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 is 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 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: \(\text{mov } S, D\) (contents of S becomes contents of D)
- mov data from memory to register and register to memory
- mov data between registers
  - Notation: registers are preceded by \% sign
- mov data to and from stack
- 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)|
|---|---|---|
|EAX| AX| 16 BITS|
|EBX| BX| 32 BITS|
|ECX| CX| 32 BITS|
|EDX| DX| 32 BITS|
|ESP| ---| 32 BITS|
|EBP| ---| 32 BITS|
|ESI| | |
|EDI| | |

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**

Instructions length varies from 1 byte to several bytes

First byte is the Opcode byte and has short names called mnemonic

Opcode byte is typically 1 byte

More recent instructions (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

#### Machine Instruction Example

**C Code**

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

**Assembly**

```c
addl $8(%ebp),%eax
```

Similar to expression

```
x = y
```

**Object Code**

```c
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;
}
```

```c
mov 1, %o0
mov 1, %o0
mov 0, %o0
mov 2, %o0
mov 0, %o0
mov 0, %o0
ld 0, %o0
ld 0, %o0
ld 0, %o0
```

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;
}
```

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

```c
8048350: c7 45 fc 01 00 00 00
8048357: c7 45 f8 02 00 00 00
804835e: 03 45 fc
8048364: 03 45 08
```

```c
objdump -d sum.o > machinecode.txt
```
**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.
  - E.g., `movl $x4040, %eax`.

---

**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:
    - `movl %eax, %ebx`.
    - Move effective address; load address directly.
  - E.g., `mov %eax, %ebx`.

---

**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.
  - Move 16 bit LSB of eax to 16 bit LSB of ebx.
  - Move %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`

```
0010 000C 0008 0014
  ebp  movl(%ebp)  esi
```

### Indexed mode addressing

- Consider the problem of initializing an array `a[5]` to 0
- One could use `movl $0, x1040, movl $0, x1044, ...`
- Each instruction is a repeat of the other expect 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
    - `movl (eab,esi), %eax`
    - `movl 8(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>0x1000 + 0x8</td>
<td>0x1008</td>
</tr>
<tr>
<td>(%edx,%ecx)</td>
<td>0x1000 + 0x100</td>
<td>0x1200</td>
</tr>
<tr>
<td>0x100(%edx,4)</td>
<td>0x1000 + 4*0x100</td>
<td>0x1400</td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0x1000 + 0x80</td>
<td>0x1a080</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>Reg</td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl $-147,(%eax)</td>
</tr>
<tr>
<td>Mem</td>
<td>Mem</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;
}
```

Stack

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

Understanding Swap

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>*yp (t1)</th>
<th>*xp (t0)</th>
<th>*xp</th>
<th>*yp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>12</td>
<td>8</td>
<td>0x104</td>
<td>0x108</td>
<td>0x108</td>
<td>0x100</td>
<td>0x120</td>
</tr>
<tr>
<td>0x124</td>
<td>4</td>
<td>4</td>
<td>0x104</td>
<td>0x100</td>
<td>0x100</td>
<td></td>
<td></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

### Address

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
<th>yp</th>
<th>xp</th>
<th>*yp (t1)</th>
<th>*xp (t0)</th>
<th>*xp</th>
<th>*yp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>12</td>
<td>8</td>
<td>0x104</td>
<td>0x108</td>
<td>0x108</td>
<td>0x100</td>
<td>0x120</td>
</tr>
<tr>
<td>0x124</td>
<td>4</td>
<td>4</td>
<td>0x104</td>
<td>0x100</td>
<td>0x100</td>
<td></td>
<td></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

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><strong>addl</strong></td>
<td>*Src, Dest = Dest + Src</td>
</tr>
<tr>
<td><strong>subl</strong></td>
<td>*Src, Dest = Dest - Src</td>
</tr>
<tr>
<td><strong>imull</strong></td>
<td>*Src, Dest = Dest × Src</td>
</tr>
<tr>
<td><strong>sall</strong></td>
<td>*Src, Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td><strong>srcl</strong></td>
<td>*Src, Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td><strong>xorl</strong></td>
<td>*Src, Dest = Dest ^ Src</td>
</tr>
<tr>
<td><strong>andl</strong></td>
<td>*Src, Dest = Dest &amp; Src</td>
</tr>
<tr>
<td><strong>orl</strong></td>
<td>*Src, Dest = Dest</td>
</tr>
</tbody>
</table>
Shift right

- `sarl k, D` arithmetic right Shift
  - Shift right k bits (D), retaining MSB
- `shrl k, D` 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>incl Dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decl Dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negl 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)

- `notl Dest  Dest = ~Dest` Bitwise negation of all the bits (each 1 is replaced by 0 and each 0 is replaced by 1)

Using `leal` for Arithmetic Expressions

```
int arith (int x, int y, int z) {
    int t1 = x+y;
    int t2 = x+t1;
    int t3 = x+4;
    int t4 = y*48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Understanding arithmetic

```assembly
    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 4(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull tecx,teax
    movl lecp,teax
    popl kebp
    ret
```

<table>
<thead>
<tr>
<th>Offset</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>x</td>
</tr>
<tr>
<td>12</td>
<td>y</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
</tbody>
</table>

```
movl 8(%ebp),teax # eax = x
movl 12(%ebp),tecx # edx = y
leal (%edx,teax),tecx # ecx = x+y (t1)
leal (%edx,tecx,2),tecx # edx = x+y (t2)
sall $4,tecx # edx = 4*y (t4)
addl 4(%ebp),tecx # ecx = x+t1 (t3)
leal 4(%edx,teax,teax) # eax = 4*t2 (t5)
imull tecx,teax # eax = t3*t2 (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;
}
```

### 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;
}
```

### Code Snippet

- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`
  - `movl 8(%ebp),%eax`
  - `xorl 12(%ebp),%eax`  
    - `eax = x ^ y`
  - `sarl $17,%eax`
    - `eax = t1 >> 17`
  - `andl $8185,%eax`
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`

- **Body**
  - `movl 8(%ebp),%eax`
  - `movl 12(%ebp),%edx`
  - `leal (%edx,%eax),%ecx`
    - `eax = (t1)`
  - `leal (%edx,%edx,2),%edx`
  - `sall $4,%edx`
  - `leal (%edx,%edx),%eax`
    - `eax = 3*y`
  - `addl 16(%ebp),%ecx`
  - `movl %eax,%edx`
    - `eax = z + t1`
  - `addl 16(%ebp),%ecx`
    - `eax = 4 + t4`
  - `addl 4(%edx,%eax),%eax`
    - `eax = t3 + t4`
  - `imull %ecx,%eax`
    - `eax = t5 * t2`
  - `movl %eax,%ecx`
    - `eax = t2 * t5`

- **Finish**
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`