Chapter 4:
Machine Language

Usage and Copyright Notice:

Copyright 2005 © Noam Nisan and Shimon Schocken


The book web site, www.idc.ac.il/tecs, features 13 such presentations, one for each book chapter. Each presentation is designed to support about 3 hours of classroom or self-study instruction.

You are welcome to use or edit this presentation for instructional and non-commercial purposes.

If you use our materials, we will appreciate it if you will include in them a reference to the book’s web site.

And, if you have any comments, you can reach us at tecs.ta@gmail.com
Where we are at:

- **Human Thought** (Chapters 9, 12)
- **Abstract design**
  - **H.L. Language & Operating Sys.** (Chapters 10 - 11)
  - **Compiler**
    - **Virtual Machine** (Chapters 7 - 8)
    - **Assembly Language**
  - **VM Translator**
- **Assembler** (Chapter 6)
- **Computer Architecture** (Chapters 4 - 5)
- **Hardware Platform** (Chapters 1 - 3)
- **Chips & Logic Gates**
- **Electrical Engineering** (Physics)
- **Software hierarchy**
  - **Machine Language**
  - **Abstract interface**
  - **Hardware hierarchy**
  - **Abstract design**

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

- Same spirit but different syntax across different hardware platforms.
Binary and symbolic notation

**Evolution:**
- Physical coding
- Symbolic documentation
- Symbolic coding
- Requires a *translator.*

1010 0011 0001 1001

ADD R3, R1, R9

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 (in typical machine language syntax)

ADD R2,R1,R3  // R2 ← R1+R3 where R1,R2,R3 are registers

AND R1,R1,R2  // R1 ← And(R1,R2) (bit-wise)

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 (in typical machine language syntax)

### Direct addressing:

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

\[
\text{// Or, assuming that bar refers to memory address 67:}
\]

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

### Immediate addressing:

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

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

### Indirect addressing:

\[
\text{// x=foo[j], also known as: } x=*(\text{foo+j}): \\
\text{ADD R1, foo, j} \quad \text{// R1} \leftarrow \text{foo+j}
\]

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

\[
\text{STORE R2, x} \quad \text{// x} \leftarrow \text{R2}
\]
Flow of control (in typical machine language syntax)

Branching

JMP foo  // unconditional jump

Conditional branching

JGT R1,foo  // If R1>0, goto foo

// in general:
cond register, label

Where: cond is JEQ, JNE, JGT, JGE, ...

   register is R1, R2, ...

   label is a user-defined label

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

- **Registers**: \( D, A \)

- **Data memory**: \( M = \text{RAM}[A] \)  
  (\( M \) stands for \( \text{RAM}[A] \))

- **ALU**: \( \{D | A | M\} = \text{ALU}(D, A, M) \)  
  (set \( D, A, \) or \( M \) to the ALU output on the inputs \( D, A, M \))

- **Instruction memory**: current instruction = \( \text{ROM}[A] \)

- **Control**: instruction memory is loaded with a sequence of instructions, one per memory location. The first instruction is stored in \( \text{ROM}[0] \)

- **Instruction set**: \( A \)-instruction, \( C \)-instruction.
A-instruction

@value // A ← value

Where value is either a number or a symbol referring to some number.

Used for:

- Entering a constant (A = value)
- Selecting a RAM location (M ≡ RAM[A])
- Selecting a ROM location (instruction = ROM[A])
C-instruction

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

Where:

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


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

\[\text{Null, M, D, MD, A, AM, AD, AMD}\]

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

\[\text{Null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP}\]
Write the Hack instructions that implement each of the following tasks:

1. Set A to 17

2. Set D to A-1

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>

@value // set A to value

dest = comp ; jump
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 \( @n \)
Write the Hack instructions that implement each of the following tasks:

**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
Flow of control operations (IF logic)

High level:

IF condition {
  code segment 1}
ELSE {
  code segment 2}
etc.

Low level (goto logic)

D \leftarrow \text{not(condition)}
D=0 \text{ GOTO if\_true}
ELSE
  code segment 2
  GOTO end
if\_true:
  code segment 1
end:
  etc.

Hack:

D \leftarrow \text{not(condition)}
@\text{IF\_TRUE}
D;\text{JEQ}
code segment 2
@end
0;\text{JMP}
(\text{IF\_TRUE})
  code segment 1
(\text{END})
  etc.

- To prevent conflicting use of the A register, in well-written Hack 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:

WHILE condition {
  code segment 1
}
code segment 2

Hack:

(LOOP)
D ← not(condition)
@END
D; jeq
code segment 1
@LOOP
0; jmp
(END)
code segment 2
Complete program example

C:

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

Hack:

// Adds 1+...+100.
@i  // i refers to some mem. location
M=1  // i=1
@sum  // sum refers to some mem. location
M=0  // sum=0
(LOOP)
    @i
    D=M  // D=i
    @100
    D=D-A  // D=i-100
    @END
    D;JGT  // If (i-100)>0 goto END
    @i
    D=M  // D=i
    @sum
    M=D+M  // sum=sum+i
    @i
    M=M+1  // i=i+1
    @LOOP
    0;JMP  // Goto LOOP
(END)
    @END
    0;JMP  // Infinite loop
Lecture plan

