Assembly I: Basic Operations

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Moving Data (1)

- **Moving data: movl source, dest**
  - Move 4-byte ("long") word
  - Lots of these in typical code

- **Operand types**
  - Immediate: constant integer data
    - Like C constant, but prefixed with ‘$’
    - e.g. $0x400, $-533
    - Encoded with 1, 2, or 4 bytes
  - Register: one of 8 integer registers
    - But %esp and %ebp reserved for special use
    - Others have special uses for particular instructions
  - Memory: 4 consecutive bytes of memory
    - Various "addressing modes"
Moving Data (2)

- **movl operand combinations**
  - Cannot do memory-memory transfers with single instruction

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>movl</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imm</td>
<td><strong>Reg</strong></td>
<td>movl $0x4,%eax</td>
</tr>
<tr>
<td>Mem</td>
<td><strong>Reg</strong></td>
<td>movl $-147,(%eax)</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movl %eax,%edx</td>
</tr>
<tr>
<td>Mem</td>
<td><strong>Reg</strong></td>
<td>movl %eax,(%edx)</td>
</tr>
<tr>
<td>Mem</td>
<td><strong>Reg</strong></td>
<td>movl (%eax),%edx</td>
</tr>
</tbody>
</table>
Simple Addressing Modes

- **Normal** \( (R) \) \( \text{Mem}[	ext{Reg}[R]] \)
  - Register \( R \) specifies memory address
  - e.g., `movl (%ecx), %eax`

- **Displacement** \( D(R) \) \( \text{Mem}[	ext{Reg}[R]+D] \)
  - Register \( R \) specifies start of memory region
  - Constant displacement \( D \) specifies offset
  - e.g., `movl 8(%ebp), %edx`
Indexed Addressing Modes (1)

- **Most general form:**
  \[ D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+D] \]
  - \( D \): constant “displacement”: 1, 2, or 4 bytes
  - \( Rb \): Base register: any of 8 integer registers
  - \( Ri \): Index register: any, except for \%esp & \%ebp
  - \( S \): Scale: 1, 2, 4, or 8

- **Special cases**
  - \( (Rb,Ri) \) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]]
  - \( D(Rb,Ri) \) \quad \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D]
  - \( (Rb,Ri,S) \) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]]
  - Useful to access arrays and structures
Indexed Addressing Modes (2)

- Address computation example

<table>
<thead>
<tr>
<th>Expression</th>
<th>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>0xf400</td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
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
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

Register Allocation (By compiler)

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<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>

Stack Offset

-4 Old %ebp
  0 Old %ebx
  4 Rtn adr
  8 xp
  12 yp

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
Understanding Swap (2)

Register Allocation (By compiler)

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<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>

Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x124</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</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
### Understanding Swap (3)

**Register Allocation (By compiler):**

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<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>

**Address: **

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0x100</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td>4</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x124</td>
</tr>
<tr>
<td>0</td>
<td>0x124</td>
</tr>
<tr>
<td>12</td>
<td>0x120</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
### Understanding Swap (4)

#### Register Allocation (By compiler)

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<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>

#### Address Table

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>0x120</td>
<td>0x120</td>
</tr>
<tr>
<td>xp</td>
<td>0x124</td>
<td>0x124</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x100</td>
<td>0x100</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
<td>0x110</td>
</tr>
<tr>
<td></td>
<td>0x114</td>
<td>0x114</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
<td>0x118</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
<td>0x11c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x124</td>
</tr>
</tbody>
</table>

#### Instructions

- `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,%ebx` # *xp = eax
- `movl %ebx,(%ecx)` # *yp = ebx

---

**SSE2030: Introduction to Computer Systems | Fall 2010 | Jin-Soo Kim (jinsookim@skku.edu)**

Page 11
Understanding Swap (5)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Variable</th>
<th>Address</th>
<th>Register Allocation (By compiler)</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td></td>
<td>0x120</td>
<td>%ecx  0x120</td>
</tr>
<tr>
<td>xp</td>
<td></td>
<td>0x124</td>
<td>%edx  0x124</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x114</td>
<td></td>
<td>%ecx, %edi</td>
</tr>
<tr>
<td></td>
<td>0x118</td>
<td></td>
<td>%esi</td>
</tr>
<tr>
<td></td>
<td>0x11c</td>
<td></td>
<td>%ebp</td>
</tr>
<tr>
<td></td>
<td>0x124</td>
<td></td>
<td>%ebx, %esi</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>%eax, %edi</td>
</tr>
<tr>
<td></td>
<td>0x120</td>
<td></td>
<td>%edi</td>
</tr>
<tr>
<td></td>
<td>0x110</td>
<td></td>
<td>%esp</td>
</tr>
<tr>
<td></td>
<td>0x108</td>
<td></td>
<td>%ebp</td>
</tr>
<tr>
<td></td>
<td>0x104</td>
<td></td>
<td>%ebp</td>
</tr>
<tr>
<td></td>
<td>0x100</td>
<td></td>
<td>%ebp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl 12(%ebp),%ecx</td>
<td># ecx = yp</td>
</tr>
<tr>
<td>movl 8(%ebp),%edx</td>
<td># edx = xp</td>
</tr>
<tr>
<td>movl (%ecx),%eax</td>
<td># eax = *yp (t1)</td>
</tr>
<tr>
<td>movl (%edx),%ebx</td>
<td># ebx = *xp (t0)</td>
</tr>
<tr>
<td>movl %eax,(%edx)</td>
<td># *xp = eax</td>
</tr>
<tr>
<td>movl %ebx,(%ecx)</td>
<td># *yp = ebx</td>
</tr>
</tbody>
</table>
Understanding Swap (6)

