4. Machine Language[1]

"Form ever follows function" , Louis Sullivan (architect, 1856-1924)

"Form IS function", Ludwig Mies van der Rohe (architect, 1886-1969)

A computer can be described *constructively*, by laying out its hardware platform and explaining how it is built from low-level chips. A computer can also be described *abstractly*, by specifying and demonstrating its machine language capabilities. And indeed, it is easier to get acquainted with a new computer system by first seeing some low-level programs written in its machine language. This helps us understand not only how to program the computer to do useful things, but also why its hardware was designed in a certain way. With that in mind, this chapter focuses on low-level programming in general, and on the Hack machine language in particular. This will set the stage for the next chapter, where we complete the construction of the Hack computer from the chips that we built in the previous chapters.

A machine language is an agreed-upon formalism, designed to code low-level programs as series of machine instructions. Using these instructions, the programmer can command the processor to perform arithmetic and logic operations, fetch and store values from and to the memory, move values from one register to another, test Boolean conditions, and so on. As opposed to high level languages, whose basic design goals are generality and power of expression, the goal of machine language's design is direct execution in, and total control of, a given hardware platform. Of course, generality, power, and elegance are still desired, but only to the extent that they adhere to the basic requirement of direct execution in hardware.

Machine language is the most profound interface in the overall computer enterprise -- the fine line where hardware and software meet. This is the point where the abstract thoughts of the programmer, as manifested in symbolic instructions, are turned into physical operations performed in silicon. Thus, machine language is construed both a programming tool and an integral part of the hardware platform. In fact, just like we say that the machine language is designed to exploit a given hardware platform, we can say that the hardware platform is designed to fetch, interpret and execute, instructions written in the given machine language.

The chapter begins with a general introduction of machine language programming. Next, we give a detailed specification of the Hack machine language, covering both its binary and symbolic assembly versions. The project that accompanies this chapter deals with writing a couple of machine language programs.

Although most people will never write programs directly in machine language, the study of low-level programming is a pre-requisite to a complete understanding of the computer's anatomy. Also, it is rather fascinating to realize how the most sophisticated software systems are, at bottom, long series of elementary instructions, each specifying a very simple and primitive operation on the underlying hardware.

[1] From *The Elements of Computing Systems*, Nisan & Schocken, MIT Press, forthcoming in 2003, www.idc.ac.il/csd

1. Background

This chapter is language-oriented. Therefore, we can abstract away most of the details of the underlying hardware platform. In particular, in order to give a general description of machine languages, it is sufficient to focus on three main hardware elements only: a *processor*, a *memory*, and a set of *registers*.

1.1 Machines

A *machine language* can be viewed as an agreed-upon formalism, designed to manipulate a *memory* using a *processor* and a set of *registers*.

Memory: The term "memory" refers loosely to the collection of hardware devices designed to store data and instructions. Some computer platforms store data and instructions in the same memory device, while others employ different data and instruction memories, each featuring a separate address space. Conceptually speaking, all memories have the same structure: a continuous array of cells of some fixed width, also called *words* or *locations*, each having a unique *address*. Hence, an individual word (representing either a data item or an instruction) is specified by supplying its address. In what follows we will refer to such individual words using the notations `Memory[address]`, `RAM[address]`, or `M[address]` for brevity.

Processor: The processor, normally called *Central Processing Unit* or *CPU*, is a device capable of performing a fixed set of operations. These typically include arithmetic and logic operations, memory access operations, and control (also called *branching*) operations. The operands of these operations are the current values of registers and selected memory locations. Likewise, the results of the operations can be stored either in registers or in selected memory locations.

Registers: Memory access is a relatively slow operation requiring long instruction formats (an address may require 32 bits). For this reason, most processors are equipped with several registers, each capable of holding a single value. Located in the processor's immediate proximity, the registers serve as a high-speed local memory, allowing the processor to quickly store and retrieve data. This setting enables the programmer to minimize the use of memory access commands, thus speeding up the program's execution. In what follows we will refer to the registers as `R0`, `R1`, `R2`, etc.

