Machine Language

Where we are at:

Human Thought

Abstract design
Chapters 9, 12

Machine Language

Computer Architecture
Chapters 4 - 5

Hardware Platform

Gate Logic
Chapters 1 - 3

Chips & Logic Gates

Electrical Engineering

Physics

Machine Language
Chapters 4 - 5

Harvard University
CS 101
Fall 2005, Shimon Schocken

Elements of Computing Systems 2 Machine Language (Ch. 4)

Copyright © Shimon Schocken
Machine language is “the soul of the machine”

Duality:

- Machine language (= instruction set) can be viewed as an abstract description of the hardware platform
- The hardware can be viewed as a means for realizing an abstract machine language

Another duality:

- Binary version
- Symbolic version

Loose definition:

- Machine language = an agreed upon formalism for manipulating a memory using a processor and a set of registers
- Varies across different hardware platforms.

---

Binary and symbolic notation

```
1010 0011 0001 1001
ADD R3, R1, R9
```

Evolution:

- Physical coding
- Symbolic documentation
- Symbolic coding
- Requires a translator.

---

Jacquard loom
(1801)

Ada Lovelace
(1815-1852)
Lecture plan

- Machine languages at a glance
- The Hack machine language:
  - Symbolic version
  - Binary version
- Perspective.

Arithmetic / logical operations

ADD R2,R1,R3 // R2 ← R1 + R3 where R1,R2,R3 are registers
AND R1,R1,R2 // R1 ← bit wise And of R1 and R2
ADD R2,R1,foo // R2 ← R1 + foo where foo stands for the value of the
// memory location pointed at by the user-defined
// label foo.
Memory access

**Direct addressing:**

\[
\text{LOAD R1, 67} \quad // \quad \text{R1} \leftarrow \text{Memory[67]}
\]

// Or, assuming that bar refers to memory address 67:

\[
\text{LOAD R1, bar} \quad // \quad \text{R1} \leftarrow \text{Memory[67]}
\]

**Immediate addressing:**

\[
\text{LOADI R1, 67} \quad // \quad \text{R1} \leftarrow 67
\]

\[
\text{STORE R1, bar} \quad // \quad \text{bar} \leftarrow \text{R1}
\]

**Indirect addressing:**

// Translation of \(x=\text{foo[j]}\) or \(x=*(\text{foo+j})\):

\[
\text{ADD R1, foo, j} \quad // \quad \text{R1} \leftarrow \text{foo+j}
\]

\[
\text{LOAD* R2, R1} \quad // \quad \text{R2} \leftarrow \text{memory[R1]}
\]

\[
\text{STORE R2, x} \quad // \quad \text{x} \leftarrow \text{R2}
\]

Flow of control

**Branching**

\[
\text{JMP foo} \quad // \quad \text{unconditional jump}
\]

**Conditional branching**

\[
\text{JGT R1, foo} \quad // \quad \text{If R1}>0, goto foo}
\]

// in general:

\[
\text{cond register, label}
\]

Where: \(\text{cond}\) is \text{JEQ, JNE, JGT, JGE, ...}

\(\text{register}\) is \text{R1, R2, ...} and

\(\text{label}\) is a user-defined label

- And that’s all you need in order to implement any high-level control structure in any programming language.
A hardware abstraction (Hack)

- **Registers**: D, A
- **Data memory**: \( M \equiv \text{RAM}[A] \)
- **ALU**: \( \{D|A|M\} = \text{ALU}(D,A,M) \)
- **Instruction memory**: current instruction = ROM[A]
- **Control**: instruction memory is loaded with a sequence of instructions, one per memory location. The first instruction is stored in ROM[0].
- **Instruction set**: A-instruction, C-instruction

A-instruction

\[ @\text{value} \quad // \quad A \leftrightarrow \text{value} \]

Where \text{value} is either a number or a symbol referring to some number.

**Used for**:

- Entering a constant (\( A = \text{value} \))
- Selecting a RAM location (\( M \equiv \text{RAM}[A] \))
- Selecting a ROM location (\( \text{instruction} = \text{ROM}[A] \)).
C-instruction

\[
\text{dest} = \text{comp} ; \text{jump} \quad // \text{comp is mandatory}\n\]
\[
\text{// dest and jump are optional}\n\]

Where:

\text{comp} \text{ is one of:}

\begin{aligned}
0, &1,-1,D,A,!D,!A,-D,-A,D+1,A+1,D-1,A-1,D+A,D-A,D&A,D|A, \\
& M, !M, -M, M+1, M-1,D+M,D-M,M-D,D&M,D|M
\end{aligned}

\text{dest} \text{ is one of:}

\begin{aligned}
\text{Null, } &M, D, MD, A, AM, AD, AMD
\end{aligned}

\text{jump} \text{ is one of:}

\begin{aligned}
\text{Null, } &JGT, JEQ, JGE, JLT, JNE, JLE, JMP
\end{aligned}

Coding examples

1. Set A to 17
   \[
   @\text{value} \quad // \text{set A to value}\n   \]
2. Set D to A-1
   \[
   \text{dest} = \text{comp} ; \text{jump}\n   \]
3. Set both A and D to A+1
4. Compute \(-1\)
5. Set D to 19
6. Set RAM[53] to 171
7. Set both A and D to A+D
8. Set RAM[5034] to D-1
9. Add 1 to RAM[7], and also store the result in D.
Higher-level coding examples

10. sum = 12
11. j = j + 1
12. q = sum + 12 - j
13. x[j] = 15

Etc.

