executing the instruction load A,200 and then entering a loop that executes the instructions load M,1 and add A,A,1 fifty times.

Flow control: While computer programs execute by default sequentially, one instruction after another, they also include occasional *jumps* to locations other than the next instruction. To facilitate such branching actions, machine languages feature several variants of conditional and unconditional *goto* instructions, as well as label declaration statements that mark the goto destinations. Figure 4.1 illustrates a simple branching action using machine language.

Using physical addresses

```
...
// Sets R1 to 0+1+2, ...
12: load R1,0
13: add R1,R1,1
...
27: goto 13
...
```

Using symbolic addresses

```
...
// Sets R1 to 0+1+2, ...
   load R1,0
(LOOP)
   add R1,R1,1
   ...
   goto LOOP
   ...
```

Figure 4.1 Two versions of the same low-level code (it is assumed that the code includes some loop termination logic, not shown here).

Symbols: Both code versions in figure 4.1 are written in assembly language; thus, both must be translated into binary code before they can be executed. Also, both versions perform exactly the same logic. However, the code version that uses symbolic references is much easier to write, debug, and maintain.

Further, unlike the code that uses physical addresses, the translated binary version of the code that uses symbolic references can be loaded into, and executed from, any memory segment that happens to be available in the computer's memory. Therefore, low-level code that mentions no physical addresses is said to be *relocatable*. Clearly, relocatable code is essential in computer systems like PCs and cell phones, which routinely load and execute multiple apps dynamically and simultaneously. Thus, we see that

symbolic references are not just a matter of cosmetics—they are used to liberate the code from unnecessary physical attachments to the host memory.

This ends our brief introduction to some machine language essentials. The next section gives a formal description of one specific machine language—the native code of the Hack computer.

4.2 The Hack Machine Language

Programmers who write low-level code (or programmers who write compilers and interpreters that generate low-level code) interact with the computer abstractly, through its *interface*, which is the computer's machine language. Although programmers don't have to be aware of all the details of the underlying computer architecture, they should be familiar with the hardware elements that come to play in their low-level programs.

With that in mind, we begin the discussion of the Hack machine language with a conceptual description of the Hack computer. Next, we give an example of a complete program written in the Hack assembly language. This sets the stage for the remainder of this section, in which we give a formal specification of the Hack language instructions.

4.2.1 Background

The design of the Hack computer, which will be presented in the next chapter, follows a widely used hardware paradigm known as the *von Neumann architecture*, named after the computing pioneer John von Neumann. Hack is a 16-bit computer, meaning that the CPU and the memory units are designed to process, move, and store, chunks of 16-bit values.

Memory: As seen in figure 4.2, the Hack platform uses two distinct memory units: a *data memory* and an *instruction memory*. The data memory stores the binary values that programs manipulate. The instruction memory stores the program's instructions, also represented as binary values. Both memories are 16-bit wide, and each has a 15-bit address space. Thus the

maximum addressable size of each memory unit is 2^{15} or 32K 16-bit words (the symbol *K*, abbreviated from *kilo*—the Greek word for *thousand*—is commonly used to stand for the number $2^{10} = 1024$). It is convenient to think about each memory unit as a linear sequence of addressable memory registers, with addresses ranging from 0 to 32K–1.

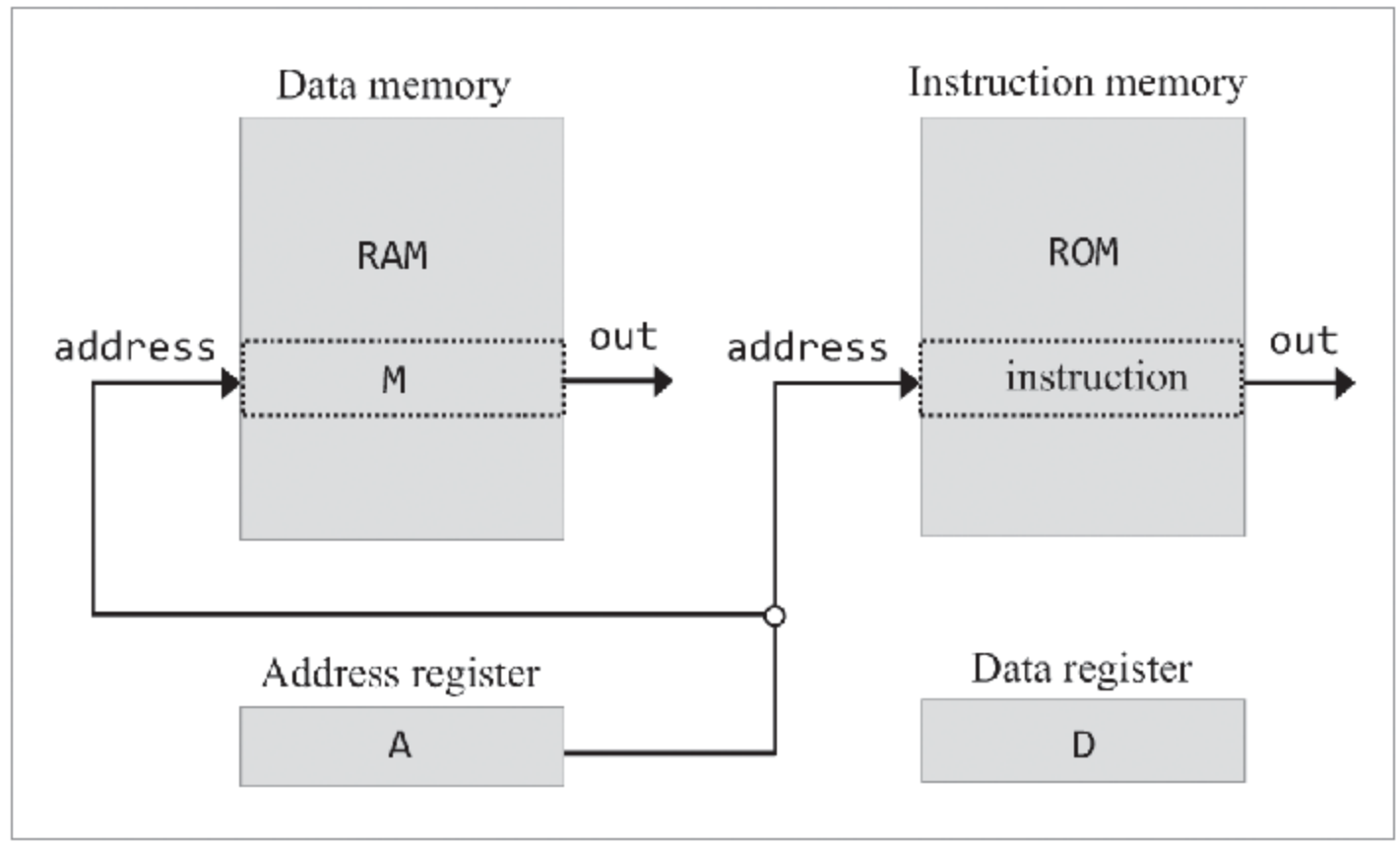


Figure 4.2 Conceptual model of the Hack memory system. Although the actual architecture is wired somewhat differently (as described in chapter 5), this model helps understand the semantics of Hack programs.

