Machine Language

Usage and Copyright Notice:

Copyright 2005 © Noam Nisan and Shimon Schocken


We provide both PPT and PDF versions.

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 as you see fit 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.

If you have any questions or comments, you can reach us at tecs.ta@gmail.com
Where we are at:

- **Abstract design**
  - Chapters 9, 12

- **Human Thought**

- **Machine Language**
  - Chapters 4 - 5

- **Hardware Platform**
  - Chapters 1 - 3

- **Computer Architecture**
  - Chapters 4 - 5

- **Gate Logic**
  - Chapters 1 - 3

- **Virtual Machine**
  - Chapters 7 - 8

- **Assembly Language**

- **Compiler**
  - Chapters 10 - 11

- **VM Translator**

- **Assembler**
  - Chapter 6
Machine language is “the soul of the machine”

Duality:

- Machine language (= instruction set) can be viewed as an abstract (programmer-oriented) description of the hardware platform
- The hardware can be viewed as a physical 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

1010 0001 0010 1011
ADD R1, R2, R3

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

Jacquard loom
(1801)

Augusta Ada King,
Countess of Lovelace
(1815-1852)
Lecture plan

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

(The assembler will be covered in lecture 6).
Instructions in a typical machine language

// In what follows R1, R2, R3 are registers, PC is program counter, and addr is a value.

ADD R1, R2, R3     // R1 ← R2 + R3

ADDI R1, R2, addr  // R1 ← R2 + addr

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

JMP addr           // PC ← addr

JEQ R1, R1, addr   // IF R1 = R2 THEN PC ← addr ELSE PC++

LOAD R1, addr      // R1 ← RAM[addr] Where v is an address

STORE R1, addr     // RAM[addr] ← R1 where v is an address

NOOP                // Do nothings

// Plus several more commands that are essentially versions
// or extensions of the above commands.
The Hack computer

The 16-bit Hack computer consists of the following elements:

**Data memory:**  RAM – a series of 16-bit words

**Instruction memory:**  ROM – a series of 16-bit words

**Registers:**  D, A, M, where M stands for RAM[A]

**Processing:**  ALU, capable of computing various functions

**Program counter:**  PC, holding an address

**Control:** The ROM is loaded with a sequence of 16-bit instructions, one per memory location, beginning at address 0. The next instruction is always fetched from ROM[PC]

**Instruction set:**  Two instructions: A-instruction, C-instruction.
A-instruction

@\text{value} \quad // \quad A \leftarrow \text{value}

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

**Used for:**

- Entering a constant value \((A = \text{value})\)

- Selecting a RAM location \((\text{register} = \text{RAM}[A])\)

- Selecting a ROM location \((\text{fetch} \ \text{ROM}[A])\)

**Coding example:**

\begin{align*}
@17 & \quad // \quad A = 17 \\
D & = A \quad // \quad D = 17
\end{align*}

@17 \quad // \quad A = 17 \\
D & = M \quad // \quad D = \text{RAM}[17]

@17 \quad // \quad A = 17 \\
\text{JMP} & \quad // \quad \text{fetch the instruction} \\
& \quad // \quad \text{stored in ROM[17]}

Later
Coding examples (programming practice)

Write the Hack instructions that implement the following tasks:

- Set A to 17
- Set D to A-1
- Set both A and D to A + 1
- Set D to 19
- Set both A and D to A + D
- Set RAM[5034] to D - 1
- Set RAM[53] to 171
- Add 1 to RAM[7], and store the result in D.

Hack commands:

@value // set A to value

dest = x op y

op is + or -
x is A, D, or M
y is A, D, M or 1
(op y) is optional
dest is D, M, MD, A, AM, AD, AMD, or null
Coding examples (cont.)

Write the Hack instructions that implement the following tasks:

- `sum = 0`
- `j = j + 1`
- `q = sum + 12 - j`
- `arr[7] = 0`

Hack commands:

```plaintext
@value  // set A to value
dest = x op y

op is + or -
x is A, D, or M
y is A, D, M or 1
(op y) is optional
dest is D, M, MD, A, AM, AD, AMD, or null
```

Symbol table:

```
j    17
sum  22
q    21
arr  16
```

(All symbols and values in are arbitrary examples)
Control (first approximation)

- ROM = instruction memory

- Program = sequence of 16-bit numbers, starting at ROM[0]

- Current instruction = ROM[A]

- To select instruction \( n \) from the ROM, we set \( A \) to \( n \), using the instruction @\( n \)

(The actual architecture is slightly different, as we’ll see in the next lecture)
Write the Hack instructions that implement the following tasks:

- GOTO 50
- IF D = 0 GOTO 112
- IF D < 9 GOTO 507
- IF RAM[12] > 0 GOTO 50
- IF sum > 0 GOTO END
- IF axis] <= 0 GOTO NEXT.