1.2 Languages

A machine language program is a series of coded instructions. For example, a typical instruction in a 16-bit computer may be "`1010001100011001`". In order to figure out what this instruction means, we have to know the rules of the game, i.e. the instruction set of the underlying hardware platform. For example, the language may be such that each instruction consists of four 4-bit fields: the left-most field codes a CPU operation, and the remaining fields represent the operation's operands. Thus the above command may code the operation "*set R3 to R1+R9*", depending of course on the hardware specification and the machine language syntax.

Since binary codes are rather cryptic, machine languages are normally specified using both binary codes and symbolic mnemonics (a *mnemonics* is a symbolic label that "stands for" something -- in our case binary codes). For example, the language designer can decide that the operation code

"1010" will be represented by the mnemonic "add", and that the registers of the machine will be symbolically referred to using the symbols R0, R1, R2, …. Using these conventions, one can specify machine language instructions either directly, as "1010001100011001", or symbolically, as, say, "ADD R3,R1,R9".

Taking this symbolic abstraction one step further, we can allow ourselves to not only *read* symbolic notation, but to actually *write* programs using symbolic commands rather then binary instructions. Next, we can use a text processing program to parse the symbolic commands into their underlying fields (mnemonics and operands), translate each field into its equivalent binary representation, and assemble the resulting codes into binary machine instructions. The symbolic notation is called *assembly language*, or simply *assembly*, and the program that translates from assembly to binary is called *assembler*.

Since different computers vary in terms of CPU operations, number and type of registers, and assembly syntax rules, the result is a tower of Babel of machine languages, each with its own obscure syntax. Yet irrespective of this variety, all machine languages support similar sets of generic commands, as we now turn to describe.

1.3 Commands

Arithmetic and logic commands: Every computer is required to perform basic arithmetic operations like addition and subtraction as well as basic Boolean operations like bit-wise negation, bit shifting, etc. Different machines feature different sets and versions of such operations, and different ways to apply them to combinations of registers and selected memory locations. Here are some typical possibilities that can be found in various machines:

```
// In all the examples, x is a user-defined label referring to a certain memory location.

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

ADD R2,x        // R2←R2+x

AND R4,R5,R2    // R4←bit wise "And" of R5 and R2

SUBD x          // D← (D-x) where D is a register

ADD x           // add the value of x to a special register called "accumulator"
```

Memory Access commands: Memory access commands fall into two categories. First, as we have just seen, in some cases arithmetic and logical commands are allowed to operate on selected memory locations. Second, all computers feature explicit *load* and *store* commands, designed to move data between the registers and the memory.

Memory access commands may use several types of *addressing modes* -- ways of specifying the address of the required memory word. As usual, different computers offer different possibilities and different notations, but three memory access modes are almost always supported:

- **Direct addressing**: The most common way to address the memory is to express a specific address or use a symbol that refers to a specific address:

```
LOAD R1,67     // R1←Memory[67]
```

```
// Assume that sum refers to memory address 67
LOAD R1,sum    // R1←Memory[67]
```

- **Immediate addressing**: This form of addressing is used to load constants – i.e. load values that appear in the instruction proper: instead of treating the field that appears in the "load" command as an address, we simply load the value of the field itself into the register.

```
LOADI R1,67  // R1←67
```

- **Indirect addressing**: In this addressing mode the address of the required memory location is not hard-coded into the instruction; instead, the instruction specifies a memory location that holds the required memory address. This addressing mode is used to manage *pointers* in high-level programming languages. For example, consider the high-level command "`x=arr[j]`" where `arr` is an array and `x` and `j` are variables. How can we translate this command into machine language? Well, when the array `arr` is declared and initialized in the high-level program, a memory segment of the correct length is allocated to hold the array data. Second, another memory location, referred to by the symbol `arr`, is allocated to hold the *base address* of the array's segment.