The *data memory* (which we also call RAM) is a read/write device. Hack instructions can read data from, and write data to, selected RAM registers. An individual register is selected by supplying its address. Since the memory's `address` input always contains some value, there is always one selected register, and this register is referred to in Hack instructions as `M`. For example, the Hack instruction `M=0` sets the selected RAM register to `0`.

The *instruction memory* (which we also call ROM) is a read-only device, and programs are loaded into it using some exogenous means (more about this in chapter 5). Just like with the RAM, the instruction memory's `address` input always contains some value; thus, there is always one selected instruction memory register. The value of this register is referred to as the *current instruction*.

Registers: Hack instructions are designed to manipulate three 16-bit registers: a *data register*, denoted `D`, an *address register*, denoted `A`, and a selected data memory register, denoted `M`. Hack instructions have a self-explanatory syntax. For example: `D=M`, `M=D+1`, `D=0`, `D=M-1`, and so on.

The role of the data register `D` is straightforward: it serves to store a 16-bit value. The second register, `A`, serves as both an address register and a data register. If we wish to store the value 17 in the `A` register, we use the Hack instruction `@17` (the reason for this syntax will become clear soon). In fact, this is the only way to get a constant into the Hack computer. For example, if we wish to set the `D` register to 17, we use the two instructions `@17`, followed by `D=A`. In addition to serving as a second data register, the hard-working `A` register is also used for addressing the data memory and the instruction memory, as we now turn to discuss.

Addressing: The Hack instruction `@` *xxx* sets the `A` register to the value *xxx*. In addition, the `@` *xxx* instruction has two side effects. First, it makes the RAM register whose address is *xxx* the selected memory register, denoted `M`. Second, it makes the value of the ROM register whose address is *xxx* the selected instruction. Therefore, setting `A` to some value has the simultaneous effect of preparing the stage, potentially, for one of two very different subsequent actions: manipulating the selected data memory register or doing something with the selected instruction. Which action to pursue (and which to ignore) is determined by the subsequent Hack instruction.

To illustrate, suppose we wish to set the value of `RAM[100]` to 17. This can be done using the Hack instructions `@17`, `D=A`, `@100`, `M=D`. Note that in the first pair of instructions, `A` serves as a data register; in the second pair of instructions, it serves as an address register. Here is another example: To set `RAM[100]` to the value of `RAM[200]`, we can use the Hack instructions `@200`, `D=M`, `@100`, `M=D`.

In both of these scenarios, the `A` register also selected registers in the instruction memory—an action which the two scenarios ignored. The next section discusses the opposite scenario: using `A` for selecting instructions while ignoring its effect on the data memory.

Branching: The code examples thus far imply that a Hack program is a sequence of instructions, to be executed one after the other. This indeed is

the default flow of control, but what happens if we wish to branch to executing not the next instruction but, say, instruction number 29 in the program? In the Hack language, this can be done using the Hack instruction @29, followed by the Hack instruction 0;JMP. The first instruction selects the ROM[29] register (it also selects RAM[29], but we don't care about it). The subsequent 0;JMP instruction realizes the Hack version of *unconditional branching*: go to execute the instruction addressed by the A register (we'll explain the ;0 prefix later). Since the ROM is assumed to contain the program that we are presently executing, starting at address 0, the two instructions @29 and 0;JMP end up making the value of ROM[29] the next instruction to be executed.

The Hack language also features *conditional branching*. For example, the logic if D==0 goto 52 can be implemented using the instruction @52, followed by the instruction D;JEQ. The semantics of the second instruction is "evaluate D; if the value equals zero, jump to execute the instruction stored in the address selected by A". The Hack language features several such *conditional branching* commands, as we'll explain later in the chapter.

To recap: The A register serves two simultaneous, yet very different, addressing functions. Following an @*xxx* instruction, we either focus on the selected data memory register (M) and ignore the selected instruction, or we focus on the selected instruction and ignore the selected data memory register. This duality is a bit confusing, but note that we got away with using one address register to control two separate memory devices (see figure 4.2). The result is a simpler computer architecture and a compact machine language. As usual in our business, simplicity and thrift reign supreme.

Variables: The *xxx* in the Hack instruction @*xxx* can be either a constant or a symbol. If the instruction is @23, the A register is set to the value 23. If the instruction is @x, where x is a symbol that is bound to some value, say 513, the instruction sets the A register to 513. The use of symbols endows Hack assembly programs with the ability to use *variables* rather than physical memory addresses. For example, the typical high-level assignment statement let x=17 can be implemented in the Hack language as @17, D=A, @x, M=D. The semantics of this code is "select the RAM register whose address is the value that is bound to the symbol x, and set this register to 17". Here we

assume that there is an agent who knows how to bind the symbols found in high-level languages, like `x`, to sensible and consistent addresses in the data memory. This agent is the assembler.

Thanks to the assembler, variables like `x` can be named and used in Hack programs at will, and as needed. For example, suppose we wish to write code that increments some counter. One option is to keep this counter in, say, `RAM[30]`, and increment it using the instructions `@30`, `M=M+1`. A more sensible approach is to use `@count`, `M=M+1`, and let the assembler worry about where to put this variable in memory. We don't care about the specific address so long as the assembler will always resolve the symbol to that address. In chapter 6 we'll learn how to develop an assembler that implements this useful mapping operation.

In addition to the symbols that can be introduced into Hack assembly programs as needed, the Hack language features sixteen built-in symbols named `R0`, `R1`, `R2`, …, `R15`. These symbols are always bound by the assembler to the values 0, 1, 2, …, 15. Thus, for example, the two Hack instructions `@R3`, `M=0` will end up setting `RAM[3]` to `0`. In what follows, we sometimes refer to `R0`, `R1`, `R2`, …, `R15` as *virtual registers*.

Before going on, we suggest you review, and make sure you fully understand, the code examples shown in figure 4.3 (some of which were already discussed).

