198:211
Computer Architecture

Lecture 10
Fall 2012

Topics: Chapter 3
- Assembly Language 3.2
- Register Transfer 3.4
- ALU 3.5
Assembly level Programming

- We are now familiar with high level programming languages such as C and Java
- Computers execute machine code
- Compilers generate machine code from source code
- We need to understand how the code actually executes on various machines (architectures)
  - Studying binary code is not very easy
- At the same time, we need a language that mimics the machine level instructions and still readable (textual format)
- Enter Assembly: compilers can generate an intermediate step of code called assembly code
- Assemblers then convert assembly code to machine code
(recall) Von Neumann Architecture

- Model of a computer that used stores programs
  - Both Data and Program stored in memory
  - Allows the computer to be “Re-programmed”
- CPU is central to the computer
Simplified hardware view

Store: Store contents of MDR to Memory address specified by MAR

Load: Load into MDR contents of Memory address specified by MAR

MAR: Memory Address Register also known as Program Counter
MDR: Memory Data Register
C: Control Switch; 0 is Load Or 1 is Store
Fetch-execute cycle

Notation: [X] or (X) contents stored at location (memory address) contained in Register X

IR ← [PC]
Execute [IR]
PC ← PC +1

- IR executes the instructions
- Where are the operands?
- As part of the execution, data may be transferred among various registers in the CPU as well as memory
- Typical data movement instruction is MOV
Assembly programmer view

- ALU: Arithmetic Logic Unit
- IR: Instruction register
- GPR: General Purpose Registers
- PC: Program Counter
- SP: Stack Pointer
- BR: Base Register
**MOV instruction**

- Most common instruction is data transfer instruction
  - Notation: \texttt{mov} \texttt{S, D} (contents of \texttt{S} becomes contents of \texttt{D})

- \texttt{mov} data from memory to register and register to memory

- \texttt{mov} data between registers
  - Notation: registers are preceded by \% sign

- \texttt{mov} data to and from stack

- \texttt{mov} constants to registers
  - Notation: constant preceded by \$
Data formats

- Data is represented in different sizes
- Byte ... 8 bits
  - E.g., Char
- Word .. 16 bits (2 bytes)
  - E.g., Short int
- Double Word (long or dword) .. 32 bits (4 bytes)
  - E.g., float
- QWORD .. 64 bits (8 bytes)
  - E.g., double
- Instructions can operate on any data size
  - movl, movw, movb
  - Operates on doubleword, word, byte respectively
Bit and byte order

Little-endian
# x86 General purpose registers (8)

<table>
<thead>
<tr>
<th>EAX</th>
<th>AX</th>
<th>AH</th>
<th>AL</th>
<th>8 BITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECX</td>
<td>CX</td>
<td>CH</td>
<td>CL</td>
<td></td>
</tr>
<tr>
<td>EDX</td>
<td>DX</td>
<td>DH</td>
<td>DL</td>
<td></td>
</tr>
<tr>
<td>EBX</td>
<td>BX</td>
<td>BH</td>
<td>BL</td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td>--Stack Pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBP</td>
<td>--- Base register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

16 BITS

32 BITS
Registers

- Registers are 32 bit (operations can access 16 bits, 8 bits within the register)
  - Operations involving registers are typically single cycle (nano seconds)...
- Types of registers
- Data registers (EAX, EBX, ECX, EDX)
  - Holds operands
- Pointer and Index registers (EBP, ESP, EIP, ESI, EDI)
  - Holds references to addresses as well as indexes
- Segment registers
  - Holds starting address of program segments (CS, DS, SS, ES)
Assembly Language Characteristics

- Minimal Data Types
  - “Integer” data of 1, 2, or 4 bytes
    - Data values
    - Addresses (untyped pointers)
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory

- Primitive Operations
  - Perform arithmetic function on register or memory data
  - Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches
Assembly level instructions

- Opcode byte | Addressing byte | Other bytes

- Instruction length varies from 1 byte to several bytes
- Instruction consists of
- First byte is the Opcode byte and has short names called mnemonic
  - Opcode byte is typically 1 byte
  - More recent instructions set (floating point, multimedia, parallel) have multiple opcode bytes
- Second byte is the Addressing byte (for data handling instructions)
- Specifies the type of Operands which one is a register, a memory and the type of addressing
- For many instructions operands are implicitly specified by designated registers
  - push
Machine Instruction Example

C Code
- Add two signed integers

Assembly
- Add 2 4-byte integers
  - “Long” words in GCC parlance
  - Same instruction whether signed or unsigned