Register Allocation (By compiler)

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<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>

Address

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td>4</td>
<td>0x10c</td>
</tr>
<tr>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

123

456

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,(%ebx)    # *xp = eax
movl %ebx,(%ecx)    # *yp = ebx
Understanding Swap (7)

Register Allocation (By compiler)

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<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>

Address

- %eax: 456
- %edx: 0x124
- %ecx: 0x120
- %ebx: 123
- %esi
- %edi
- %esp
- %ebp: 0x104

Offset

- yp: 12, Address: 0x120
- xp: 8, Address: 0x124
- %ebp: 0, Address: 0x100
- -4

Code:

- 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

---

SSE2030: Introduction to Computer Systems | Fall 2010 | Jin-Soo Kim (jinsookim@skku.edu)
Understanding Swap (8)

Register Allocation (By compiler)

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
</tr>
<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
# Arithmetic/Logical Ops. (1)

## Two operands instructions

- **addl**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} + \text{Src} \]

- **subl**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} - \text{Src} \]

- **mull**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} \times \text{Src} \text{ (unsigned)} \]

- **imull**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} \times \text{Src} \text{ (signed)} \]

- **sall**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} \ll \text{Src} \text{ (Arithmetic)} \]

- **sarl**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} \gg \text{Src} \text{ (Arithmetic)} \]

- **shrl**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} \gg \text{Src} \text{ (Logical)} \]

- **xorl**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} \oplus \text{Src} \]

- **andl**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} \& \text{Src} \]

- **orl**  `Src, Dest`  
  
  \[ \text{Dest} = \text{Dest} \mid \text{Src} \]
Arithmetic/Logical Ops. (2)

- One operand instructions
  - incl Dest
    Dest = Dest + 1
  - decl Dest
    Dest = Dest - 1
  - negl Dest
    Dest = - Dest
  - notl Dest
    Dest = ~Dest
Address Computation

- **leal** \( \text{Src, Dest} \)
  - \( \text{Src} \) is address mode expression
  - Set \( \text{Dest} \) to address denoted by expression

- **Uses**
  - Computing address without doing memory reference
    - e.g., translation of \( p = \&x[i] \);
  - Computing arithmetic expressions of the form \( x + k*y \)
    - \( k = 1, 2, 4, \) or 8
Example: arith (1)

```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;
}
```

arith:

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

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

movl %ebp,%esp
popl %ebp
ret
```

Set Up

Body

Finish
Example: arith (2)

```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;
}
```

Stack

- Offset: 16
  - z
- Offset: 12
  - y
- Offset: 8
  - x
- Offset: 4
  - Rtn adr
- Offset: 0
  - Old %ebp
  - %ebp

Assembly code:

- `movl 8(%ebp),%eax`  # eax = x
- `movl 12(%ebp),%edx`  # edx = y
- `leal (%edx,%eax),%ecx`  # ecx = x+y (t1)
- `leal (%edx,%edx,2),%edx`  # edx = 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)
Example: logical

```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;
}
```

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

```
logical:
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%eax
xorl 12(%ebp),%eax
sar1 $17,%eax
andl $8185,%eax
movl %ebp,%esp
popl %ebp
ret
```

```
movl 8(%ebp),%eax    eax = x
xorl 12(%ebp),%eax   eax = x^y    (t1)
sar1 $17,%eax        eax = t1>>17  (t2)
andl $8185,%eax      eax = t2 & 8185
```
CISC Properties

- **CISC (Complex Instruction Set Computer)**
  - Instruction can reference different operand types
    - Immediate, register, memory
  - Arithmetic operations can read/write memory
  - Memory reference can involve complex computation
    - \( R_b + S \times R_i + D \)
    - Useful for arithmetic expressions, too.
  - Instructions can have varying lengths
    - IA-32 instructions can range from 1 to 15 bytes
Summary (1)

- **Machine level programming**
  - Assembly code is textual form of binary object code
  - Low-level representation of program
    - Explicit manipulation of registers
    - Simple and explicit instructions
    - Minimal concept of data types
    - Many C control constructs must be implemented with multiple instructions
Machine Models

C

mem proc

Data
1) char
2) int, float
3) double
4) struct, array
5) pointer

Control
1) loops
2) conditionals
3) switch
4) Proc. call
5) Proc. return

Compiler

Assembly

mem Stack

regs Cond. Codes

alu processor

1) byte
2) 2-byte word
3) 4-byte long word
4) contiguous byte allocation
5) address of initial byte