp
    movl %esp,%ebp
} Set Up

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
} Body

    popl %ebp
    ret
} Finish
```

```
movl 12(%ebp),%eax    # eax = y
xorl 8(%ebp),%eax    # eax = x^y      (t1)
sarl $17,%eax        # eax = t1>>17    (t2)
andl $8185,%eax      # eax = t2 & mask (rval)
```


Another Example

```
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

logical:

```
    pushl %ebp
    movl %esp,%ebp
} Set Up

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax
} Body

    popl %ebp
    ret
} Finish
```

<code>movl 12(%ebp),%eax</code>	<code># eax = y</code>
<code>xorl 8(%ebp),%eax</code>	<code># eax = x^y (t1)</code>
<code>sarl \$17,%eax</code>	<code># eax = t1>>17 (t2)</code>
<code>andl \$8185,%eax</code>	<code># eax = t2 & mask (rval)</code>

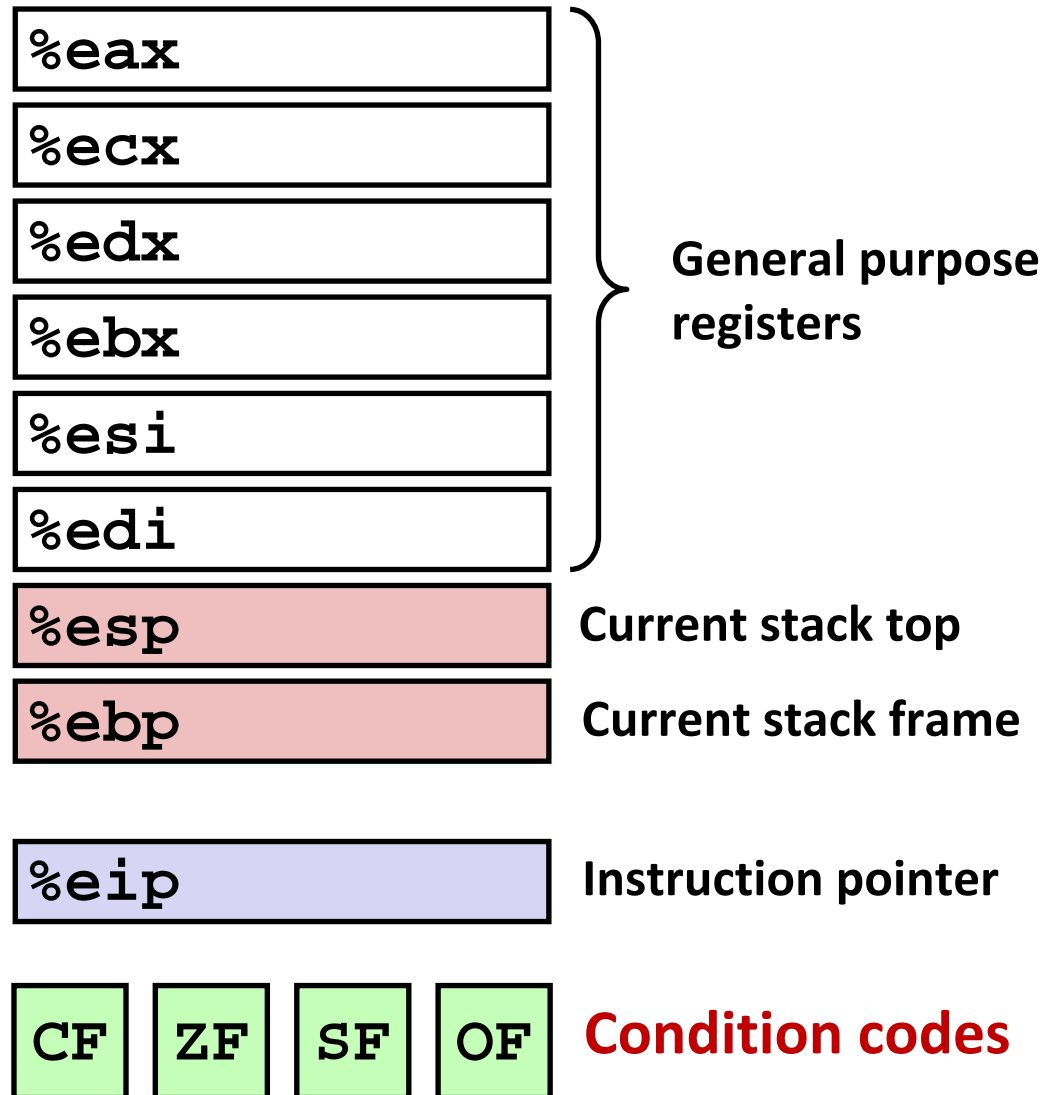
Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- **Control: Condition codes**
- Conditional branches
- Loops

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`)



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` \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 `leaq` instruction

■ [Full documentation \(IA32\)](#), link on course website

Condition Codes (Explicit Setting: Compare)

■ 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)

■ 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** when `a&b == 0`

- **SF set** when `a&b < 0`

Reading Condition Codes

■ SetX Instructions

- Set single byte based on combinations of condition codes

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	\sim ZF	Not Equal / Not Zero
sets	SF	Negative
setns	\sim SF	Nonnegative
setg	$\sim (SF \wedge OF) \& \sim ZF$	Greater (Signed)
setge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
setl	$(SF \wedge OF)$	Less (Signed)
setle	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
seta	$\sim CF \& \sim ZF$	Above (unsigned)
setb	CF	Below (unsigned)

Use?