Symbol table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>200</td>
</tr>
<tr>
<td>x</td>
<td>4000</td>
</tr>
<tr>
<td>i</td>
<td>151</td>
</tr>
<tr>
<td>END</td>
<td>50</td>
</tr>
<tr>
<td>NEXT</td>
<td>120</td>
</tr>
</tbody>
</table>

(All symbols and values in are arbitrary examples)
C-instruction syntax (final version)

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

Where:

- **comp** is one of:

- **dest** is one of:
  - \{null, M, D, MD, A, AM, AD, AMD\}

- **jump** is one of:
  - \{null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP\}
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.
WHILE logic – Hack style

High level:

while condition {
   code segment 1
}
// next instruction

Hack:

(LOOP)
   D ← not condition)
   @END
   D;JEQ
   code segment 1
   @LOOP
   0;JMP
(END)
// next instruction
Complete program example

C:

// Adds 1+...+100.
  into i = 1;
  into sum = 0;
  while (i <= 100)
    sum += i;
    i++;

Hack:

// Adds 1+...+100.
  @i  // i refers to some memo. location
  M=1  // i=1
  @sum  // sum refers to some memo. 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 got END
  @i
  D=M  // D = i
  @sum
  M=D+M  // sum += i
  @i
  M=M+1  // i++
  @LOOP
  0;JMP  // Got LOOP

(END)
  @END
  0;JMP  // Infinite loop
Lecture plan

- **Symbolic machine language**

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

### Symbolic:
\[
\text{@value} \quad // \text{Where value is either a non-negative decimal number} \\
\quad // \text{or a symbol referring to such number.}
\]

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

### Binary:

<table>
<thead>
<tr>
<th>0</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
</tr>
</thead>
</table>
### C-instruction

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

**Binary:**

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

<table>
<thead>
<tr>
<th>(\text{comp})</th>
<th>(\text{dest})</th>
<th>(\text{jump})</th>
<th>(\text{Mnemonic})</th>
<th>(\text{Destination (where to store the computed value)})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>null</td>
<td>The value is not stored anywhere</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>N</td>
<td>Memory[A] (memory register addressed by A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>D</td>
<td>D register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>MD</td>
<td>Memory[A] and D register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>A</td>
<td>A register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>AM</td>
<td>A register and Memory[A]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>AD</td>
<td>A register and D register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>AMD</td>
<td>A register, Memory[A], and D register</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(j1)</th>
<th>(j2)</th>
<th>(j3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{out} &lt; 0)</td>
<td>(\text{out} = 0)</td>
<td>(\text{out} &gt; 0)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(\text{Mnemonic})</th>
<th>(\text{Effect})</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>No jump</td>
</tr>
<tr>
<td>JGT</td>
<td>If out &gt; 0 jump</td>
</tr>
<tr>
<td>JEQ</td>
<td>If out = 0 jump</td>
</tr>
<tr>
<td>JGE</td>
<td>If out ≥ 0 jump</td>
</tr>
<tr>
<td>JLT</td>
<td>If out &lt; 0 jump</td>
</tr>
<tr>
<td>JNE</td>
<td>If out ≠ 0 jump</td>
</tr>
<tr>
<td>JLE</td>
<td>If out ≤ 0 jump</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
</tbody>
</table>
Symbols (user-defined)

- **Label symbols**: User-defined symbols, used to label destinations of got 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.

- By convention, label symbols are upper-case and variable symbols are lower-case.

```plaintext
// Recto program
@R0 D=M
@INFINITE_LOOP D;JLE @counter M=D @SCREEN D=A @addr M=D
(LOOP)
@addr A=M M=-1@addr D=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.
Perspective

- Hack is a simple machine 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

- A Hack assembler is needed and will be discussed and developed later in the course.
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 $+$, $-$, $\text{and}$, $\text{or}$, etc.

Control:
5. GOTO $s$
6. IF $\text{condo}$ THEN GOTO $s$
   where $\text{condo}$ is an expression $(x \ op \ y) \ {=} |<|>|\ldots|0|1$
   e.g. IF $x+17>0$ got loop

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