Machine Language

COMP375
Computer Architecture and Organization

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 is binary codes that the computer executes.
• The computer fetches the instructions from memory and executes them.

Machine language for a square root 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>05</td>
</tr>
<tr>
<td>cmp</td>
<td>3B</td>
</tr>
<tr>
<td>dec</td>
<td>FF</td>
</tr>
<tr>
<td>div</td>
<td>7F</td>
</tr>
<tr>
<td>jmp</td>
<td>39</td>
</tr>
<tr>
<td>push</td>
<td>68</td>
</tr>
<tr>
<td>sar</td>
<td>D0</td>
</tr>
</tbody>
</table>

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.

Intel Assembler

• 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 Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04 ib ADD AL,imm8</td>
<td>ADD imm8 to AL</td>
</tr>
<tr>
<td>05 iw ADD AX,imm16</td>
<td>ADD imm16 to AX</td>
</tr>
<tr>
<td>05 id ADD EAX,imm32</td>
<td>ADD imm32 to EAX</td>
</tr>
<tr>
<td>80 0 ib ADD r/m8,imm8</td>
<td>ADD imm8 to r/m8</td>
</tr>
<tr>
<td>81 0 ib ADD r/m16,imm16</td>
<td>ADD imm16 to r/m16</td>
</tr>
<tr>
<td>81 0 id ADD r/m16,imm32</td>
<td>ADD imm32 to r/m16</td>
</tr>
<tr>
<td>83 0 ib ADD r/m16,imm8</td>
<td>ADD sign-extended</td>
</tr>
<tr>
<td>00 r ADD r/m8,r/m</td>
<td>ADD r8 to r/m8</td>
</tr>
<tr>
<td>01 r ADD r/m16,r16</td>
<td>ADD r16 to r/m16</td>
</tr>
<tr>
<td>02 r ADD r/m32,r32</td>
<td>ADD r32 to r/m32</td>
</tr>
<tr>
<td>03 r ADD r16,r/m16</td>
<td>ADD r16 to r/m16</td>
</tr>
<tr>
<td>03 r ADD r32,r/m32</td>
<td>ADD r32 to r/m32</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
  – Move a byte from a register to memory
• 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

OneOperand Instructions

- Some unary operations require only one operand

  inc al  - increment the al 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.

  add al, varname
  add bl, al
  imul eax, varname

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

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

\[ \text{add } R1, R2, R3 \]

Additional Instruction Fields

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

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

- 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>
Questions about the code

• What does SHORT mean after the `jne`?
• What is the displacement of the `jne`?

Example Machine Language

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

```
8 4 4 16 bits
```

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

label: mnemonic reg, address[index reg]

Assembler language format

Machine Language Program

```
00 01100018 LOAD R1, y
04 2110001C DIV R1, z
08 05100020 ADD R1, five
0C 02100014 STORE R1, x
10 47000000 RET
14 x res
18 y res
1C z res
20 00000005 five +5
```

World of Numbers

• The opcodes are numbers
• The address is a number
• The register field is a number

```
8 4 4 16 bits
```

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

Add R3, [no indexing] xyz
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?

<table>
<thead>
<tr>
<th>Opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Instruction Formats

- Load and Store
  - opcode: 3
  - register: 4
  - index: 4
  - memory address: 21

- Add, Sub, Mult and Divide
  - opcode: 3
  - register: 1
  - unused: 1
  - reg 2: 4
  - reg 3: 4

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?
Instruction Cycle

• Fetch the instruction from the memory address in the Program Counter register
• Increment the Program Counter
• Decode the type of instruction
• Fetch the operands
• Execute the instruction
• Store the results

Basic Processor Components

• **Program Counter** – contains the address of the next instruction to execute.
• **Arithmetic Logic Unit** – logic to perform arithmetic and logical functions
• **User registers** – hold data
• **Memory Address Register** – contains the address to be copied to or from RAM
• **Memory Buffer Register** – contains data copied to or from RAM.
Consider an arithmetic instruction followed by a jump instruction.

- The arithmetic instruction sets bits in the status register.
Execution Stage of Instruction 1

Result Save Stage of Instruction 1

Instruction 2 Fetch

Instruction 2 Fetch
Increment Program Counter

Program Counter → Instruction Register → Memory Address Reg → Memory Buffer Reg

Status Register

Memory Address Reg → Memory Buffer Reg

Program Counter → Instruction Register

Instruction Register → ALU +1

R1

R2

... 

R16

Memory Address Reg → Memory Buffer Reg

Exit

Increment Program Counter

Program Counter → Instruction Register → ALU

Status Register

R1

R2

... 

R16

Memory Address Reg → Memory Buffer Reg

Exit

Decode Instruction

Program Counter → Instruction Register → Status Register

ALU

Memory Address Reg → Memory Buffer Reg

Exit

Execution of Instruction 2

Program Counter → Instruction Register → Status Register

ALU

Memory Address Reg → Memory Buffer Reg

Exit
Saving Result of Instruction 2