  Now, when the compiler is asked to translate a reference to cell `arr[j]`, it goes through the following process. First, note that the `j`'th entry of the array should be physically stored in a memory location that is at a displacement *j* from the array's base address (assuming, for simplicity, that each array element uses a single word). Hence the address corresponding to the expression `arr[j]` can be easily calculated by adding the value of `j` to the value of `arr`. Thus in the C programming language, for example, a command like `x=arr[j]` can be also expressed as `x=*(arr+j)`, where the notation "`*n`" stands for "the value of `Memory[n]`". When translated into machine language, such commands typically yield the following code (depending on the assembly language syntax):

```
// translation of x=arr[j] or x=*(arr+j):
ADD R2,arr,j   // R2←arr+j
LOAD* R1,R2    // R1←memory[R2]
STR R1,x       // x←R1
```

Flow of control commands: While programs normally execute in a linear fashion, one command after the other, they also include occasional branches to locations other than the next command. Branching serves several purposes including *repetition* (jump backward to the beginning of a loop), *conditional execution* (if a Boolean condition is false, jump forward to the location after the "if-then" clause), and *subroutine calling* (jump to the first command of some other code segment). In order to support these programming constructs, every machine language features means to jump to various locations in the program, both conditionally and unconditionally. In assembly languages, locations in the program can also be given symbolic names, using some syntax for specifying labels. Program 1 illustrates a typical example.

High level

```
// a while loop
while (R1>=0) {
   code segment 1
}
code segment 2
```

Low level

```
// typical translation
beginWhile:
  JNG R1,endWhile // if R1<0 goto endWhile
  here comes the translation of code segment 1
  JMP beginWhile    // goto beginWhile
endWhile:
  here comes the translation of code segment 2
```

PROGRAM 1: High- and low-level branching logic. The syntax of *goto commands* varies from one language to another, but the basic idea is the same.

Unconditional jump commands like "`JMP beginWhile`" specify only the address of the target location. *Conditional jump* commands like "`JNG R1,endWhile`" must also specify a condition, expressed in some way. In some languages the condition is an explicit part of the command, while in others it is a by-product of a previous command. Here are some possible examples (noting again that the commands' syntax is less important than their general spirit):

```
// Assume that the foo label is defined elsewhere in the program (not shown here).

JGE R1,foo          // if R1>=0 then goto foo

SUB R1,R2;JEQ foo   // R1←R1-R2; if (result=0) then goto foo

JZR foo             // if (result of the previous command = 0) then goto foo

JMP foo             // goto foo (unconditionally)
```

* * *

This ends our general and informal introduction of machine languages, and the generic commands that can typically found in various hardware platforms. The next section will be more formal, since it describes one specific machine language -- the native code of the computer that we will build in the next chapter.

2. Hack Machine Language Specification

2.1 Overview

Hack is a typical Von Neumann platform: a 16-bit machine, consisting of a CPU, two separate memory modules serving as instruction memory and data memory, and two memory-mapped I/O devices: a screen and a keyboard.

Memory Address Spaces: The Hack programmer is aware of two distinct memory address spaces: an *instruction memory* and a *data memory*. Both memories are 16-bit wide and have a 15-bit address space, meaning that the maximum size of each memory is 32K 16-bit words.

The CPU can only execute programs that reside in the instruction memory. The instruction memory is a read-only device, and thus programs are loaded into it using some exogenous means. For example, the instruction memory can be implemented in a ROM chip which is pre-burned with the required program. Loading a new program can be done by replacing the entire ROM chip (similar to replacing a cartridge in a game computer). In order to simulate this operation, hardware simulators of the Hack platform must provide means to load the instruction memory from a text file containing a machine language program.

Registers: The Hack programmer is aware of two registers called `D` and `A`. These general-purpose 16-bit registers can be manipulated explicitly by arithmetic and logical instructions, e.g. `A=D-1` or `D=!A` (where "`!`" means 16-bit "`not`"). While `D` is used solely to store data values, `A` doubles as both a data register and an address register. That is to say, depending on the instruction context, the contents of `A` can be interpreted either as a data value, or as an address in the data memory, or as an address in the instruction memory, as we now turn to explain.