Memory access examples

```
// D = 17
@17
D=A

// RAM[100] = 17
@17
D=A
@100
M=D

// RAM[100] = RAM[200]
@200
D=M
@100
M=D
```

Branching examples

```
// goto 29
@29
0;JMP

// if D>0 goto 63
@63
D;JGT
```

Variables use examples

```
// x = -1
@x
M=-1

// count = count - 1
@count
M=M-1

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

Figure 4.3 Hack assembly code examples.

4.2.2 Program Example

Jumping into the cold water, let's review a complete Hack assembly program, deferring a formal description of the Hack language to the next section. Before we do so, a word of caution: Most readers will probably be mystified by the obscure style of this program. To which we say: Welcome to machine language programming. Unlike high-level languages, machine languages are not designed to please programmers. Rather, they are designed to control a hardware platform, efficiently and plainly.

Suppose we wish to compute the sum $1+2+3+\cdots+n$, for a given value n. To operationalize things, we'll put the input n in RAM[0] and the output sum in RAM[1]. The program that computes this sum is listed in figure 4.4. Beginning with the pseudocode, note that instead of utilizing the well-known formula for computing the sum of an arithmetic series, we use brute-force addition. This is done for illustrating conditional and iterative processing in the Hack machine language.

Pseudocode

```
// Program: Sum1ToN
// Computes RAM[1]=1+2+3+...+RAM[0]
// Usage: put a value>=1 in RAM[0]
   i = 1
   sum = 0
LOOP:
   if (i > R0) goto STOP
   sum = sum + i
   i = i + 1
   goto LOOP
STOP:
   R1 = sum
```

Hack assembly code

```
// 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 + i
   @sum
   D=M
   @i
   D=D+M
   @sum
   M=D
   // i = i + 1
   @i
   M=M+1
   // goto LOOP
   @LOOP
   0;JMP
(STOP)
   // R1 = sum
   @sum
   D=M
   @R1
   M=D
(END)
   @END
   0;JMP
```

Figure 4.4 A Hack assembly program (example). Note that RAM[0] and RAM[1] can be referred to as R0 and R1.

Later in the chapter you will understand this program completely. For now, we suggest ignoring the details, and observing instead the following pattern: In the Hack language, every operation involving a memory location entails two instructions. The first instruction, @*addr*, is used to select a target memory address; the subsequent instruction specifies what to do at this address. To support this logic, the Hack language features two generic instructions, several examples of which we have already seen: an *address instruction*, also called *A*-instruction (the instructions that start with @), and a *compute instruction*, also called *C*-instruction (all the other instructions).

Each instruction has a symbolic representation, a binary representation, and an effect on the computer, as we now turn to describe.

4.2.3 The Hack Language Specification

The Hack machine language consists of two instructions, specified in figure 4.5.

A-instruction:

Symbolic: `@`*xxx* (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

Binary: `0` *vvvvvvvvvvvvvvv* (*vv ... v* = 15-bit value of *xxx*)

C-instruction:

Symbolic: *dest* = *comp* ; *jump* (*comp* is mandatory. If *dest* is empty, the = is omitted; If *jump* is empty, the ; is omitted)

Binary: `111`*accccccdddjjj*

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

dest	*d*	*d*	*d*	Effect: store *comp* in:
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 and RAM[A]
A	1	0	0	A reg
AM	1	0	1	A reg and RAM[A]
AD	1	1	0	A reg and D reg
ADM	1	1	1	A reg, D reg, and RAM[A]

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

Figure 4.5 The Hack instruction set, showing symbolic mnemonics and their corresponding binary codes.

The *A*-instruction

The *A*-instruction sets the A register to some 15-bit value. The binary version consists of two fields: an operation code, also known as *op-code*, which is 0 (the leftmost bit), followed by fifteen bits that code a nonnegative binary number. For example, the symbolic instruction @5, whose binary version is 0000000000000101, stores 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 that manipulates a selected RAM register, referred to as `M`, by first setting `A` to the address of that register. Third, it sets the stage for a subsequent *C*-instruction that specifies a jump by first setting `A` to the address of the jump destination.

The *C*-instruction

The *C*-instruction answers three questions: what to compute (an ALU operation, denoted *comp*), where to store the computed value (*dest*), and what to do next (*jump*). Along with the *A*-instruction, the *C*-instruction specifies all the possible operations of the computer.

In the binary version, the leftmost bit is the *C*-instruction's op-code, which is `1`. The next two bits are not used, and are set by convention to `1`. The next seven bits are the binary representation of the *comp* field. The next three bits are the binary representation of the *dest* field. The rightmost three bits are the binary representation of the *jump* field. We now describe the syntax and semantics of these three fields.

Computation specification (*comp*): The Hack ALU is designed to compute one out of a fixed set of functions on two given 16-bit inputs. In the Hack computer, the two ALU data inputs are wired as follows. The first ALU input feeds from the `D` register. The second ALU input feeds either from the `A` register (when the `a`-bit is 0) or from `M`, the selected data memory register (when the `a`-bit is 1). Taken together, 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 calculations, of which only the twenty-eight listed in figure 4.5 are documented in the language specification.

Recall that the format of the *C*-instruction is `111accccccdddjjj`. Suppose we want to compute the value of the `D` register, minus 1. According to figure 4.5, this can be done using the symbolic instruction `D-1`, which is `1110001110000000` in binary (the relevant 7-bit *comp* field is underlined for emphasis). To compute a bitwise Or between the values of the `D` and `M` registers, we use the instruction `D|M` (in binary: `1111010101000000`). To compute the constant –1, we use the instruction `-1` (in binary: `1110111010000000`), and so on.

Destination specification (*dest*): The ALU output can be stored in zero, one, two, or three possible destinations, simultaneously. 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, the currently selected memory register. One, more than one, or none of these three bits may be asserted.

Recall that the format of the *C*-instruction is 111accccccdddjjj. Suppose we wish to increment the value of the memory register whose address is 7 and also to store the new value in the D register. According to figure 4.5, this can be accomplished using the two instructions:

```
0000000000000111 // @7
1111110111011000 // DM=M+1
```

Jump directive (*jump*): The *jump* field of the *C*-instruction specifies what to do next. There are two possibilities: fetch and execute the next instruction in the program, which is the default, or fetch and execute some other, designated instruction. In the latter case, we assume that the A register was already set to the address of the target instruction.

During run-time, whether or not to jump is determined jointly by the three j-bits of the instruction's *jump* field and by the ALU output. The first, second, and third j-bits specify whether to jump in case the ALU output is negative, zero, or positive, respectively. This gives eight possible jump conditions, listed at the bottom right of figure 4.5. The convention for specifying an unconditional goto instruction is 0;JMP (since the *comp* field is mandatory, the convention is to compute 0—an arbitrarily chosen ALU operation—which is ignored).

Preventing conflicting uses of the *A* register: The Hack computer uses one address register for addressing both the RAM and the ROM. Thus, when we execute the instruction @*n*, we select both RAM[*n*] and ROM[*n*]. This is done in order to set the stage for either a subsequent *C*-instruction that operates on the selected data memory register, M, or a subsequent *C*-instruction that specifies a jump. To make sure that we perform exactly one of these two operations, we issue the following best-practice advice: A *C*-

instruction that contains a reference to M should specify no jump, and vice versa: a *C*-instruction that specifies a jump should make no reference to M.

4.2.4 Symbols

Assembly instructions can specify memory locations (addresses) using either constants or symbols. The symbols fall into three functional categories: *predefined symbols*, representing special memory addresses; *label symbols*, representing destinations of goto instructions; and *variable symbols*, representing variables.

Predefined symbols: There are several kinds of predefined symbols, designed to promote consistency and readability of low-level Hack programs.

R0, **R1**, …, **R15**: These symbols are bound to the values 0 to 15. This predefined binding helps make Hack programs more readable. To illustrate, consider the following code segment:

