Machine Programming II: Instructions (cont’d)

- Move instructions, registers, and operands
- Complete addressing mode, address computation (lea)
- 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)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

*Or any other pointer*
x86-64 Integer Registers

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

Instructions

- Long word \( 1 \) (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

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

```
swap:
pushl %ebp
    movl %esp,%ebp
pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
movl %ebp,%esp
    popl %ebp
ret
```

Swap in 64-bit Mode

```c
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

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

- **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)

- **General purpose registers**
  - %eax
  - %ecx
  - %edx
  - %ebx
  - %esi
  - %edi

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

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

- **Not set by lea instruction (beware!)**

- **Full documentation** (IA32)

---

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - \( \text{cmpl/cmpq~Src2,Src1} \)
  - \( \text{cmpl b,a} \) 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
    - \( (a>0 \&\& b<0 \&\& (a-b)<0) \) \( \mid \mid \) \( (a<0 \&\& b>0 \&\& (a-b)>0) \)
Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  
  \[
  \text{testl/testq } \text{Src2,Src1} \\
  \text{testl b,a } \text{like computing } a\&b \text{ without setting destination}
  \]

  - Sets condition codes based on value of \text{Src1} & \text{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

<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>setsns</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>setle</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 based on combination of condition codes

- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use movzb1 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 and y
setg %al            # al = x > y
movzb1 %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;
}
```

```assembly
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.)

```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;
}
```

Conditional Branch Example (Cont.)

```assembly
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.)

```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
    movl %edx, %eax
.L8:
    leave
    ret
.L7:
    subl %edx, %eax
    jmp .L8
```

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
Conditionals: x86-64

```c
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

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

Conditional Move Version

```c
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