Operands:
- x: Register %eax
- y: Memory M[ebp+8]
- t: Register %eax
  - Return function value in %eax

Object Code
- 3-byte instruction
- Stored at address 0x401046

C Code
```c
int t = x+y;
```

Assembly
```assembly
addl 8(%ebp),%eax
```

Similar to expression
```c
x += y
```

Object Code
```assembly
0x401046: 03 45 08
```


Sample code in C, assembly, machine code on a SUN Sparc System

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
    int x=1,
    y=2;
    int sum;
    sum=x+y;
}
```

```asm
mov          1, %o0
st           %o0, [%fp-20]
mov          2, %o0
st           %o0, [%fp-24]
ld            [%fp-20], %o0
ld            [%fp-24], %o1
add          %o0, %o1, %o0
st           %o0, [%fp-28]
```

```gcc
gcc  -s sum.c
```

```objdump
objdump –d  sum.o > machinecode.txt
```
Sample code in C, assembly, machine code on a intel x86 machine

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
    int x=1,
    y=2;
    int sum;
    sum=x+y;
}
```

```
movl $1, -4(%ebp)
movl $2, -8(%ebp)
movl -8(%ebp), %eax
addl -4(%ebp), %eax
movl %eax, -12(%ebp)
```

```
8048350:c7 45 fc 01 00 00 00     movl $0x1,0xffffffffc(%ebp)
8048357:c7 45 f8 02 00 00 00     movl $0x2,0xffffffff8(%ebp)
804835e:8b 45 f8                 mov 0xffffffff8(%ebp),%eax
8048361:03 45 fc                 add 0xffffffffc(%ebp),%eax
8048364:89 45 f4                 mov  %eax,0xffffffff4(%ebp)
```
Addressing byte

- The second byte indicates how to get the data

**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 “address modes”
Immediate addressing

- Operand is immediate
  - Operand value is found immediately following the instruction
- `movl $x4040, %eax`

![Diagram showing processor, memory, and instruction execution]
Direct addressing

- Address of operand is found immediately after the instruction
  - Also known as direct addressing or absolute address
  - Move source of operand found at address I
  - `movl x000f, %eax,`
  - `/* source of operand is at address*/`
  - Load effective address; load address directly
  - `leal x 0013, %ebx`

2009

eax

000f

movl eax

0000f

0100

lea ebx

0013

1234
Register-mode addressing

- One or more operands are found in registers
  - `movl %eax, %ebx`
  - Move contents of eax to ebx
  - `add %ebx, %eax`
  - Add contents of ebx and eax and store result in eax
  - `mov %ea, %eb`
  - Move 16 bit LSB of eax to 16 bit LSB of ebx
  - `mov %al, %bl`
  - Move 8 bit LSB of eax to 8 bit LSB of ebx
Indirect mode addressing

- Content of operand is an address
- Offset can be specified as immediate mode
  - `movl (%ebp), %eax`
  - `movl -4(%ebp), %eax`

```assembly
movl (%ebp), %eax
movl -4(%ebp), %eax
```
Indexed mode addressing

- Consider the problem of initializing an array \( a[5] \) to 0
- One could use \texttt{movl} $0, x1040$, \texttt{movl} $0, x1044$, ...
- Each instruction is a repeat of the other except the address
  - E.g., for loop
- Execute the same instruction with a different operand each time
- Alter the address of operand by adding the contents of another register
  - \texttt{movl} \((\text{eab,esi}), \%eax\)
  - \texttt{movl} \(8(\text{eab,esi}), \%eax\)
Scaled index mode

- Same piece of code need to iterate over multiple byte sizes

- `movl (eab,esi,4), %eax`
  - Move content of address=eab+esi*4 to eax

- `leal (%edx,%edx,4), %eax`
  - Copy effective address = %edx + %edx*4=5*%edx to %eax
## Address Computation Examples

<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>

<table>
<thead>
<tr>
<th>%edx</th>
<th>0xf000</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>0x100</td>
</tr>
</tbody>
</table>
### movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Immn</strong></td>
<td><strong>Reg</strong></td>
<td>movl $\text{0x4}, %eax</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movl $\text{-147}, (%eax)</td>
</tr>
<tr>
<td><strong>Regn</strong></td>
<td><strong>Reg</strong></td>
<td>movl (%eax, %edx)</td>
</tr>
<tr>
<td></td>
<td><strong>Mem</strong></td>
<td>movl (%eax, (%edx))</td>
</tr>
<tr>
<td><strong>Memn</strong></td>
<td><strong>Reg</strong></td>
<td>movl ((%eax), %edx)</td>
</tr>
</tbody>
</table>