```
// Sets RAM[3] to 7:
@7
D=A
@R3
M=D
```

The instruction @7 sets the A register to 7, and @R3 sets the A register to 3. Why do we use R in the latter and not in the former? Because it makes the code more self-explanatory. In the instruction @7, the syntax hints that A is used as a *data register*, ignoring the side effect of also selecting RAM[7]. In the instruction @R3, the syntax hints that A is used *to select a data memory address*. In general, the predefined symbols R0, R1, …, R15 can be viewed as ready-made working variables, sometimes referred to as *virtual registers*.

SP, **LCL**, **ARG**, **THIS**, **THAT**: These symbols are bound to the values 0, 1, 2, 3, and 4, respectively. For example, address 2 can be selected using either @2, @R2, or @ARG. The symbols SP, LCL, ARG, THIS, and THAT will be used in part II of the book, when we implement the compiler and the virtual

machine that run on top of the Hack platform. These symbols can be completely ignored for now; we specify them for completeness.

SCREEN, KBD: Hack programs can read data from a keyboard and display data on a screen. The screen and the keyboard interface with the computer via two designated memory blocks known as *memory maps*. The symbols SCREEN and KBD are bound, respectively, to the values 16384 and 24576 (in hexadecimal: 4000 and 6000), which are the agreed-upon base addresses of the *screen memory map* and the *keyboard memory map*, respectively. These symbols are used by Hack programs that manipulate the screen and the keyboard, as we'll see in the next section.

Label symbols: Labels can appear anywhere in a Hack assembly program and are declared using the syntax (*xxx*). This directive binds the symbol *xxx* to the address of the next instruction in the program. Goto instructions that make use of label symbols can appear anywhere in the program, even before the label has been declared. By convention, label symbols are written using uppercase letters. The program listed in figure 4.4 uses three label symbols: LOOP, STOP and END.

Variable symbols: Any symbol *xxx* appearing in a Hack assembly program that is not predefined and is not declared elsewhere using (*xxx*) is treated as a variable and is bound to a unique running number starting at 16. By convention, variable symbols are written using lowercase letters. For example, the program listed in figure 4.4 uses two variables: i and sum. These symbols are bound by the assembler to 16 and 17, respectively. Therefore, following translation, instructions like @i and @sum end up selecting memory addresses 16 and 17, respectively. The beauty of this contract is that the assembly program is completely oblivious of the physical addresses. The assembly program uses symbols only, trusting that the assembler will know how to resolve them into actual addresses.

4.2.5 Input/Output Handling

The Hack hardware platform can be connected to two peripheral I/O devices: a screen and a keyboard. Both devices interact with the computer

platform through *memory maps*.

Drawing pixels on the screen is done by writing binary values into a designated memory segment associated with the screen, and listening to the keyboard is done by reading a designated memory location associated with the keyboard. The physical I/O devices and their memory maps are synchronized via continuous refresh loops that are external to the main hardware platform.

Screen: The Hack computer interacts with a black-and-white screen organized as 256 rows of 512 pixels per row. The screen's contents are represented by a memory map, stored in an 8K memory block of 16-bit words, starting at RAM address 16384 (in hexadecimal: 4000), also referred to by the predefined symbol SCREEN. Each row in the physical screen, starting at the screen's top-left corner, is represented in the RAM by thirty-two consecutive 16-bit words. Following convention, the screen origin is the top-left corner, which is considered row 0 and column 0. With that in mind, the pixel at row *row* and column *col* is mapped onto the *col* % 16 bit (counting from LSB to MSB) of the word located at RAM[SCREEN $+ row \cdot 32 + col/16$]. This pixel can be either read (probing whether it is black or white), made black by setting it to 1, or made white by setting it to 0. For example, consider the following code segment, which blackens the first 16 pixels at the top left of the screen:

```
// Sets the A register to the address of the RAM register that represents
// the 16 left-most pixels at the screen's top row:
@SCREEN
// Sets this RAM register to 1111111111111111:
M=-1
```

Note that Hack instructions cannot access individual pixels/bits directly. Instead, we must fetch a complete 16-bit word from the memory map, figure out which bit or bits we wish to manipulate, carry out the manipulation using arithmetic/logical operations (without touching the other bits), and then write the modified 16-bit word to the memory. In the example given above, we got away with not doing bit-specific

manipulations since the task could be implemented using one bulk manipulation.

Keyboard: The Hack computer can interact with a standard physical keyboard via a single-word memory map located at RAM address 24576 (in hexadecimal: 6000), also referred to by the predefined symbol KBD. The contract is as follows: When a key is pressed on the physical keyboard, its 16-bit character code appears at RAM[KBD]. When no key is pressed, the code 0 appears. The Hack character set is listed in appendix 5.

By now, readers with programming experience have probably noticed that manipulating input/output devices using assembly language is a tedious affair. That's because they are accustomed to using high-level statements like write ("hello") or draw Circle (x,y, radius). As you can now appreciate, there is a considerable gap between these abstract, high-level I/O statements and the bit-by-bit machine instructions that end up realizing them in silicon. One of the agents that closes this gap is the *operating system*—a program that knows, among many other things, how to render text and draw graphics using pixel manipulations. We will discuss and write one such OS in part II of the book.

4.2.7 Syntax Conventions and File Formats

Binary code files: By convention, programs written in the binary Hack language are stored in text files with a hack extension, for example, Prog.hack. Each line in the file codes a single binary instruction, using a sequence of sixteen 0 and 1 characters. Taken together, all the lines in the file represent a machine language program. The contract is as follows: When a machine language program is loaded into the computer's instruction memory, the binary code appearing in the file's *n*th line is stored at address *n* of the instruction memory. The counts of program lines, instructions, and memory addresses start at 0.

Assembly language files: By convention, programs written in the symbolic Hack assembly language are stored in text files with an asm extension, for example, Prog.asm. An assembly language file is composed of text lines,

each being an *A*-instruction, a *C*-instruction, a label declaration, or a comment.

A label declaration is a text line of the form (*symbol*). The assembler handles such a declaration by binding *symbol* to the address of the next instruction in the program. This is the only action that the assembler takes when handling a label declaration; no binary code is generated. That's why label declarations are sometimes referred to as *pseudo-instructions*: they exist only at the symbolic level, generating no code.

Constants and symbols: These are the *xxx*'s in *A*-instructions of the form @*xxx*. *Constants* are nonnegative values from 0 to $2^{15}-1$, and are written in decimal notation. A *symbol* is any sequence of letters, digits, underscore (_), dot (.), dollar sign ($), and colon (:) that does not begin with a digit.

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

White space: Leading space characters and empty lines are ignored.

Case conventions: All the assembly mnemonics (figure 4.5) must be written in uppercase. By convention, label symbols are written in uppercase, and variable symbols in lowercase. See figure 4.4 for examples.

4.3 Hack Programming

We now turn to present three examples of low-level programming, using the Hack assembly language. Since project 4 focuses on writing Hack assembly programs, it will serve you well to carefully read and understand these examples.

Example 1: Figure 4.6 shows a program that adds up the values of the first two RAM registers, adds 17 to the sum, and stores the result in the third RAM register. Before running the program, the user (or a test script) is expected to put some values in RAM[0] and RAM[1].

