The Hardware/Software Interface
CSE351 Winter 2013

x86 Programming I

Today

- Move instructions, registers, and operands
- Memory addressing modes
- swap example: 32-bit vs. 64-bit
- Arithmetic operations

Why learn assembly language?

- Not to be able to write programs directly in assembly
  - Compilers do that for you
- But to be able to understand the code generated by compilers, so that you can then:
  - Optimize performance of critical sections of code
  - Investigate unexpected or even buggy behavior
  - Understand how security vulnerabilities arise, and how to protect against them
Three Basic Kinds of Instructions

- Transfer data between memory and register
  - Load data from memory into register
    - %reg = Mem[address]
  - Store register data into memory
    - Mem[address] = %reg
  - \( c = a + b; \)

- Perform arithmetic function on register or memory data

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Moving Data: IA32

- Moving Data
  - movx Source, Dest
  - x is one of \{b, w, l\}
  - movl Source, Dest: Move 4-byte “long word”
  - movw Source, Dest: Move 2-byte “word”
  - movb Source, Dest: Move 1-byte “byte”

- Lots of these in typical code

Moving Data: IA32

- Moving Data
  - movl Source, Dest:

- Operand Types
  - Immediate: Constant integer data
    - Example: \$0x400, \$-533
    - Like C constant, but prefixed with \'$\'
    - Encoded with 1, 2, or 4 bytes
  - Register: One of 8 integer registers
    - Example: %eax, %edx
    - But %esp and %ebp reserved for special use
    - Others have special uses for particular instructions
  - Memory: 4 consecutive bytes of memory at address given by register
    - Simplest example: \((%eax)\)
    - Various other “address modes”
**movl Operand Combinations**

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td>movl $0x4, %eax</td>
<td>var_a = 0x4;</td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td>movl $-147, (%eax)</td>
<td>*p_a = -147;</td>
<td></td>
</tr>
<tr>
<td>movl</td>
<td>Reg</td>
<td>(%eax),%edx</td>
<td>var_d = var_a;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>*p_d = var_a;</td>
</tr>
</tbody>
</table>

*Cannot do memory-memory transfer with a single instruction.*

**Memory vs. registers**

- Why both?

**Memory Addressing Modes: Basic**

- **Indirect (R)** Mem[Reg[R]]
  - Register R specifies the memory address
    - `movl (%ecx), %eax`

- **Displacement D(R)** Mem[Reg[R]+D]
  - Register R specifies a memory address
    - (e.g. the start of some memory region)
  - Constant displacement D specifies the offset from that address
    - `movl 8(%ebp), %edx`

**Using Basic Addressing Modes**

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

- **Set Up**
  - `movl %ebp
    movl %esp,%ebp
    pushl %ebx`

- **Body**
  - `movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax, (%edx)
    movl %ebx, (%ecx)`

- **Finish**
  - `movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret`
Understanding Swap

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

Stack (in memory)

<table>
<thead>
<tr>
<th>Offset</th>
<th>YP</th>
<th>XP</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
<td>%ebp</td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
<td>%ebx</td>
</tr>
</tbody>
</table>

Register Value

<table>
<thead>
<tr>
<th>%ecx</th>
<th>YP</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>XP</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
</tr>
</tbody>
</table>

movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx  # edx = xp
movl (%ecx),%eax  # eax = *yp (t1)
movl (%edx),%ebx  # ebx = *xp (t0)
movl %eax,(%edx)  # *xp = eax
movl %ebx,(%ecx)  # *yp = ebx
x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Extend existing registers, and add 8 new ones; all accessible as 8, 16, 32, 64 bits.

32-bit vs. 64-bit operands

- Long word 1 (4 Bytes) ↔ Quad word q (8 Bytes)

- New instruction forms:
  - `movl` ↔ `movq`
  - `addl` ↔ `addq`
  - `sall` ↔ `salq`
  - etc.

- x86-64 can still use 32-bit instructions that generate 32-bit results
  - Higher-order bits of destination register are just set to 0
  - Example: `addl`

Swap Ints in 32-bit Mode

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

Swap In 64-bit Mode

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

64-bit data
- Data held in registers `%rax` and `%rdx`
- `movq` operation
- "q" stands for quad-word

Complete Memory Addressing Modes

- Remember, the addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
- Most General Form:
  
  \[ D(Rb,Ri,S) \quad Mem[Reg[Rb]+S*Reg[Ri]+D] \]

  - \(D\): Constant "displacement" 1, 2, or 4 bytes
  - \(Rb\): Base register: Any of the 8/16 integer registers
  - \(Ri\): Index register: Any, except for `%esp` or `%ebp`
  - Unlikely you’d use `%ebp`, either
  - \(S\): Scale: 1, 2, 4, or 8 (\(why\ these\ numbers?\))
- Special Cases: can use any combination of \(D, Rb, Ri\) and \(S\)

\[ D(Rb,Ri,S) \quad Mem[Reg[Rb]+Reg[Ri]] \]
\[ D(Rb,Ri) \quad Mem[Reg[Rb]+Reg[Ri]+D] \]

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0x8(%edx))</td>
<td>(0xf000 + 0x8)</td>
<td>(0xf008)</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>(0xf000 + 0x100)</td>
<td>(0xf100)</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>(0xf000 + 4*0x100)</td>
<td>(0xf000)</td>
</tr>
<tr>
<td>(0x80(,%edx,2))</td>
<td>(2*0xf000 + 0x80)</td>
<td>(0x1e080)</td>
</tr>
</tbody>
</table>

Address Computation Instruction

- **leal** \(Src,\ Dest\)
  - \(Src\) is address mode expression
  - Set \(Dest\) to address computed by expression
    - (\(leal\) stands for \textit{load effective address})
  - Example: \(leal (\%edx,\%ecx,4), \%eax\)

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of \(p = 6*i\):
  - Computing arithmetic expressions of the form \(x + k*i\)
    - \(k = 1, 2, 4,\ or \ 8\)
Some Arithmetic Operations

- Two Operand (Binary) Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imull</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>sal1</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrl</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>xorl</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- One Operand (Unary) Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decl</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negl</td>
<td>Dest = -Dest</td>
</tr>
<tr>
<td>notl</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>

- See textbook section 3.5.5 for more instructions: mull, cltd, idivl, divl

Watch out for argument order! (especially subl)

No distinction between signed and unsigned int (why?)

Using leal for Arithmetic Expressions

```c
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;
}
```

Understanding arith

```asm
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;
}
```
Understanding arith

```c
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 %ebp, %eax  # eax = x
movl 12(%ebp), %edx  # edx = y
leal (%edx,%eax), %ecx  # ecx = x+y (t1)
leal (%edx,%edx,2), %edx  # edx = y + 2*y = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp), %ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax), %eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
```

Observations about arith

```c
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 %ebp, %eax  # eax = x
movl 12(%ebp), %edx  # edx = y
leal (%edx,%eax), %ecx  # ecx = x+y (t1)
leal (%edx,%edx,2), %edx  # edx = y + 2*y = 3*y
sall $4,%edx  # edx = 48*y (t4)
addl 16(%ebp), %ecx  # ecx = z+t1 (t2)
leal 4(%edx,%eax), %eax  # eax = 4+t4+x (t5)
imull %ecx,%eax  # eax = t5*t2 (rval)
```

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

```c
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;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl 17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Another Example

```c
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;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl 17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Another Example

```c
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;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl 17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Another Example

```c
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;
}
```

```assembly
logical:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl 17,%eax
    andl $8185,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```