- Cannot do memory-memory transfers with single instruction
Stack operations

- `%esp` contains the address of top of stack
  - `pushl %eax, popl %ebx`
- `pushl` stores operand in top of stack
- `popl` stores top of stack in destination operand
Using Simple Addressing Modes

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

```assembly
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
```

Set Up

Body

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

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

```
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
```

<table>
<thead>
<tr>
<th>Address</th>
<th>0x124</th>
<th>0x120</th>
<th>0x11c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rtn adr</td>
<td>0x118</td>
<td>0x114</td>
<td>0x10c</td>
</tr>
<tr>
<td>Offset</td>
<td>0x120</td>
<td>0x124</td>
<td>0x10c</td>
</tr>
<tr>
<td>yp</td>
<td>12</td>
<td>0x104</td>
<td>0x100</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Understanding Swap

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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
</tr>
<tr>
<td>0x120</td>
</tr>
<tr>
<td>0x11c</td>
</tr>
<tr>
<td>0x118</td>
</tr>
<tr>
<td>0x114</td>
</tr>
<tr>
<td>0x110</td>
</tr>
<tr>
<td>0x10c</td>
</tr>
<tr>
<td>0x108</td>
</tr>
<tr>
<td>0x104</td>
</tr>
<tr>
<td>0x100</td>
</tr>
</tbody>
</table>

Variables:
- %eax
- %edx
- %ecx: 0x120
- %ebx
- %esi
- %edi
- %esp
- %ebp: 0x104
Understanding Swap

```
          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

- `%eax` = 456
- `%edx` = 0x124
- `%ecx` = 0x120
- `%ebx` = 0x104
- `%esi` = 
- `%edi` = 
- `%esp` = 0x104
- `%ebp` = 0x104

```
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

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

```
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

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

#### Code Snippet

```
    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
```
## Some Arithmetic Operations

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Two Operand Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>addl</td>
<td><em>Src,Dest</em></td>
</tr>
<tr>
<td>subl</td>
<td><em>Src,Dest</em></td>
</tr>
<tr>
<td>imull</td>
<td><em>Src,Dest</em></td>
</tr>
<tr>
<td>sall</td>
<td><em>Src,Dest</em></td>
</tr>
<tr>
<td>sarl</td>
<td><em>Src,Dest</em></td>
</tr>
<tr>
<td>shrl</td>
<td><em>Src,Dest</em></td>
</tr>
<tr>
<td>xorl</td>
<td><em>Src,Dest</em></td>
</tr>
<tr>
<td>andl</td>
<td><em>Src,Dest</em></td>
</tr>
<tr>
<td>orl</td>
<td><em>Src,Dest</em></td>
</tr>
</tbody>
</table>
Shift right

- **sarl** arithmetic right Shift
  - `sarl k, D`
    - Shift right k bits (D), retaining MSB
    - `sarl 1, 10001 → ?`

- **shrl** logical right shift
  - Shift right k bits (D), fill with 0s from the right
  - `shrl 1, 10001 → ?`
# Some Arithmetic Operations: Unary

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>incl</strong> Dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td><strong>decl</strong> Dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td><strong>negl</strong> Dest</td>
<td>Dest = - Dest  twos complement negation</td>
</tr>
</tbody>
</table>

Replaces the value of the operand with its twos complement (equivalent to subtracting from 0)

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>notl</strong> Dest</td>
<td>Dest = ~ Dest</td>
</tr>
</tbody>
</table>

Bitwise negation of all the bits (each 1 is replaced by 0 and each 0 is replaced by 1)
Using \texttt{leal} for Arithmetic Expressions

\begin{verbatim}
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;
}
\end{verbatim}

\texttt{arith}:

\begin{verbatim}
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
\end{verbatim}
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),%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)
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;
}

# eax = x
  movl 8(%ebp),%eax
# edx = y
  movl 12(%ebp),%edx
# ecx = x+y  (t1)
  leal (%edx,%eax),%ecx
# edx = 3*y
  leal (%edx,%edx,2),%edx
# edx = 48*y (t4)
  sall $4,%edx
# ecx = z+t1 (t2)
  addl 16(%ebp),%ecx
# eax = 4+t4+x (t5)
  leal 4(%edx,%eax),%eax
# eax = t5*t2 (rval)
  imull %ecx,%eax
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
```

\[2^{13} = 8192, \ 2^{13} - 7 = 8185\]