Symbol table:

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>20</td>
</tr>
<tr>
<td>j</td>
<td>17</td>
</tr>
<tr>
<td>sum</td>
<td>22</td>
</tr>
<tr>
<td>q</td>
<td>21</td>
</tr>
<tr>
<td>X</td>
<td>16</td>
</tr>
<tr>
<td>end</td>
<td>507</td>
</tr>
<tr>
<td>next</td>
<td>112</td>
</tr>
</tbody>
</table>

Control (first approximation)

- ROM = instruction memory
- Program = sequence of 16-bit numbers, starting at ROM[0]
- Current instruction = ROM[address]
- To select instruction n from the ROM, we set A to n, using the command @n
Coding branching operations (examples)

Low level:
1. IF D=0 GOTO 112
2. IF D-12<5 GOTO 507
3. IF D-1>RAM[12] GOTO 112

Higher level:
4. IF sum>0 GOTO end
5. IF x[i]-12<=y GOTO next

C-instruction

dest = comp ; jump

where jump is one of:
Null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP

Symbol table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>20</td>
</tr>
<tr>
<td>j</td>
<td>17</td>
</tr>
<tr>
<td>sum</td>
<td>22</td>
</tr>
<tr>
<td>q</td>
<td>21</td>
</tr>
<tr>
<td>X</td>
<td>16</td>
</tr>
<tr>
<td>end</td>
<td>507</td>
</tr>
<tr>
<td>next</td>
<td>112</td>
</tr>
</tbody>
</table>

Flow of control operations (IF logic)

High level:
IF condition {
    code segment 1
} ELSE {
    code segment 2
} Etc.

Low level (goto logic)
D ← not(condition)
    IF D=0 GOTO if_true
ELSE
    code segment 2
    GOTO end
    if_true:
        code segment 1
    end:
    Etc.

Hack:
D ← not(condition)
@IF_TRUE
    D;JEQ
    code segment 2
    @END
    0;JMP
    (IF_TRUE)
    code segment 1
    (END)
    Etc.

To prevent conflicting use of the A register, in well-written programs a C-instruction that includes a jump directive should not contain a reference to M, and vice versa.
Flow of control operations (WHILE logic)

**High level:**

```plaintext
WHILE condition {
    code segment 1
}  
code segment 2
```

**Hack:**

```plaintext
(LOOP)
    @END
    D ← not(condition)
    D;jeq
    code segment 1
    @LOOP
    0;jmp
(END)
    code segment 2
```

Complete program example

**C:**

```c
// Adds 1+...+100.
int i = 1;
int sum = 0;
while (i <= 100){  
    sum += i;
    i++;  
}
```
Lecture plan

- Symbolic machine language
- Binary machine language

A-instruction

**Symbolic:** \( \emptyset \text{value} \) // Where value is either a non-negative decimal number // or a symbol referring to such number.

\[ \text{value} (v = 0 \text{ or } 1) \]

**Binary:**

| 0 | V | V | V | V | V | V | V | V | V | V | V | V | V |
---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Copyright © Shimon Schocken
Symbols (user-defined)

- **Label symbols**: User-defined symbols, used to label destinations of goto commands. Declared by the pseudo command \((Xxx)\). This directive defines the symbol \(Xxx\) to refer to the instruction memory location holding the next command in the program.

- **Variable symbols**: Any user-defined symbol \(Xxx\) appearing in an assembly program that is not defined elsewhere using the \((Xxx)\) directive is treated as a variable, and is assigned a unique memory address by the assembler, starting at RAM address 16.
Symbols (pre-defined)

- **Virtual registers**: \texttt{R0,...,R15} are predefined to be 0,...,15

- **I/O pointers**: The symbols \texttt{SCREEN} and \texttt{KBD} are predefined to be 16384 and 24576, respectively (base addresses of the screen and keyboard memory maps)

- **Predefined pointers**: the symbols \texttt{SP, LCL, ARG, THIS, and THAT} are predefined to be 0 to 4, respectively.

---

Perspective

- Hack is a simple language

- User friendly syntax: \texttt{D=D+A} instead of \texttt{ADD D,D,A}

- Hack is a “½-address machine”

- A Macro-language can be easily developed

- Assembler.