Machine Language

Saturday, February 12, 2022 10:29 PM

"Works of imagination should be written in very plain language; the more purely imaginative they are, the more necessary it is to be plain."
- Samuel Taylor Coleridge (1772-1834)

Machine Language

"Form Follows Function" - The form of something will define what its functionality is capable of.

Machine Language - An agreed-upon formalism designed to code machine instruction.

With these instructions we can ask the computer to then perform arithmetic, read/write to memory, test Boolean logic, and decide which instructions to fetch and execute next.

Machine language is the connection between hardware and higher level software.

Overview

4.1.1 - Hardware Elements

Memory - A collection of hardware devices that store data and instructions in a computer. Also referred to as locations or memory registers. Each have a unique address in

Processor - Central Processing Unit (CPU) is a device that is capable of performing a fixed set of primitive operations.

- Arithmetic
- Boolean logic
- Memory access operations
- Control (branching) operations

Consists of an ALU, a set of registers, and gate logic that enables it to parse and execute binary instructions

Registers - Processor and memory are implemented as two separate chips. Moving data from ne location to another is a fairly slow process. Due to this processors are equipped with registers. These registers store 1 single value (the size of your machine for Nand it will be a 16 bit value). They can be thought of as local memory.

4.1.2 - Languages

Machine language can be written in two ways, binary and symbolic.

Example

```
Set R1 to the value of R1 + R2

Addition = 101011
R1       =  00001
R2       =  00010

This can be stored as a 16 bit instruction

[Addition][R1][R2]
1010110000100010
```

After people noticed you can right R, 1, 2 instead of the binary equivalent they also realized that the symbol + could be used as an agreed upon symbol for addition. This is when the whole ideal of symbolic programming came from.

4.1.3 - Instructions

Arithmetic and logic operations:

```
Load R1, 17      // R1 <- 17
Load R2, 4       // R2 <- 4
Add  R1, R1, R2  // R1 <- R1 + R2

Load R1, true    // R1 <- 17
Load R2, false   // R2 <- 4
And  R1, R1, R2  // R1 <- R1 & R2 ( bit-wise And )

For these instructions to execute they must first be turned
into machine code. This process is done by an assembler.
```

Memory access:

Every machine allows you to access and manipulate memory.

This is typically done using an address register.

Load A, 17
Load M, 1

M stands for memory address selected by A

Flow Control:

Machine language doesn't always work in a straight line, working through the next line of instructions. There are occasionally jumps that will interrupt the program.

There are set goto instructions i.e. jmp jnz

Symbols:

Symbolic code is easier to write, debug, and maintain

4.2 The Hack Machine Language

Programmers who write low-level code interact with the computer abstractly, through its interface

They are familiar with the hardware of the machine so they know how to properly implement instructions.

4.2.1 - Background

Hack is a 16-bit machine, meaning the CPU and the memory units are designed to process, move, and store, chunks of 16-bit values.

Memory

Data-Memory - Data that is stored in RAM to be manipulated (Read/Write)

Instruction-Memory - Data that is stored in ROM to be used to execute instructions

Both are 16-bit memory and each has a 15-bit address space meaning 2^{15} or 32K 16-bit addresses can be stored

Registers

Hack Instructions are set to manipulate three 16-bit registers
- *Data-register (D)*
- *Address-register (A)*
- *Selected-data-memory-Register (M)*

D - Stores a 16-bit value
A - Address and data register

Addressing

The Hack instruction @XXX sets the hack A register to XXX

It also does two things
1. Makes the RAM register whose address is XXX the selected register (M)
2. It makes the value of the ROM register whose address XXX the selected instruction

Ram[100] 17

```
@17    // A=17
D=A    // D=17
@100  //  Setting M = Ram[100]
M=D   // Ram[100] = 17
```

Branching

Sometimes in code we don't want to execute the next line of code. This is when we would use branching

```
@29 // Selects the ROM[29] register and selects RAM[29] but we don't care
0;JMP // Then jmp to the instruction at ROM[29]
```

This basically makes the next executed instruction the instruction at ROM[29]

```
@52 // Selected ROM[52]/RAM[52]
D;JEQ // Jump to ROM[52] if D==0
```

Variables

XXX in @XXX can be either a constant or a symbol. If the instruction is @23 then register A is set to the value of 23. If the instruction is @x where x is a symbol then the register, A, is set to the value of x

```
Let x=17

@17 // A=17
D=A // D=17
@x  // A=x (some register that is now defined as variable x)
M=D // M(or x) = 17
```

