Machine Language

COMP375
Computer Architecture and Organization
“If you thought assembly language was low level, try programming in machine language.”
Assembler Assignment

• The second assembler programming assignment has been posted on Blackboard
• You are required to write four short program segments in assembler
• Upload your .cpp files to Blackboard by noon on tomorrow
Assembler and Machine

• Assembler language is the easy way to write machine language
• Each line of an assembler program generates one machine language instruction
• The assembler allows you to use variable names instead of numerical addresses and instruction mnemonics instead of numerical operation codes
Bunch of Bytes

• Machine language contains the binary codes that the computer executes
• The computer fetches the instructions from memory and executes them

Machine language for a student program

8b 45 e0 89 45 f8 89 45 ec 8b 45 ec
89 45 f8 8b 45 e0 ba 00 00 00 00 f7
7d f8 03 45 f8 d1 f8 89 45 ec 3b 45
f8 75 e2 8b f4
Instruction Format

• The general format for a machine language instruction is

  Op code  |  Operands

The operands can be a memory address, a register or a value
Op codes

• Each assembler instruction represents a numerical machine language opcode

  add   05
  cmp   3B
  dec   FF
  idiv  7F
  jmp   39
  push  68
  sar   D0
If the computer has 210 different instructions, how long is the opcode?

A. 7 bits  
B. 8 bits  
C. 9 bits  
D. 210 bits  
E. Does not matter
Data Location

- **Register** – The data is in a CPU register
- **Memory** – The data is in a location in RAM
- **Immediate** – The data is part of the instruction
  Immediate data items are read-only
The Intel assembler allows you to use one mnemonic for different op codes.

There are several versions of the add instruction based on the size of the operands. The assembler picks the correct op code.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 ib</td>
<td>ADD imm8 to AL</td>
</tr>
<tr>
<td>05 iw</td>
<td>ADD imm16 to AX</td>
</tr>
<tr>
<td>05 id</td>
<td>ADD imm32 to EAX</td>
</tr>
<tr>
<td>80 /0 ib</td>
<td>ADD imm8 to r/m8</td>
</tr>
<tr>
<td>81 /0 iw</td>
<td>ADD imm16 to r/m16</td>
</tr>
<tr>
<td>81 /0 id</td>
<td>ADD imm32 to r/m32</td>
</tr>
<tr>
<td>83 /0 ib</td>
<td>ADD sign-extended</td>
</tr>
<tr>
<td>83 /0 ib</td>
<td>ADD sign-extended i</td>
</tr>
<tr>
<td>00 /r</td>
<td>ADD r8 to r/m8</td>
</tr>
<tr>
<td>01 /r</td>
<td>ADD r16 to r/m16</td>
</tr>
<tr>
<td>01 /r</td>
<td>ADD r32 to r/m32</td>
</tr>
<tr>
<td>02 /r</td>
<td>ADD r/m8 to r8</td>
</tr>
<tr>
<td>03 /r</td>
<td>ADD r/m16 to r16</td>
</tr>
<tr>
<td>03 /r</td>
<td>ADD r/m32 to r32</td>
</tr>
</tbody>
</table>
Machine Language for Add

• The following assembler add instructions generate machine language with different opcodes

<table>
<thead>
<tr>
<th>Machine language</th>
<th>Assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>03 45 f8</td>
<td>add eax, anInt</td>
</tr>
<tr>
<td>66 03 45 ec</td>
<td>add ax, aShort</td>
</tr>
<tr>
<td>02 45 e3</td>
<td>add al, aChar</td>
</tr>
<tr>
<td>03 c3</td>
<td>add eax, ebx</td>
</tr>
</tbody>
</table>
Mnemonic to Op Code Mapping

• Intel assembler uses the same mnemonic for the machine language instruction to:
  – Move a byte from memory to a register (load)
  – Move a byte from a register to memory (store)

• The Intel mov instruction generates different machine language op codes depending upon the size of the operands
Number of Operands

• In addition to the op code, the instruction might contain zero, one, two or three operands
• Different architectures use different number of operands
• A single architecture may have instructions with differing number of operands
No operand instructions

• Some instructions do not require any operands. The data affected is implied in the instruction.

• **RET** — Return from function
• **HLT** — Halt
• **CPUID** — Get details about the CPU
• **LAHF** — Load Status Flags into AH Reg
One Operand Instructions

• Some unary operations require only one operand

  inc ebx - increment the ebx register
  jmp address – jump to the address
Two Operand Instructions

• Many instructions act on two operands
• Most math instructions use two operands and return the results in one of them

\[
\begin{align*}
\text{add} & \quad \text{edx, varname} \\
\text{add} & \quad \text{ebx, eax} \\
\text{imul} & \quad \text{eax, varname}
\end{align*}
\]

Op code | register | address
--- | --- | ---
Op code | reg1 | reg2
Three Operand Instructions