- **Symbolic machine language**
- **Binary machine language**
A-instruction

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

\(value\) (\(v = 0\) or \(1\))

**Binary:**

```
0  v  v  v
v  v  v  v
v  v  v  v
v  v  v  v
v  v  v  v
```
C-instruction

Symbolic: \( dest = comp \); \( jump \)  // Either the dest or jump fields may be empty.

\[
\begin{array}{c|cccccc}
\text{comp} & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \\
\hline
D & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-1 & 1 & 1 & 1 & 0 & 1 & 0 \\
D & 0 & 0 & 1 & 1 & 0 & 0 \\
A & 1 & 1 & 0 & 0 & 0 & 0 \\
!D & 0 & 0 & 1 & 1 & 0 & 1 \\
!A & 1 & 1 & 0 & 0 & 0 & 1 \\
-D & 0 & 0 & 1 & 1 & 1 & 1 \\
-A & 1 & 1 & 0 & 0 & 1 & 1 \\
D+1 & 0 & 1 & 1 & 1 & 1 & 1 \\
A+1 & 1 & 1 & 0 & 1 & 1 & 1 \\
D-1 & 0 & 0 & 1 & 1 & 1 & 0 \\
A-1 & 1 & 1 & 0 & 0 & 1 & 0 \\
D+A & 0 & 0 & 0 & 0 & 0 & 0 \\
D-A & 0 & 1 & 0 & 0 & 1 & 1 \\
A-D & 0 & 0 & 0 & 1 & 1 & 1 \\
D&A & 0 & 0 & 0 & 0 & 0 & 0 \\
D|A & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

Binary: 1 1 1 a  c1 c2 c3 c4  c5 c6  d1 d2  d3 j1 j2 j3

\[
\begin{array}{c|ccc|c|l}
\text{d1} & \text{d2} & \text{d3} & \text{Mnemonic} & \text{Destination (where to store the computed value)} \\
\hline
0 & 0 & 0 & null & The value is not stored anywhere \\
0 & 0 & 1 & M & Memory[A] (memory register addressed by A) \\
0 & 1 & 0 & D & D register \\
0 & 1 & 1 & MD & Memory[A] and D register \\
1 & 0 & 0 & A & A register \\
1 & 0 & 1 & AM & A register and Memory[A] \\
1 & 1 & 0 & AD & A register and D register \\
1 & 1 & 1 & AMD & A register, Memory[A], and D register \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|l}
j_1 & j_2 & j_3 & \text{Mnemonic} & \text{Effect} \\
\hline
0 & 0 & 0 & null & No jump \\
0 & 0 & 1 & JGT & If out > 0 jump \\
0 & 1 & 0 & JEQ & If out = 0 jump \\
0 & 1 & 1 & JGE & If out ≥ 0 jump \\
1 & 0 & 0 & JLT & If out < 0 jump \\
1 & 0 & 1 & JNE & If out ≠ 0 jump \\
1 & 1 & 0 & JLE & If out ≤ 0 jump \\
1 & 1 & 1 & JMP & Jump \\
\end{array}
\]
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.

```assembly
// Rect program
@R0
D=M
@INFINITE_LOOP
D;JLE @counter
M=D
@SCREEN
D=A
@addr
M=D
(LOOP)
@addrA=M
M=-1
@addrD=M
@32
D=D+A
@addr
M=D
@counter
MD=M-1
@LOOP
D;JGT
(INFINITE_LOOP)
@INFINITE_LOOP
0;JMP
```
Symbols (pre-defined)

- **Virtual registers**: \( R_0, \ldots, R_{15} \) are predefined to be \( 0, \ldots, 15 \)

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

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

---

```
// Rect program
@R0
D=M
@INFINITE_LOOP
D;JLE @counter
M=D
@SCREEN
D=A
@addrM=D
(LOOP)
@addrA=M
M=-1
@addrD=M
@32D=D+A
@addr
M=D
@counterMD=M-1
LOOPD;JGT
(INFINITE_LOOP)
@INFINITY LOOP
0;JMP
```

---

Virtual registers: \( R_0, \ldots, R_{15} \) are predefined to be \( 0, \ldots, 15 \)

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

Predefined pointers: the symbols `SP`, `LCL`, `ARG`, `THIS`, and `THAT` are predefined to be 0 to 4, respectively.
Perspective

- Hack is a simple language
- User friendly syntax: \( D = D + A \) instead of \( \text{ADD D, D, A} \)
- Hack is a “\( \frac{1}{2} \)-address machine”
- A Macro-language can be easily developed
- Assembler.
Assignment:

1. \( x = \text{constant} \) (e.g. \( x=17 \))
2. \( x = y \)
3. \( x = 0 \), \( x = 1 \), \( x = -1 \)

Arithmetic / logical:

4. \( x = y \ op \ z \)
   where \( y \), \( z \) are variables or constants and \( op \) is some ALU operation like +, -, and, or, etc.

Control:

5. GOTO s
6. IF cond THEN GOTO s
   where \( \text{cond} \) is an expression \( (x \ op \ y) \ \{=|<|>|...\} \ \{0|1\} \)
   e.g. IF \( x+17>0 \) goto loop

White space or comments:

7. White space: ignore
8. // comment to the end of the line: ignore.