Reading Condition Codes (Cont.)

■ SetX Instructions:

- Set single byte 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;
}
```

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
```

%eax	%ah	%al
------	-----	-----

%ecx	%ch	%cl
------	-----	-----

%edx	%dh	%dl
------	-----	-----

%ebx	%bh	%bl
------	-----	-----

%esi

%edi

%esp

%ebp

Reading Condition Codes: x86-64

■ SetX Instructions:

- Set single byte based on combination of condition codes
- Does not alter remaining 3 bytes

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

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

Bodies

```
cmpl %esi, %edi
setg %al
movzbl %al, %eax
```

```
cmpq %rsi, %rdi
setg %al
movzbl %al, %eax
```

Is `%rax` zero?

Yes: 32-bit instructions set high order 32 bits to 0!

Note: Let the compiler have fun for you.

Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- **Conditional branches & Moves**
- Loops

Jumping

■ jX Instructions

- Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \& \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
j1	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
ja	$\sim CF \& \sim ZF$	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
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L6
    subl   %eax, %edx
    movl   %edx, %eax
    jmp    .L7
.L6:
    subl   %edx, %eax
.L7:
    popl   %ebp
    ret
```

Setup

Body1

Body2a

Body2b

Finish

Conditional Branch Example (Cont.)

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

- C allows “goto” as means of transferring control
 - Closer to machine-level programming style
- This is a very bad coding style – don’t use it in practice!

```
absdiff:
    pushl   %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L6
    subl   %eax, %edx
    movl   %edx, %eax
    jmp    .L7
.L6:
    subl   %edx, %eax
.L7:
    popl   %ebp
    ret
```

Setup

Body1

Body2a

Body2b

Finish

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle   .L6
    subl   %eax, %edx
    movl   %edx, %eax
    jmp   .L7
.L6:
    subl   %edx, %eax
.L7:
    popl   %ebp
    ret
```

Setup

Body1

Body2a

Body2b

Finish

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L6
    subl   %eax, %edx
    movl   %edx, %eax
    jmp    .L7
.L6:
    subl   %edx, %eax
.L7:
    popl   %ebp
    ret
```

Setup

Body1

Body2a

Body2b

Finish

Conditional Branch Example (Cont.)

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

```
absdiff:
    pushl   %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L6
    subl   %eax, %edx
    movl   %edx, %eax
    jmp    .L7
.L6:
    subl   %edx, %eax
.L7:
    popl   %ebp
    ret
```

Setup

Body1

Body2a

Body2b

Finish

General Conditional Expression Translation

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

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

Goto Version

```
nt = !Test;  
if (nt) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
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

Using Conditional Moves

■ Conditional Move Instructions

- Instruction supports:
if (Test) Dest ← Src
- Supported in post-1995 x86 processors
- GCC does not always use them
 - Wants to preserve compatibility with ancient processors
 - Enabled for x86-64
 - Use switch `-march=686` for IA32

■ Why?

- Branches are very disruptive to instruction flow through pipelines
- Conditional move do not require control transfer

C Code

```
val = Test  
    ? Then_Expr  
    : Else_Expr;
```

Goto Version

```
tval = Then_Expr;  
result = Else_Expr;  
t = Test;  
if (t) result = tval;  
return result;
```

Conditional Move Example: x86-64

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

x in %edi

y in %esi

absdiff:

```
movl    %edi, %edx  
subl    %esi, %edx    # tval = x-y  
movl    %esi, %eax  
subl    %edi, %eax    # result = y-x  
cmpl    %esi, %edi    # Compare x:y  
cmovg   %edx, %eax    # If >, result = tval  
ret
```

Aside: cmov

- **cmov is available under IA32, but not all generations.**
 - Compilers are conservative by default.
 - You can generate code with cmov with the right options.
- **In 64-bit mode: all CPUs supporting 64-bit support cmov as well!**

Bad Cases for Conditional Move – Compiler's Job

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects - **crash**

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches and moves
- **Loops – self read summarized**

Summary

■ Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- Control: Condition codes
- Conditional branches & conditional moves
- Loops

■ Next Time

- Switch statements
- Stack
- Call / return
- Procedure call discipline

Lessons

■ Specific to x86

- Addressing modes
- Operands and order

■ General concepts

- Different addressing modes
- State flags
- Arithmetic instructions modify state flags
- Special test/cmp instructions modify state flags
- Conditional/unconditional jumps
- Pattern to transform high-level constructs into low-level instructions