Machine and Assembly Language
Machine Language

- The machine language for a particular computer is tied to the architecture of the CPU.
- For example: G4 Macs have a different machine language than Intel PC's.
- We will look at the machine language of a simple, simulated computer.
Von Neumann Architecture
Von Neumann Architecture

- Program and Data are both stored in memory.
- Fetch, Decode, Execute cycle…
Machine Language Example

- Our computer has 4 registers and 32 memory locations.
- Each instruction is 16 bits.
- Here is a machine language program for our simulated computer:

```
1000000100100101
1000000101000101
1010000100000110
1000001000000110
1111111111111111
```
## Sample Instruction

<table>
<thead>
<tr>
<th>Instruction ID</th>
<th>Register #</th>
<th>Memory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD contents of memory location into register</td>
<td>1000000010 RR MMMMM</td>
<td>example: R0 = Mem[3] 100000010 00 00011</td>
</tr>
</tbody>
</table>
## Machine Language

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Code</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOAD</strong> contents of memory location into register</td>
<td>1000000010 RR MMMMMM</td>
<td>ex: R0 = Mem[3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000000010 00 00011</td>
</tr>
<tr>
<td><strong>STORE</strong> contents of register into memory location</td>
<td>1000000100 RR MMMMMM</td>
<td>ex: Mem[4] = R0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000001000 00 00100</td>
</tr>
<tr>
<td><strong>MOVE</strong> contents of one register into another register</td>
<td>100100010000 RR RR</td>
<td>ex: R0 = R1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100100010000 00 00 01 }</td>
</tr>
</tbody>
</table>
## Machine Language

<table>
<thead>
<tr>
<th>Operation</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD contents of 2 registers, store result in third.</td>
<td>1010000100 RR RR RR RR</td>
</tr>
<tr>
<td>ex: $R0 = R1 + R2$</td>
<td>1010001000 00 01 10</td>
</tr>
<tr>
<td>Halt the program</td>
<td>111111111111111</td>
</tr>
</tbody>
</table>

| SUBTRACT contents of 2 registers, store result into third | 1010001000 RR RR RR |
| ex: $R0 = R1 – R2$ | 1010001000 00 01 10 |
Reading Machine Language

- In our case, first nine bits specifies the operation, last 6 (or 7) bits specifies the arguments:

  100000010 01 00101  Load Memory 5 -> R1
  100000010 10 00101  Load Memory 5 -> R2
  1010000100 00 01 10  R1 + R2 -> R0
  100000100 00 00110  Store R0 -> Memory 6
  1111111111111111

- It is very tedious to program in machine language.
Assembly Language

- Assembly instructions are just shorthand for machine instructions:

<table>
<thead>
<tr>
<th>Machine Language</th>
<th>Equivalent Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000100100101</td>
<td>LOAD R1 5</td>
</tr>
<tr>
<td>1000000101000101</td>
<td>LOAD R2 5</td>
</tr>
<tr>
<td>1010000100000110</td>
<td>ADD R0 R1 R2</td>
</tr>
<tr>
<td>1000001000000110</td>
<td>SAVE R0 6</td>
</tr>
<tr>
<td>1111111111111111</td>
<td>HALT</td>
</tr>
</tbody>
</table>

- (For all assembly instructions that compute a result, the first argument is the destination.)
- Very easy to write an Assembly Language -> Machine language translator.
Exercise

- What would be the assembly instructions to swap the contents of registers 1 & 2?
- STORE [MEM] [REG]
- LOAD [REG] [MEM]
- MOVE [REG] [REG]
- ADD [REG] [REG] [REG]
- SUB [REG] [REG] [REG]
- HALT
Exercise Solution

STORE 1 R1
MOVE R1 R2
LOAD R2 1
HALT
Some More Instructions…

- We are missing some crucial functionality…
- ??
Some More Instructions…

- We are missing some crucial functionality…
- Loops!

<table>
<thead>
<tr>
<th>Branch to a location in memory</th>
<th>BRANCH [MEM]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch if the ALU result is zero.</td>
<td>BZERO [MEM]</td>
</tr>
<tr>
<td>Branch if the ALU result is negative.</td>
<td>BNEG [MEM]</td>
</tr>
</tbody>
</table>
A More Complex Example

<table>
<thead>
<tr>
<th>R0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>Number</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>ADD R3 R2 R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SUB R0 R0 R1</td>
</tr>
<tr>
<td>2</td>
<td>BZERO 4</td>
</tr>
<tr>
<td>3</td>
<td>BRANCH 0</td>
</tr>
<tr>
<td>4</td>
<td>MOVE R2 R3</td>
</tr>
<tr>
<td>5</td>
<td>HALT</td>
</tr>
</tbody>
</table>
In Matlab

- The same program in Matlab would be the following:
  
  ```matlab
  z = 0;
  x = 3;
  while x ~= 0
      z = z + y;
      x = x - 1;
  end
  y = z;
  ```

- Or the following:
  
  ```matlab
  y = y*3;
  ```
Problems with Assembly

- Why might we avoid writing in assembly?
High Level Languages: Compilation and Interpretation

• Typically, we write in a language that is (relatively) easy to use.
• A translator program converts our instructions to machine instructions for the target computer.
  – Interpreter – Translation is on-the-fly. (Matlab)
  – Compiler – Translation happens all at once.
• Advantages and disadvantages...
Because it is not fun to program in Assembly, we have “high level” programming languages.

- Matlab
- Python, C, C++, Java, Fortran, Cobol, Pascal, M, Ada, Lisp, Ruby, Smalltalk, C#, Haskell, Prolog…

- Compiler/Interpreter translates from the high-level language to machine language.
Program Translation

Compiler/Interpreter

```
z = 0;
x = 3;
while x ~= 0
    z = z + y;
    x = x - 1;
y = z;
```

Assembler

```
ADD R3 R2 R3
SUB R0 R0 R1
BZERO 4
BRANCH 0
MOVE R2 R3
HALT
```

```
1010000100000110
1010010000001110
0000011000000100
0000001000000000
1001000100001011
1111111111111111
```