First, the `A` register can be used to facilitate direct access to the data memory (which, from now on, will be often referred to as "memory"). As the next section will describe, the syntax of the Hack language is such that memory access instructions do not specify an explicit address. Instead, they operate on an implicit memory location labeled "`M`", e.g. `D=M+1`. In order to resolve this address, the contract is such that `M` always refers to the memory word whose address is the current value of `A`. For example, if we want to effect the operation `D=Memory[516]-1`, we have to set the `A` register to 516, and then issue the instruction `D=M-1`.

Second, in addition to doubling as a general-purpose register and as an address register for the data memory, the hard working `A` register is also used to facilitate direct access to the instruction memory. As we will see shortly, the syntax of the Hack language is such that jump instructions do not specify a particular address. Instead, the contract is such that any jump operation always affects a jump to the instruction memory word addressed by `A`. For example, if we want to effect the operation "goto 35", we set `A` to 35 and issue a "goto" command. This will cause the computer to fetch the instruction located in InstructionMemory[35] in the next clock cycle.

Example: Since the Hack language is quite self-explanatory, we start with an example. The only non-obvious command in the language is "`@address`", where *address* is either a number or a symbol representing a number. This command simply stores the specified value into the A register. For example, if `sum` refers to memory location 17, then both "`@17`" and "`@sum`" will have the same effect: `A`←`17`.

And now to the example: Suppose we have to add all the numbers between 1 and 100, using repetitive addition. Program 2 gives a C language solution and a possible compilation into the Hack language.

C language

```
// sum the numbers 1...100
int i=1;
int sum=0;
while (i<=100){
   sum+=I;
   i++;
}
```

Hack machine language

```
// sum the numbers 1...100
      @i     // i refers to some mem. loc.
      M=1    // i=1
      @sum   // sum refers to some mem. loc.
      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)
```

PROGRAM 2: C and assembly versions of the same program.

Although the Hack syntax is more accessible than that of typical machine languages, it may still look rather obscure for readers who are not used to low-level programming. In particular, note that every operation involving a memory location requires two Hack commands: one for selecting the address on which we want to operate, and one for specifying the desired operation. Indeed, the Hack language consists of two generic instructions: an *address instruction*, also called *A*-instruction, and a *compute instruction*, also called *C*-instruction. Each instruction has a binary representation, a symbolic representation, and an effect on the computer, as we now turn to specify.

2.2 The *A*-Instruction

The *A*-instruction is used to set the `A` register to a 15-bit value:

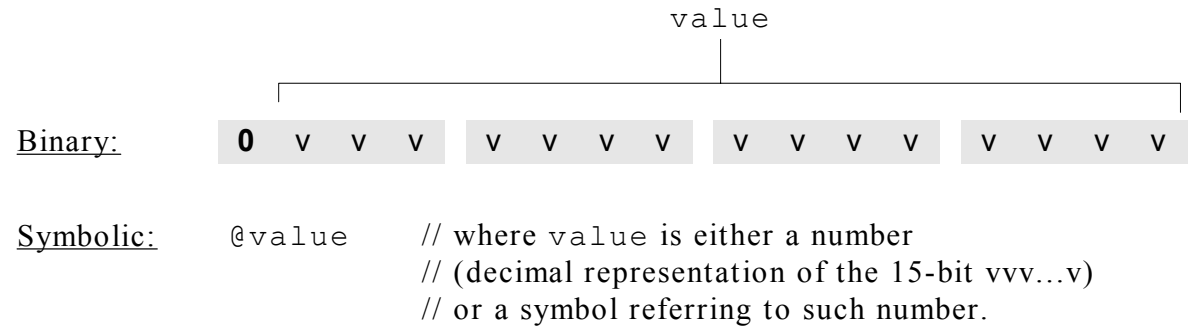

FIGURE 3: *A*-Instruction syntax.

This instruction causes the computer to store a constant in the `A` register. For example, the instruction `@5`, which is equivalent to `0000000000000101`, causes the computer to store the binary representation of 5 in the `A` register.

The *A*-instruction is used for three different purposes. First, it provides the only way to enter a constant into the computer under program control. Second, it sets the stage for a subsequent *C*-instruction designed to manipulate a certain data memory location, by first setting `A` to the address of that location. Third, it sets the stage for a subsequent *C*-instruction that involves a jump, by first loading the address of the jump destination to the `A` register. These uses will be demonstrated below.

