

Machine Programming II: Instructions (cont'd)

- Move instructions, registers, and operands
- Complete addressing mode, address computation (`leal`)
- Arithmetic operations (including some x86-64 instructions)
- Condition codes
- Control, unconditional and conditional branches
- While loops

Data Representations: IA32 + x86-64

■ Sizes of C Objects (in Bytes)

<i>C Data Type</i>	<i>Typical 32-bit</i>	<i>Intel IA32</i>	<i>x86-64</i>
▪ unsigned	4	4	4
▪ int	4	4	4
▪ long int	4	4	8
▪ char	1	1	1
▪ short	2	2	2
▪ float	4	4	4
▪ double	8	8	8
▪ long double	8	10/12	16
▪ char *	4	4	8

Or any other pointer

x86-64 Integer Registers

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

- Extend existing registers. Add 8 new ones.
- Make `%ebp/%rbp` general purpose

Instructions

- Long word `l` (4 Bytes) \leftrightarrow Quad word `q` (8 Bytes)
- New instructions:
 - `movl` \rightarrow `movq`
 - `addl` \rightarrow `addq`
 - `sall` \rightarrow `salq`
 - etc.
- 32-bit instructions that generate 32-bit results
 - Set higher order bits of destination register to 0
 - Example: `addl`

Swap in 32-bit Mode

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
                                     } Setup

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
                                     } Body

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
                                     } Finish
```

Swap in 64-bit Mode

```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movl  (%rdi), %edx
    movl  (%rsi), %eax
    movl  %eax, (%rdi)
    movl  %edx, (%rsi)
    retq
```

- **Operands passed in registers (why is this useful?)**
 - First (**xp**) in **%rdi**, second (**yp**) in **%rsi**
 - 64-bit pointers
- **No stack operations required**
- **32-bit data**
 - Data held in registers **%eax** and **%edx**
 - **movl** operation

Swap Long Ints in 64-bit Mode

```
void swap_l
(long int *xp, long int *yp)
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap_l:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    retq
```

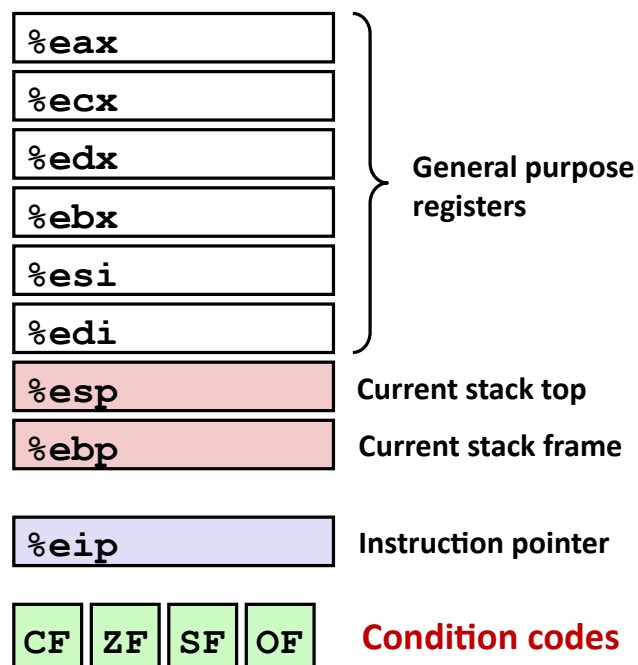
■ 64-bit data

- Data held in registers `%rax` and `%rdx`
- `movq` operation
- “q” stands for quad-word

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 `leal` instruction (beware!)

■ Full documentation (IA32)

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

Reading Condition Codes

■ SetX Instructions

- Set a single byte based on combinations of condition codes

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

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;
}
```

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

Body

```
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 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

```
movl 12(%ebp), %eax    # eax = y
cmpl %eax, 8(%ebp)    # Compare x and y
setg %al               # al = x > y
movzbl %al, %eax      # Zero rest of %eax
```

Note inverted ordering!

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
    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

```

Setup
 Body1
 Finish
 Body2

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;
}
```

- 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 .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;
}
```

```
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;
}
```

```
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;
}
```

```
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;
}
```

```
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;
```

```
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

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
    movl   %esi, %edx
    subl   %esi, %eax
    subl   %edi, %edx
    cmpl   %esi, %edi
    cmovle %edx, %eax
    ret
```

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

General Form with Conditional Move

C Code

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

Conditional Move Version

```
val1 = Then-Expr;  
val2 = Else-Expr;  
val1 = val2 if !Test;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn't hold
- **Don't use when:**
 - Then or else expression have side effects
 - Then and else expression are too expensive