```
// Program: Add.asm
// Computes: RAM[2] = RAM[0] + RAM[1] + 17
// Usage: put values in RAM[0] and in RAM[1]
   // D = RAM[0]
   @R0
   D=M
   // D = D + RAM[1]
   @R1
   D=D+M
   // D = D + 17
   @17
   D=D+A
   // RAM[2] = D
   @R2
   M=D
(END)
   @END
   0;JMP
```

Figure 4.6 A Hack assembly program that computes a simple arithmetic expression.

Among other things, the program illustrates how the so-called virtual registers R0, R1, R2, … can be used as working variables. The program also illustrates the recommended way of terminating Hack programs, which is staging and entering an infinite loop. In the absence of this infinite loop, the CPU's fetch-execute logic (explained in the next chapter) will merrily glide forward, trying to execute whatever instructions are stored in the computer's memory following the last instruction in the current program. This may lead to unpredictable and potentially hazardous consequences. The deliberate infinite loop serves to control and contain the CPU's operation after completing the program's execution.

Example 2: The second example computes the sum $1+2+3+\cdots+n$, where n is the value of the first RAM register, and puts the sum in the second RAM

register. This program is shown in figure 4.4, and now we have what it takes to understand it fully.

Among other things, this program illustrates the use of symbolic variables—in this case i and sum. The example also illustrates our recommended practice for low-level program development: instead of writing assembly code directly, start by writing goto-oriented pseudocode. Next, test your pseudocode on paper, tracing the values of key variables. When convinced that the program's logic is correct, and that it does what it's supposed to do, proceed to express each pseudo-instruction as one or more assembly instructions.

The virtues of writing and debugging symbolic (rather than physical) instructions were observed by the gifted mathematician and writer Augusta Ada King-Noel, Countess of Lovelace, back in 1843. This important insight has contributed to her lasting fame as history's first programmer. Before Ada Lovelace, proto-programmers who worked with early mechanical computers were reduced to tinkering with machine operations directly, and coding was hard and error prone. What was true in 1843 about symbolic and physical programming is equally true today about pseudo and assembly programming: When it comes to nontrivial programs, writing and testing pseudocode and then translating it into assembly instructions is easier and safer than writing assembly code directly.

Example 3: Consider the high-level array processing idiom for `i = 0 ... n` {do something with arr[i]}. If we wish to express this logic in assembly, then our first challenge is that the array abstraction does not exist in machine language. However, if we know the base address of the array in the RAM, we can readily implement this logic in assembly, using pointer-based access to the array elements.

To illustrate the notion of a pointer, suppose that variable x contains the value 523, and consider the two possible pseudo-instructions `x=17` and `*x=17` (of which we execute only one). The first instruction sets the value of x to 17. The second instruction informs that x is to be treated as a *pointer*, that is, a variable whose value is interpreted as a memory address. Hence, the instruction ends up setting RAM[523] to 17, leaving the value of x intact.

The program in figure 4.7 illustrates pointer-based array processing in the Hack machine language. The key instructions of interest are `A=D+M`,

followed by `M=-1`. In the Hack language, the basic pointer-processing idiom is implemented by an instruction of the form `A=...`, followed by a *C*-instruction that operates on `M` (which stands for `RAM[A]`, the memory location selected by A). As we will see when we write the compiler in the second part of the book, this humble low-level programming idiom enables implementing, in Hack assembly, any array access or object-based get/set operation expressed in any high-level language.

Pseudocode

```
// Program: PointerDemo
// Starting at base address R0,
// sets the first R1 words to -1
   n = 0
LOOP:
   if (n == R1) goto END
   *(R0 + n) = -1
   n = n + 1
   goto LOOP
END:
```

Hack assembly code