2.3 The *C*-Instruction

The *C*-instruction is the programming workhorse of the Hack platform -- the instruction that gets almost everything done. The instruction code is a specification that answers three questions: (a) what to compute? (b) where to store the computed value? and (c) what to do next? Along with the *A*-instruction, these specifications determine all the possible operations of the computer.

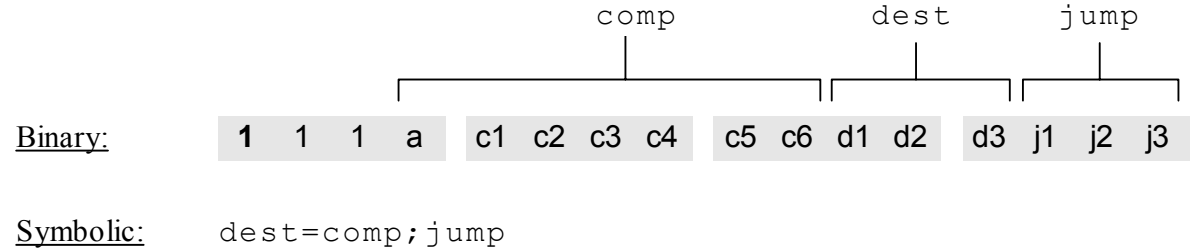

FIGURE 4: *C*-Instruction syntax.

The MSB is the *C*-instruction code, which is 1. The next two bits are not used. The remaining bits form three fields that correspond to the three parts of the instruction's symbolic representation. Taken together, the semantics of the symbolic instruction `dest=comp;jump` is as follow. The `comp` field instructs the CPU what to compute. The `dest` field instructs where to

store the computed value. The `jump` field specifies a jump condition. Either the `dest` field or the `jump` field or both may be empty. If the `dest` field is empty then the "=" sign may be omitted. If the `jump` field is empty then the ";" symbol may be omitted. We now turn to describe the format and semantics of each of the three fields.

The computation specification: The Hack ALU is designed to compute a fixed set of functions on the `D`, `A`, and `M` registers (where `M`=Memory[`A`]). The computed function is specified by the `a`-bit and the six `c`-bits comprising the instruction's `comp` field. This 7-bit pattern can potentially code 128 different functions, of which only the 28 listed in Table 5 are documented in the language specification.

a=0							
mnemonic	c1	c2	c3	c4	c5	c6	
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	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M
	c1	c2	c3	c4	c5	c6	**mnemonic**
a=1							

TABLE 5: The "compute" specification of the *C*-instruction. `D` and `A` are names of registers. `M` refers to the memory location addressed by `A`, i.e. to Memory[`A`]. The symbols "+" and "−" denote 16-bit 2's complement addition and subtraction, while "`!`", "`|`", and "`&`" denote the 16-bit bit-wise Boolean operators `Not`, `Or`, `And`, respectively. Note the similarity between this instruction set and the ALU specification given in Table 7 of Chapter 2.

Recall that the format of the *C*-instruction is "`111a cccc ccdd djjj`". Suppose we want to compute `D-1`, i.e. "the current value of the `D` register minus 1". According to Table 5, this can be

done by issuing the instruction "1110 0011 10xx xxxx" (we use "x" to label bits that are irrelevant to the given example). To compute the value of D|M, we issue the instruction "1111 0101 01xx xxxx". To compute the constant -1, we issue the instruction "1110 1110 10xx xxxx", and so on.

The destination specification: The value computed by the comp part of the *C*-instruction can be simultaneously stored in several destinations, as specified by the instruction's dest part. The first and second d-bits code whether to store the computed value in the A register and in the D register, respectively. The third d-bit codes whether to store the computed value in M (i.e. in Memory[A]). One, more than one, or none of these bits may be asserted.

d1	d2	d3	*mnemonic*	*destination (where to store the computed value)*
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

TABLE 6: The "destination" specification of the *C*-instruction.