- Some machines support three operands (Intel Pentium does not)
- Most ARM instructions use three operands

add R1, R2, R3
Additional Instruction Fields

• Most architectures support an addressing mode that combines an address field in the instruction and the contents of a register

\[
\text{add} \quad R3, \text{addr}[R7]
\]

• This instruction adds the contents of register R3 with the memory location whose address is the sum of the address field and R7
Variable or Fixed Length

• Some architectures use variable length instructions. Instructions with more operands or memory addresses are longer
  – saves memory
• Some architectures always use the same length instruction
  – easier to find the beginning of instructions
  – instructions are in aligned words
Assembled Code

<table>
<thead>
<tr>
<th>addr</th>
<th>machine</th>
<th>assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>004a</td>
<td>8b 45 e0</td>
<td>mov eax, number[ebp]</td>
</tr>
<tr>
<td>004d</td>
<td>89 45 f8</td>
<td>mov good[ebp], eax</td>
</tr>
<tr>
<td>0050</td>
<td>89 45 ec</td>
<td>mov better[ebp], eax</td>
</tr>
<tr>
<td>0053</td>
<td>8b 45 ec</td>
<td>mov eax, better[ebp]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>again:</td>
</tr>
<tr>
<td>0056</td>
<td>89 45 f8</td>
<td>mov good[ebp], eax</td>
</tr>
<tr>
<td>0059</td>
<td>8b 45 e0</td>
<td>mov eax, number[ebp]</td>
</tr>
<tr>
<td>005c</td>
<td>ba 00 00 00 00</td>
<td>mov edx, 0</td>
</tr>
<tr>
<td>0061</td>
<td>f7 7d f8</td>
<td>idiv good[ebp]</td>
</tr>
<tr>
<td>0064</td>
<td>03 45 f8</td>
<td>add eax, good[ebp]</td>
</tr>
<tr>
<td>0067</td>
<td>d1 f8</td>
<td>sar eax, 1</td>
</tr>
<tr>
<td>0069</td>
<td>89 45 ec</td>
<td>mov better[ebp], eax</td>
</tr>
<tr>
<td>006c</td>
<td>3b 45 f8</td>
<td>cmp eax, good[ebp]</td>
</tr>
<tr>
<td>006f</td>
<td>75 e2</td>
<td>jne SHORT again</td>
</tr>
<tr>
<td>0071</td>
<td>8b f4</td>
<td>mov esi, esp</td>
</tr>
</tbody>
</table>
The first two *mov* instructions generated different op codes. Why?

A. different indexing
B. different sizes
C. different direction
D. different address
Example Machine Language

- Assume each instruction of this imaginary computer is 32 bits in length

```
8             4        4               16 bits
label:   mnemonic
reg
address
```

<table>
<thead>
<tr>
<th>opcode</th>
<th>reg</th>
<th>index reg</th>
<th>address</th>
</tr>
</thead>
</table>

- label: mnemonic
- reg, address[index reg]

*imaginary Assembler language format*
World of Numbers

- The opcodes are numbers.
- The address is a number.
- The register field is a number.