```
// Program: PointerDemo.asm
// Starting at base address R0,
// sets the first R1 words to -1
    // n = 0
    @n
    M=0
(LOOP)
    // if (n == R1) goto END
    @n
    D=M
    @R1
    D=D-M
    @END
    D;JEQ
    // *(R0 + n) = -1
    @R0
    D=M
    @n
    A=D+M
    M=-1
    // n = n + 1
    @n
    M=M+1
    // goto LOOP
    @LOOP
    0;JMP
(END)
    @END
    0;JMP
```

Figure 4.7 Array processing example, using pointer-based access to array elements.

4.4 Project

Objective: Acquire a taste of low-level programming, and get acquainted with the Hack computer system. This will be done by writing and executing two low-level programs, written in the Hack assembly language.

Resources: The only resources required to complete the project are the Hack *CPU emulator*, available in `nand2tetris/tools`, and the test scripts

described below, available in the projects/04 folder.

Contract: Write and test the two programs described below. When executed on the supplied CPU emulator, your programs should realize the described behaviors.

Multiplication (Mult.asm): The inputs of this program are the values stored in R0 and R1 (RAM[0] and RAM[1]). The program computes the product R0 * R1 and stores the result in R2. Assume that R0 ≥ 0, R1 ≥ 0, and R0 * R1 < 32768 (your program need not test these assertions). The supplied Mult.tst and Mult.cmp scripts are designed to test your program on representative data values.

I/O handling (Fill.asm): This program runs an infinite loop that listens to the keyboard. When a key is pressed (any key), the program blackens the screen by writing *black* in every pixel. When no key is pressed, the program clears the screen by writing *white* in every pixel. You may choose to blacken and clear the screen in any spatial pattern, 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. This program has a test script (Fill.tst) but no compare file—it should be checked by visibly inspecting the simulated screen in the CPU emulator.

CPU emulator: This program, available in nand2tetris/tools, provides a visual simulation of the Hack computer (see figure 4.8). The program's GUI shows the current states of the computer's instruction memory (ROM), data memory (RAM), the two registers A and D, the program counter PC, and the ALU. It also displays the current state of the computer's screen and allows entering inputs through the keyboard.

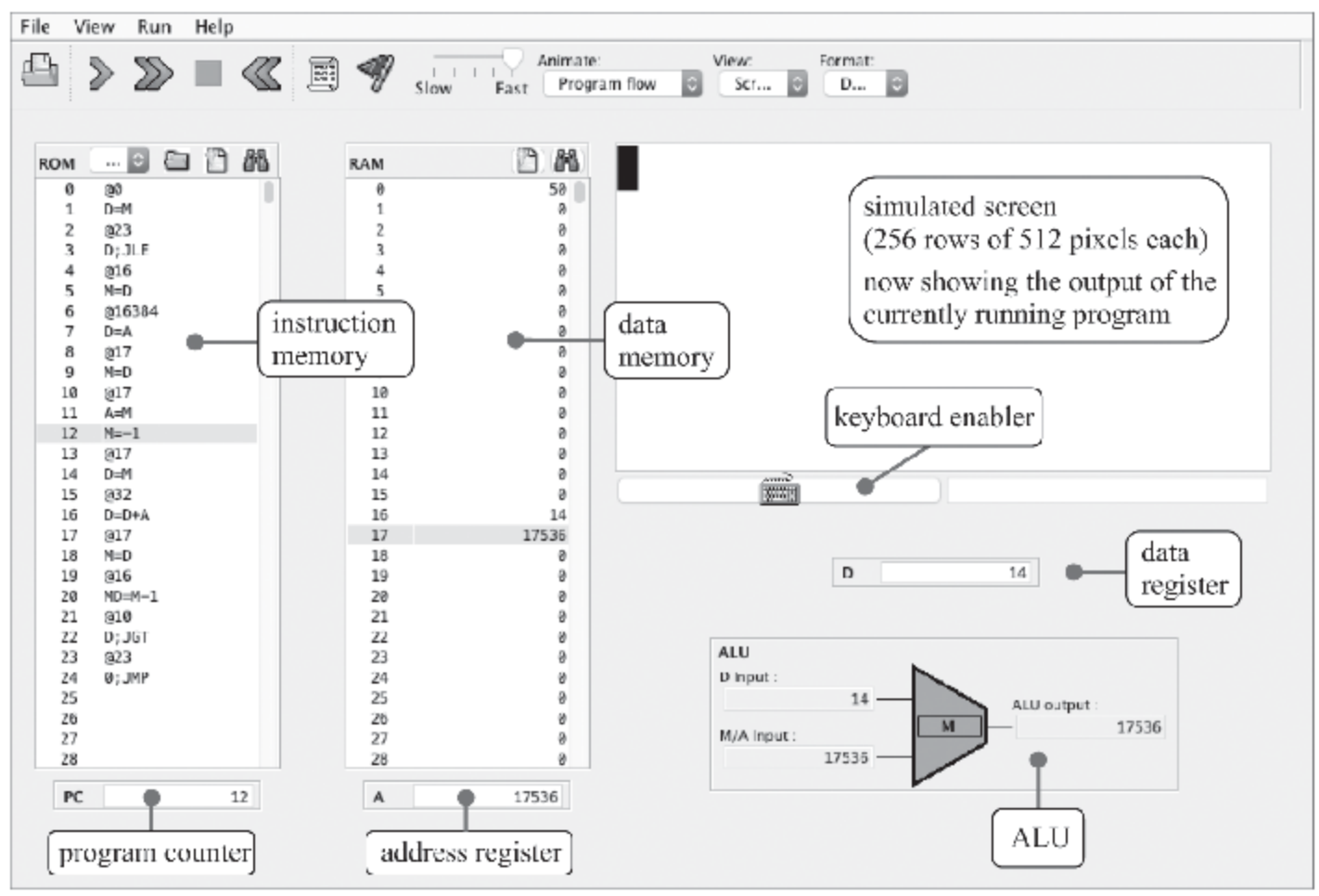


Figure 4.8 The CPU emulator, with a program loaded in the instruction memory (ROM) and some data in the data memory (RAM). The figure shows a snapshot taken during the program's execution.

The typical way to use the CPU emulator is to load a machine language program into the ROM, execute the code, and observe its impact on the simulated hardware elements. Importantly, the CPU emulator enables loading binary.hack files as well as symbolic .asm files, written in the Hack assembly language. In the latter case, the emulator translates the assembly program into binary code on the fly. Conveniently, the loaded code can be viewed in both its binary and symbolic representations.

Since the supplied CPU emulator features a built-in assembler, there is no need to use a standalone Hack assembler in this project.

Steps: We recommend proceeding as follows:

0. The supplied CPU emulator is available in the nand2tetris/tools folder. If you need help, consult the tutorial available at www.nand2tetris.org.
1. Write/edit the Mult.asm program using a plain text editor. Start with the skeletal program stored in projects/04/mult/Mult.asm.
2. Load Mult.asm into the CPU emulator. This can be done either interactively or by loading and executing the supplied Mult.tst script.
3. Run the script. If you get any translation or run-time errors, go to step 1.

Follow steps 1–3 for writing the second program, using the projects/04/fill folder.

Debugging tip: The Hack language is case-sensitive. A common assembly programming error occurs when one writes, say, @foo and @Foo in different parts of the program, thinking that both instructions refer to the same symbol. In fact, the assembler will generate and manage two variables that have nothing in common.

A web-based version of project 4 is available at www.nand2tetris.org.

4.5 Perspective

The Hack machine language is basic. Typical machine languages feature more operations, more data types, more registers, and more instruction formats. In terms of syntax, we have chosen to give Hack a lighter look and feel than that of conventional assembly languages. In particular, we have chosen a friendly syntax for the *C*-instruction, for example, `D=D+M`, instead of the more common prefix syntax `add M,D` used in many machine languages. The reader should note, however, that this is just a syntax effect. For example, the + character in the operation code `D+M` plays no algebraic role whatsoever. Rather, the three-character string `D+M`, taken as a whole, is treated as a single assembly mnemonic, designed to code a single ALU operation.

One of the main characteristics that gives machine languages their particular flavor is the number of memory addresses that can be squeezed into a single instruction. In this respect, the austere Hack language may be described as a 1/2 *address* machine language: Since there is no room to pack both an instruction code and a 15-bit address in a single 16-bit instruction, operations involving memory access require two Hack instructions: one for specifying the address on which we wish to operate, and one for specifying the operation. In comparison, many machine languages can specify an operation and at least one address in every machine instruction.

Indeed, Hack assembly code typically ends up being mostly an alternating sequence of *A*- and *C*-instructions: `@sum` followed by `M=0`, `@LOOP` followed by `0;JMP`, and so on. If you find this coding style tedious or peculiar, you should note that friendlier *macro-instructions* like `sum=0` and `goto LOOP` can be easily introduced into the language, making Hack assembly code shorter and more readable. The trick is to extend the assembler to translate each macro-instruction into the two Hack instructions that it entails—a relatively simple tweak.

The *assembler*, mentioned many 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 resolving them into physical memory addresses that are injected into the generated binary code. We will return to this translation task in chapter 6, which is dedicated to understanding and building assemblers.

---

1. By *instantaneously* we mean within the same clock cycle, or time unit.

5 Computer Architecture

Make everything as simple as possible, but not simpler.
—Albert Einstein (1879–1955)

This chapter is the pinnacle of the hardware part of our journey. We are now ready to take the chips that we built in chapters 1–3 and integrate them into a general-purpose computer system, capable of running programs written in the machine language presented in chapter 4. The specific computer that we will build, named Hack, has two important virtues. On the one hand, Hack is a simple machine that can be constructed in a few hours, using previously built chips and the supplied hardware simulator. On the other hand, Hack is sufficiently powerful to illustrate the key operating principles and hardware elements of any general-purpose computer. Therefore, building it will give you a hands-on understanding of how modern computers work, and how they are built.