Recall that the format of the *C*-instruction is "111a cccc ccdd djjj". Suppose we want the computer to increment the value of Memory[7] by 1, and also store the result in the D register. According to tables 5 and 6, this can be accomplished by the instructions:

```
0000 0000 0000 0111    // @7
1111 1101 1101 1xxx    // DM=M+1  (x=irrelevant bits)
```

The *A*-instruction causes the computer to select the memory register whose address is 7 (the so called "M register"). The subsequent *C*-instruction computes the value of M+1 and stores the result in both D and M. The role of the 3 LSB bits of the second instruction is explained next.

The jump specification: The jump field of the *C*-instruction tells the computer what to do next. There are two possibilities: the computer should either fetch and execute the next instruction in the program, which is the default, or it should fetch and execute an instruction located elsewhere in the program. In the latter case, we assume that the A register has been previously set to the address to which we want to jump.

The jump itself is performed conditionally according to the value computed in the “comp” part of this instruction. The first j-bit specifies whether to jump in case this value is negative, the second j-bit in case the value is zero, and the third j-bit in case it is positive. This gives 8 possible jump conditions.

j1 ($out \prec 0$)	**j2** ($out = 0$)	**j3** ($out \succ 0$)	**Mnemonic**	**Effect**
0	0	0	`null`	no jump
0	0	1	`JGT`	if $out \succ 0$ jump
0	1	0	`JEQ`	if $out = 0$ jump
0	1	1	`JGE`	if $out \geq 0$ jump
1	0	0	`JLT`	if $out \prec 0$ jump
1	0	1	`JNE`	if $out \neq 0$ jump
1	1	0	`JLE`	if $out \leq 0$ jump
1	1	1	`JMP`	jump

TABLE 7: The "jump" specification of the *C*-instruction. *Out* refers to the value computed by the instruction's `comp` part, and *jump* implies "continue execution with the instruction addressed by the `A` register".

The following example illustrates the jump commands in action:

Logic

```
if Memory[3]=5 then
   goto 100
else goto 200
```

Implementation

```
@3
D=M    // D=Memory[3]
@5
D=D-A  // D=D-5
@100
D;JEQ  // if D=0 goto 100
@200
0;JMP  // goto 200
```

The last instruction ("`0;JMP`") effects an unconditional jump. Since the *C*-instruction syntax requires that we always effect *some* computation, we instruct the ALU to compute 0 (an arbitrary choice), which is ignored.

Conflicting uses of the A register: As was just illustrated, the programmer can use the `A` register in order to select either a *data memory* location for a subsequent *C*-instruction involving `M`, or an *instruction memory* location for a subsequent *C*-instruction involving a jump. Thus, in order to prevent conflicting use of the `A` register, we require that in well written programs, *a C-instruction that may cause a jump (i.e. with some non-zero j bits) should not contain a reference to* `M`.

2.4 Symbols

Assembly commands can refer to memory locations (addresses) using either constants or *symbols*. Symbols are introduced into assembly programs in three ways:

- **Predefined symbols**: A special subset of RAM (data memory) addresses can be referred to by any assembly program using pre-defined symbols, as follows.
  - **Virtual registers**: the symbols `R0` to `R15` are pre-defined to refer to RAM addresses 0 to 15, respectively. This syntactic convention is designed to simplify assembly programming.
  - **VM pointers**: the symbols `SP`, `LCL`, `ARG`, `THIS`, and `THAT` are pre-defined to refer to RAM addresses 0 to 4, respectively. Note that each of these memory locations has two labels, e.g. address 2 can be referred to using either `R2` or `ARG`. This syntactic convention will come to play in the implementation of the virtual machine, discussed in Chapters 7 and 8.
  - **I/O Pointers**: the symbols `SCREEN` and `KBD` are pre-defined to refer to RAM addresses 16384 (0x4000) and 24576 (0x6000), respectively, which are the base addresses of the screen and keyboard memory maps. The use of these I/O devices is explained below.
- **Label symbols:** These user-defined symbols, which serve to label destinations of *goto* commands, are 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. A label can be defined only once and can be used anywhere in the assembly program, even before the line in which it is defined.
- **Variable symbols**: Any user-defined symbol `Xxx` appearing in an assembly program that is not predefined and is not defined elsewhere using the "`(Xxx)`" command is treated as a *variable*, and is mapped by the assembler to an available RAM location. Variables are mapped, as they are first encountered, to consecutive memory locations starting at RAM address 16 (0x0010).