Add R3, [no indexing] xyz

```
8
4
4
16 bits
```

<table>
<thead>
<tr>
<th>10100101</th>
<th>0011</th>
<th>0000</th>
<th>000000001011010</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>reg</td>
<td>index</td>
<td>address</td>
</tr>
<tr>
<td>00110101</td>
<td>0011</td>
<td>0000</td>
<td>000000001011010</td>
</tr>
</tbody>
</table>

```
Machine Language Program

00  01100018  LOAD   R1, y
04  2113001C  DIV    R1, z[R3]
08  05100020  ADD    R1, five
0C  02100014  STORE  R1, x
10  47000000  RET

Simplified, not Intel
Short Instructions

- Register to register instructions and those not needing additional operands can use a shorter format.
- Instruction lengths should be an even multiple of a byte.

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>Register</th>
<th>Register</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td>4 bit</td>
</tr>
</tbody>
</table>
Disassembling

• Any bunch of bits in memory can be considered a program
• Most random bits may not produce a logical program and may generate errors due to bad opcodes or addressing errors
• A disassembler is a program that interprets the values in memory as instructions
• Some software asks not to be disassembled
What are the Instructions?

### Opcodes

<table>
<thead>
<tr>
<th></th>
<th>Add</th>
<th>Subtract</th>
<th>Multiply</th>
<th>Divide</th>
<th>Load</th>
<th>Store</th>
<th>JumpEql</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

### Instruction Formats

#### Load and Store

<table>
<thead>
<tr>
<th>opcode</th>
<th>register</th>
<th>index register</th>
<th>memory address</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>21</td>
</tr>
</tbody>
</table>

#### Add, Sub, Mult and Divide

<table>
<thead>
<tr>
<th>opcode</th>
<th>unused</th>
<th>reg 1</th>
<th>reg 2</th>
<th>reg 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- Opcodes:
  - Add: 0
  - Subtract: 1
  - Multiply: 2
  - Divide: 3
  - Load: 4
  - Store: 5
  - JumpEql: 6
  - Jump: 7

- Instruction Formats:
  - Load and Store: `opcode register index register memory address`
    - 3 4 4 21
  - Add, Sub, Mult and Divide: `opcode unused reg 1 reg 2 reg 3`
What are the Instructions?

Opcodes

<table>
<thead>
<tr>
<th>Add</th>
<th>Subtract</th>
<th>Multiply</th>
<th>Divide</th>
<th>Load</th>
<th>Store</th>
<th>JumpEql</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Instruction Formats

Load and Store

<table>
<thead>
<tr>
<th>opcode</th>
<th>register</th>
<th>index register</th>
<th>memory address</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>21</td>
</tr>
</tbody>
</table>

Add, Sub, Mult and Divide

<table>
<thead>
<tr>
<th>opcode</th>
<th>unused</th>
<th>reg 1</th>
<th>reg 2</th>
<th>reg 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 100000001011000000000000000000000000000000000100000 |
First Instruction

Opcodes

<table>
<thead>
<tr>
<th>Add</th>
<th>Subtract</th>
<th>Multiply</th>
<th>Divide</th>
<th>Load</th>
<th>Store</th>
<th>JumpEql</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Instruction Formats

Load and Store

- Opcode: 3
- Register: 4
- Index Register: 4
- Memory Address: 21

Add, Sub, Mult and Divide

- Opcode: 3
- Unused: 1
- Reg 1: 4
- Reg 2: 4
- Reg 3: 4

Opcode 100 = 4 = Load instruction
Register 0001 = R1
Index 0110 = R6
Address 0001000 = 8

Load R1, 8[R6]
This instruction is

A. Add R1,6
B. Multiply R2, R1, R6
C. Load R2, 1[R6]
D. Multiply R2, 6[R1]
Notes on the Example Architecture

• This is an example of a “Load/Store” architecture. Only the load and store instructions access memory.
• Why is there a one bit unused field in the arithmetic instructions?
• Why is the opcode always the left most bits?
• The format of the jump instructions was not shown. What might be a good format?
Univac Register Addressing

- The Univac 1100 had 16 A registers, 16 X registers and 16 R registers. Four of the A regs overlapped with the X for a total of 44.
- There was a separate register set for the OS.
- Program addresses started at address 1000 (octal) instead of zero like most architectures.
- “Memory” addresses less than 1000 were used to specify a register instead of memory.

<table>
<thead>
<tr>
<th>opcode</th>
<th>register</th>
<th>index register</th>
<th>I</th>
<th>U</th>
<th>memory address or register</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>18</td>
</tr>
</tbody>
</table>
PDP-8 Instruction Set

• The PDP-8 was a popular minicomputer made by DEC in the 1970’s
• The 12 bit fixed length instructions had a 3 bit op code, an address field and two addressing mode bits
• There were 7 instructions with the normal format
• One op code was for zero operand instructions. The remaining bits of the instruction specified what to do
PDP-8 Memory Addressing

• The PDP-8 instruction only has a 7 bit field to hold a memory address
• $2^7$ is only 128 words, a very small amount of memory
• The two address bits specified
  – page zero or this page
  – direct or indirect addressing
• This allows the program to address 128 pages of 128 bytes or 16K words
Where is the data?

• The operands for an instruction can be in RAM, a register or in the instruction
• The instruction specifies where the operand is located
• There are several ways the operand’s address can be specified
• Different addressing modes have been created for different program situations
Addressing modes

• immediate
• register
• memory direct
• register indirect
• register indirect with offset
• memory indirect
• displacement
Immediate

- The data is part of the instruction
- Immediate data items are read-only
- There is usually a size limit

```
sub eax, 5
```
Register

- The data is in a CPU register
- The instruction *might* indicate which register

```
dec ebx
```
Memory Direct

- The data is in memory
- The instruction contains the address of the memory location

`sub ebx, dog`
Register Indirect

- The address of the data is in a CPU register
- Useful if the address is calculated

\[ \text{sub } \text{ebx}, [\text{edx}] \]
Register Indirect with Offset

• The address of the data is the sum of the instruction offset field and a register value

• Useful when addressing an array

\[
\text{sub ebx, dog[edx]}
\]
Memory Indirect

• A memory location contains the address of the data
• Useful for pointers

\[ \text{sub ebx, } \ast\text{dog} \]  // not real Intel

Instruction → address

data register

data

memory
Stack Addressing

• The same as Register Indirect with Offset using the stack pointer register
• Useful when addressing local variables or parameters

\[ \text{sub ebx, dog[esp]} \]
Displacement

- The address of the data is the sum of the instruction offset field and the program counter
- Used for short jumps: `jmp SHORT cat`
The fastest addressing mode in this list is

A. Immediate
B. Direct
C. Memory Indirect
D. Register Indirect with Offset
Assembler Assignment

• The second assembler programming assignment has been posted on Blackboard
• You are required to write four short program segments in assembler
• Upload your .cpp files to Blackboard by noon on Friday, February 3, 2017