Section 5.1 begins with an overview of the *von Neumann architecture*—a central dogma in computer science underlying the design of almost all modern computers. The Hack platform is a von Neumann machine variant, and section 5.2 gives its exact hardware specification. Section 5.3 describes how the Hack platform can be implemented from previously built chips, in particular the ALU built in project 2 and the registers and memory devices built in project 3. Section 5.4 describes the project in which you will build the computer. Section 5.5 provides perspective. In particular, we compare the Hack machine to industrial-strength computers and emphasize the critical role that optimization plays in the latter.

The computer that will emerge from this effort will be as simple as possible, but not simpler. On the one hand, the computer will be based on a

minimal and elegant hardware configuration. On the other hand, the resulting configuration will be sufficiently powerful for executing programs written in a Java-like programming language, presented in part II of the book. This language will enable developing interactive computer games and applications involving graphics and animation, delivering a solid performance and a satisfying user experience. In order to realize these high-level applications on the barebone hardware platform, we will need to build a compiler, a virtual machine, and an operating system. This will be done in part II. For now, let's complete part I by integrating the chips that we've built so far into a complete, general-purpose hardware platform.

5.1 Computer Architecture Fundamentals

5.1.1 The Stored Program Concept

Compared to all the machines around us, the most remarkable feature of the digital computer is its amazing versatility. Here is a machine with a finite and fixed hardware that can perform an infinite number of tasks, from playing games to typesetting books to driving cars. This remarkable versatility—a boon that we have come to take for granted—is the fruit of a brilliant early idea called the *stored program* concept. Formulated independently by several scientists and engineers in the 1930s, the stored program concept is still considered the most profound invention in, if not the very foundation of, modern computer science.

Like many scientific breakthroughs, the basic idea is simple. The computer is based on a fixed hardware platform capable of executing a fixed repertoire of simple instructions. At the same time, these instructions can be combined like building blocks, yielding arbitrarily sophisticated programs. Moreover, the logic of these programs is not embedded in the hardware, as was customary in mechanical computers predating 1930. Instead, the program's code is temporarily stored in the computer's memory, *like data*, becoming what is known as *software*. Since the computer's operation manifests itself to the user through the currently executing software, the same hardware platform can be made to behave completely differently each time it is loaded with a different program.

5.1.2 The von Neumann Architecture

The stored program concept is the key element of both abstract and practical computer models, most notably the *Turing machine* (1936) and the *von Neumann machine* (1945). The Turing machine—an abstract artifact describing a deceptively simple computer—is used mainly in theoretical computer science for analyzing the logical foundations of computation. In contrast, the von Neumann machine is a practical model that informs the construction of almost all computer platforms today.

The von Neumann architecture, shown in figure 5.1, is based on a *Central Processing Unit* (CPU), interacting with a *memory* device, receiving data from some *input* device, and emitting data to some *output* device. At the heart of this architecture lies the *stored program* concept: the computer's memory stores not only the data that the computer manipulates but also the instructions that tell the computer what to do. Let us explore this architecture in some detail.

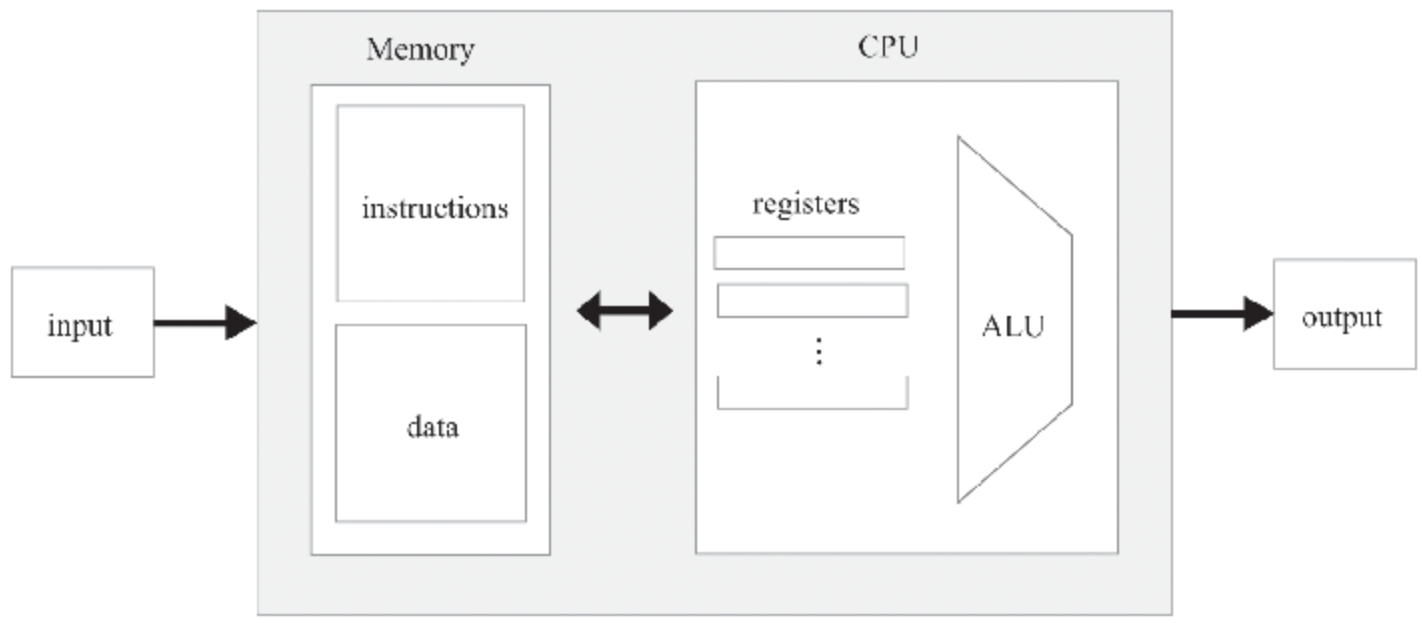


Figure 5.1 A generic von Neumann computer architecture.

5.1.3 Memory

The computer's *Memory* can be discussed from both physical and logical perspectives. Physically, the memory is a linear sequence of addressable, fixed-size *registers*, each having a unique address and a value. Logically, this address space serves two purposes: storing data and storing instructions. Both the "instruction words" and the "data words" are implemented exactly the same way—as sequences of bits.

All the memory registers—irrespective of their roles—are handled the same way: to access a particular memory register, we supply the register's address. This action, also referred to as *addressing*, provides an immediate access to the register's data. The term *Random Access Memory* derives from the important requirement that each randomly selected memory register can be reached instantaneously, that is, within the same cycle (or time step), irrespective of the memory size and the register's location. This requirement clearly carries its weight in memory units that have billions of registers. Readers who built the RAM devices in project 3 know that we've already satisfied this requirement.

In what follows, we'll refer to the memory area dedicated to data as *data memory* and to the memory area dedicated to instructions as *instruction memory*. In some variants of the von Neumann architecture, the data memory and the instruction memory are allocated and managed dynamically, as needed, within the same physical address space. In other variants, the data memory and the instruction memory are kept in two physically separate memory units, each having its own distinct address space. Both variants have pros and cons, as we'll discuss later.

Data memory: High-level programs are designed to manipulate abstract artifacts like variables, arrays, and objects. Yet at the hardware level, these data abstractions are realized by binary values stored in memory registers. In particular, following translation to machine language, abstract array processing and get/set operations on objects are reduced to *reading* and *writing* selected memory registers. To read a register, we supply an address and probe the value of the selected register. To write to a register, we supply an address and store a new value in the selected register, overriding its previous value.

Instruction memory: Before a high-level program can be executed on a target computer, it must first be translated into the machine language of the target computer. Each high-level statement is translated into one or more low-level instructions, which are then written as binary values to a file called the *binary*, or *executable*, version of the program. Before running a program, we must first load its binary version from a mass storage device, and serialize its instructions into the computer's *instruction memory*.