2.5 Input / Output Handling

The Hack platform can be connected to two peripheral devices: a screen and a keyboard. Both devices interact with the computer platform through *memory maps*. This means that drawing pixels on the screen is achieved by writing binary values into a memory segment associated with the screen. Likewise, "listening" to the keyboard is done by reading a memory location associated with the keyboard. The physical I/O devices and their memory maps are synchronized via continuous refresh loops.

Screen: The Hack computer can be connected to a black-and-white screen organized as 256 rows of 512 pixels per row. The screen's contents are represented by an 8K memory map that starts at RAM address 16384 (0x4000). Each row in the physical screen, starting at the screen's top left corner, is represented in the RAM by 32 consecutive 16-bit words. Thus the pixel at row *r* from the top and column *c* from the left is mapped on the c%16 bit (counting from LSB to

MSB) of the word located at `RAM[16384+r*32+c/16]`. To write or read a pixel of the physical screen, one reads or writes the corresponding bit in the RAM-resident memory map (1=black, 0=white). Example:

```
// Draw a single black dot at the top left corner of the screen:
@SCREEN     // Set the A register to point to the memory word that is mapped
            // to the 16 left-most pixels of the top row of the screen
M=1         // Blacken the left-most pixel
```

Keyboard: The Hack computer interfaces with the physical keyboard via a single-word memory map located in RAM address 24576 (0x6000). Whenever a key is pressed on the physical keyboard, its 16-bit ASCII code appears in `RAM[24576]`. When no key is pressed, the code 0 appears in this location. In addition to the usual ASCII codes, the Hack keyboard recognizes the following keys:

Key pressed	Code	Key pressed	Code
new line	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
right arrow	131	insert	138
up Arrow	132	delete	139
down arrow	133	esc	140
home	134	f1-f12	141-152

TABLE 8: Special keyboard codes in the Hack language

2.6 Syntax Conventions and Files Format

Binary code files: A binary code file is composed of text lines. Each line is a sequence of 16 "0" and "1" ASCII characters, coding a single machine language instruction. Taken together, all the lines in the file represent a machine language program. The contract is such that when a machine language program is loaded into the computer's instruction memory, the binary code represented by the file's *n*-th line is stored in address *n* of the instruction memory (the count of both program lines and memory addresses starts at 0).

By convention, machine language programs are stored in text files with a "`hack`" extension, e.g. `Prog.hack`.

Assembly language files: By convention, assembly language programs are stored in text files with an "`asm`" extension, e.g. `Prog.asm`. An assembly language file is composed of text lines, each representing either an *instruction* or a *symbol declaration*:

- **Instruction:** an *A*-instruction or a *C*-instruction.
- **(Symbol):** This pseudo-command causes the assembler to assign the label Symbol to the memory location into which the next command in the program will be stored. It is called "pseudo-command" since it generates no machine code.

Constants and symbols in assembly programs: *Constants* must be non-negative and are always written in decimal notation. A user-defined *symbol* can be any sequence of letters, digits, underscore ("_"), dot ("."), dollar sign ("$"), and colon (":") that does not begin with a digit.

Comments in assembly programs: text beginning with two slashes ("//") and ending at the end of the line is considered a comment and is ignored.

White space in assembly programs: space characters are ignored. Empty lines are ignored.

Case conventions: All the assembly mnemonics must be written in upper-case. The rest (user-defined labels and variable names) is case sensitive. The convention is to use upper-case for labels and lower-case for variable names.

3. Perspective

The Hack machine language is almost as simple as machine languages get. Most computers usually have more instructions, more data-types, more registers, more instruction formats, and more addressing modes. At the same time, any feature not supported by the Hack machine language may still be implemented in software, at a performance cost. For example, the Hack platform does not supply multiplication and division as machine-language operations. Since these operations are obviously required by any high-level language, we will later implement them at the operating system level (Chapter 12).