Hack uses R0,...,R15 as a set of primitive variables.

```
@R3 // Get RAM[3]
M=0 // set RAM[3]=0

// sum = sum + x

@sum
D=M
@x
D=D+M
@sum
M=D
```

4.2.2 - Program Example

```
// File: Sum1ToN.asm
// Computes RAM[1]=1+2+3+⋯+RAM[0]
// Usage: put a value>=1 in RAM[0]

    // i = 1
    @i
    M=1
    // sum = 0
    @sum
```

```
    M=0
(LOOP)
    // if (I > R0) goto STOP
    @i
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    // sum = sum + 1
    @sum
    D=M
    @i
    D=D+M
    @sum
    M=D
    // i = i + 1
    @i
    M=M+1
    @LOOP
    0;JMP
(STOP)
    // R1 = sum
    @sum
    D=M
    @R1
    M=D
(END)
    @END
    0;JMP
```

4.2.3 - The Hack Language Specification

A - Instruction:

Symbolic

@xxx

Binary

0*vvvvvvvvvvvvvvv*

C - Instruction

Symbolic

Destination = comp; jump

Binary

111accccccdddjjj

xxxaccccccxxxxxx - c instructions for arithmetic

comp		c	c	c	c	c	c
0		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	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
A==0	A==1						

xxxxxxxxxdddxxx - d instruction for where to store comp

	C[5]=A	C[4]=D	C[3]=M	
dest	d	d	d	Effect
Null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register (reg)

DM	0	1	1	D reg & RAM[A]
A	1	0	0	A reg
AM	1	0	1	A reg & RAM[A]
AD	1	1	0	A reg and D reg
ADM	1	1	1	A & D reg, & RAM[A]

xxxxxxxxxxxxjjj - j for jump instruction

jump	j	j	j	Effect
Null	0	0	0	No jump
JGT	0	0	1	If comp > 0 jump
JEQ	0	1	0	If comp == 0 jump
JGE	0	1	1	If comp >= 0 jump
JLT	1	0	0	If comp < 0 jump
JNE	1	0	1	If comp != 0 jump
JLE	1	1	0	If comp <= 0 jump
JMP	1	1	1	Unconditional jump

The C-instruction:

Declares three things:

- What to compute (The ALU operations)
- Where to store the computation
- And whether or not to jump next

The semantics of the C-instruction are specified in the above notes

Computation specification (comp):

The hack ALU takes in two 16-bit values

One from D reg and the other from A reg (when the a-bit is 0) or M reg (when a-bit is 1).

The 16-bit bus will split and direct outputs to the according inputs of the ALU to configure the arithmetic to be done and where to send output

Destination Specification (dest):

The ALU can store its output in 0, 1, 2, or 3 possible registers. The 3 bit dest value will tell the output where to go.

Jump Specification (jump):

This specifies what to do next. Whether to jump to a different part of the code (ROM memory) or to proceed to the next instruction.

Preventing conflicting uses of the A register:

When you're going to jump the C-instruction must not specify M as M is the value from RAM. You must only jump while the C-instruction has no tie to M.

4.2.4 - Symbols

Three types of symbols

- Predefined symbols - Special memory address (R0,...,R15)
- Label symbols - destinations to goto instructions (XXX)
- Variable Symbols - representing variables @XXX

SP = 0
LCL = 1
ARG = 2
THIS = 3
THAT = 4

Memory Maps - memory blocks bounded by the following numbers

- SCREEN - 16384 (HEX 4000)
- KBD - 24576 (HEX 6000)

Label Symbols use (xxx) notations

4.2.5 - Input/output Handlin

Hack maps input and output by addressing memory within memory maps as specified above

Screen:

Screen for Hack is a black and white screen organized as 256 rows of 512 pixels per row

256 x 512 = 131,072 pixels

Stored in an 8K memory map starting at 16348 (HEX 4000)

Refer to the screen as @SCREEN

RAM[SCREEN + row * 32 + col / 16]

```
@SCREEN  // M=0000000000000000 Gets the first memory location of the screen top-left
M=-1     // M=1111111111111111 blackens the top-left part of screen
```

Keyboard:

Stored in an 16-bit memory map at 24576 (HEX 6000)

Refer to the keyboard as @KBD