

CS 33

Machine Programming (2)

All of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.

Today

- **Arithmetic operations**
- Control: condition codes
- Conditional branches & moves

Supplied by CMU.

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?)

Supplied by CMU.

Some Arithmetic Operations

- **One-operand Instructions**

`incl` $Dest = Dest + 1$

`decl` $Dest = Dest - 1$

`negl` $Dest = -Dest$

`notl` $Dest = \sim Dest$

- **See book for more instructions**

Supplied by CMU.

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
```

} Set
Up

```
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
```

} Body

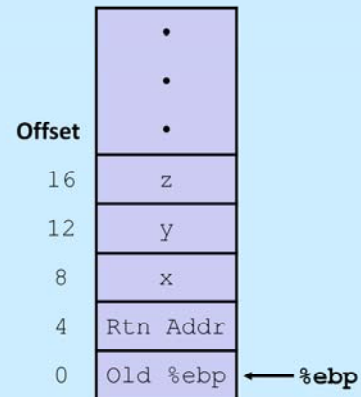
```
popl  %ebp
ret
```

} 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;
}
```

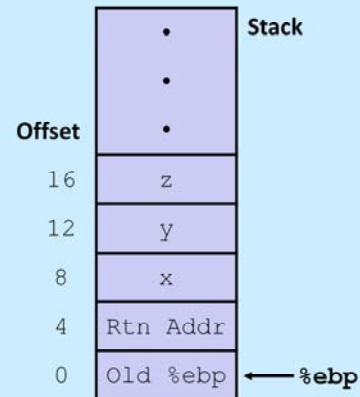
```
movl    8(%ebp), %ecx
movl    12(%ebp), %edx
leal    (%edx,%edx,2), %eax
sall    $4, %eax
leal    4(%ecx,%eax), %eax
addl    %ecx, %edx
addl    16(%ebp), %edx
imull   %edx, %eax
```



Supplied by CMU.

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)
```

Supplied by CMU.

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:
 $(x+y+z) * (x+4+48*y)$

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

Supplied by CMU.

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>

Supplied by CMU.

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>

Supplied by CMU.

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)

Supplied by CMU.

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>

Supplied by CMU.

Today

- Arithmetic operations
- **Control: condition codes**
- Conditional branches & moves

Supplied by CMU.

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`)

`%eax`

`%ecx`

`%edx`

`%ebx`

`%esi`

`%edi`

General purpose registers

`%esp`

Current stack top

`%ebp`

Current stack frame

`%eip`

Instruction pointer

`CF`

`ZF`

`SF`

`OF`

Condition codes

Supplied by CMU.

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 `leal` instruction**

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
 $(a > 0 \ \&\& \ b < 0 \ \&\& \ (a-b) < 0) \ || \ (a < 0 \ \&\& \ b > 0 \ \&\& \ (a-b) > 0)$

Supplied by CMU.

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`

Supplied by CMU.

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)

Supplied by CMU.

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!

Supplied by CMU.

Today

- Arithmetic operations
- x86-64
- Control: condition codes
- **Conditional branches & moves**

Supplied by CMU.

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)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \ \ ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

Supplied by CMU.

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
- Generally considered bad coding style

```
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

Supplied by CMU.

Using Conditional Moves

- **Conditional move instructions**

- instruction supports:
if (Test) Dest \leftarrow 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 moves 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;  
}
```

	absdiff:
x in %edi	movl %edi, %edx
y in %esi	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

Supplied by CMU.

Bad Cases for Conditional Move

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

Computations with side effects

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

- both values get computed
- must be side-effect free

Supplied by CMU.