One of the main characteristics that give machine languages their particular flavor is the number of memory addresses that can appear in a single command. In this respect, Hack may be described as a ½-address machine: we usually require two Hack instructions to perform an operation involving a single memory address: an *A*-instruction to specify the address, and a *C*-instruction to specify the operation. In comparison, most machine languages can directly specify at least one address in every machine instruction.

In terms of assembly style, we have chosen to give Hack a somewhat different look-and-feel than the mechanical nature of most assembly languages. In particular, we have chosen a high-level language-like syntax for the *C*-command, e.g. "`D=M`" and "`D=D+M`" instead of the more traditional "`LOAD`" and "`ADD`" commands, respectively. The reader should note however that these are just syntactic details. Further, one can design a macro-Hack language with commands like "`D=M[address]`", "`goto address`", and so on. These macro-commands can be easily translated by the assembler into the sequences "`@address`" followed by "`D=M`" and "`@address`" followed by "`0;jmp`", and so on.

The *assembler*, which was mentioned several times in this chapter, is the program responsible for translating symbolic assembly programs into executable programs, written in binary code. In addition, the assembler is responsible for managing all the system- and user-defined symbols found in the assembly program, and for replacing them with physical addresses of actual memory locations. We will return to this translation task in Chapter 7, in which we build an assembler for the Hack language. But first, we have to complete the construction of the Hack hardware platform, a challenge that is taken up in the next chapter.

4. Build it

Objective: To get a taste of low-level programming in machine language, and to get acquainted with the Hack computer platform. In the process of working on this project, you will get a hands-on understanding of the assembly process, and you will appreciate visually how the translated binary code executes on the target hardware.

Resources: In this project you will use two main tools supplied with the book: an *Assembler*, designed to translate Hack assembly programs into binary code, and a *CPU Emulator*, designed to run binary programs on a simulated Hack platform.

Contract: Write and test the two programs described below. When executed on the CPU Emulator, your programs should generate the results mandated by the supplied test scripts.

- **Multiplication program** (`Mult.asm`): The inputs of this program are the current values stored in `R0` and `R1` (i.e. the two top RAM locations). The program computes the product `R0*R1` and stores the result in `R2`. The algorithm can be iterative addition. We assume (in this program) that `R0`≥ 0, `R1`≥ 0, and `R0*R1`<32768. Your program need not test these conditions, but rather assume that they hold. The supplied `Mult.tst` and `Mult.cmp` scripts will test your program on several representative data values.

- **I/O-Handling Program** (`Fill.asm`): This program runs an infinite loop that "listens" to the keyboard input. When a key is pressed (any key), the program blackens the screen, i.e. writes "black" in every pixel. When no key is pressed, the screen should be cleared. You may choose to blacken and clear the screen in any order, as long as pressing a key continuously for long enough will result in a fully blackened screen and not pressing any key for long enough will result in a cleared screen. Note: this program has a test script (`Fill.tst`) but no compare file – it should be checked by visibly inspecting the simulated screen.

Steps: We recommend proceeding as follows:

1. The *Assembler* and *CPU Emulator* programs needed for this project are available in the *tools* directory of the book software suite.
2. Go through the *Assembler Tutorial* and the *CPU Emulator Tutorial.*
3. Download the `project4.zip` file and extract its contents to a directory called `project4` on your computer. This will create two directories called `project4/mult` and `project4/fill`. As a rule, all the files related to each program (`.asm`, `.hack`, `.tst` and `.cmp`) must be stored in the same directory.
4. Use a text editor to write the first program in assembly, and save it as `.../mult/Mult.asm`.
5. Use the supplied Assembler (in either batch or interactive mode) to debug and translate your program. The result will be a binary file called `Mult.hack`.
6. Use the supplied CPUemulator to test your `Mult.hack` code. You can begin by loading `Mult.hack` into the simulator and testing it in interactive fashion. Then load the supplied `Mult.tst` script into the simulator, and execute it in order to run our "certified test".
7. Repeat stages 4-6 for the second program (`Fill.asm`).