From the pure focus of computer architecture, *how* a program is loaded into the computer's memory is considered an external issue. What's important is that when the CPU is called upon to execute a program, the program's code will already reside in the computer's memory.

5.1.4 Central Processing Unit

The Central Processing Unit (CPU)—the centerpiece of the computer's architecture—is in charge of executing the instructions of the currently running program. Each instruction tells the CPU which computation to perform, which registers to access, and which instruction to fetch and execute next. The CPU executes these tasks using three main elements: An Arithmetic Logic Unit (ALU), a set of registers, and a control unit.

Arithmetic Logic Unit: The ALU chip is built to perform all the low-level arithmetic and logical operations featured by the computer. A typical ALU can add two given values, compute their bitwise And, compare them for equality, and so on. How much functionality the ALU should feature is a design decision. In general, any function not supported by the ALU can be realized later, using system software running on top of the hardware platform. The trade-off is simple: hardware implementations are typically more efficient but result in more expensive hardware, while software implementations are inexpensive and less efficient.

Registers: In the course of performing computations, the CPU is often required to store interim values temporarily. In theory, we could have stored these values in memory registers, but this would entail long-distance trips between the CPU and the RAM, which are two separate chips. These delays would frustrate the CPU-resident ALU, which is an ultra-fast combinational calculator. The result will be a condition known as *starvation*, which is what happens when a fast processor depends on a sluggish data store for supplying its inputs and consuming its outputs.

In order to avert starvation and boost performance, we normally equip the CPU with a small set of high-speed (and relatively expensive) *registers*, acting as the processor's immediate memory. These registers serve various purposes: *data registers* store interim values, *address registers* store values

that are used to address the RAM, the *program counter* stores the address of the instruction that should be fetched and executed next, and the *instruction register* stores the current instruction. A typical CPU uses a few dozen such registers, but our frugal Hack computer will need only three.

Control: A computer instruction is a structured package of agreed-upon micro-codes, that is, sequences of one or more bits designed to signal different devices what to do. Thus, before an instruction can be executed, it must first be decoded into its micro-codes. Next, each micro-code is routed to its designated hardware device (ALU, registers, memory) within the CPU, where it tells the device how to partake in the overall execution of the instruction.

Fetch-Execute: In each step (cycle) of the program's execution, the CPU fetches a binary machine instruction from the instruction memory, decodes it, and executes it. As a side effect of the instruction's execution, the CPU also figures out which instruction to fetch and execute next. This repetitive process is sometimes referred to as the *fetch-execute cycle*.

5.1.5 Input and Output

Computers interact with their external environments using a great variety of input and output (I/O) devices: screens, keyboards, storage devices, printers, microphones, speakers, network interface cards, and so on, not to mention the bewildering array of sensors and activators embedded in automobiles, cameras, hearing aids, alarm systems, and all the gadgets around us. There are two reasons why we don't concern ourselves with these I/O devices. First, every one of them represents a unique piece of machinery, requiring a unique knowledge of engineering. Second, for that very same reason, computer scientists have devised clever schemes for abstracting away this complexity and making all I/O devices look exactly the same to the computer. The key element in this abstraction is called *memory-mapped* I/O.

The basic idea is to create a binary emulation of the I/O device, making it appear to the CPU as if it were a regular linear memory segment. This is done by allocating, for each I/O device, a designated area in the computer's

memory that acts as its memory map. In the case of an input device like a keyboard, the memory map is made to continuously reflect the physical state of the device: when the user presses a key on the keyboard, a binary code representing that key appears in the keyboard's memory map. In the case of an output device like a screen, the screen is made to continuously reflect the state of its designated memory map: when we write a bit in the screen's memory map, a respective pixel is turned on or off on the screen.

The I/O devices and the memory maps are refreshed, or synchronized, many times per second, so the response time from the user's perspective appears to be instantaneous. Programmatically, the key implication is that low-level computer programs can access any I/O device by manipulating its designated memory map.

The memory map convention is based on several agreed-upon contracts. First, the data that drives each I/O device must be serialized, or mapped, onto the computer's memory, hence the name *memory map*. For example, the screen, which is a two-dimensional grid of pixels, is mapped on a one-dimensional block of fixed-size memory registers. Second, each I/O device is required to support an agreed-upon interaction protocol so that programs will be able to access it in a predictable manner. For example, it should be decided which binary codes should represent which keys on the keyboard. Given the multitude of computer platforms, I/O devices, and different hardware and software vendors, one can appreciate the crucial role that agreed-upon industry-wide *standards* play in realizing these low-level interaction contracts.

The practical implications of memory-mapped I/O are significant: The computer system is totally independent of the number, nature, or make of the I/O devices that interact, or may interact, with it. Whenever we want to connect a new I/O device to the computer, all we have to do is allocate to it a new memory map and take note of the map's base address (these onetime configurations are carried out by the so-called *installer* programs). Another necessary element is a *device driver* program, which is added to the computer's operating system. This program bridges the gap between the I/O device's memory map data and the way this data is actually rendered on, or generated by, the physical I/O device.

5.2 The Hack Hardware Platform: Specification

The architectural framework described thus far is characteristic of any general-purpose computer system. We now turn to describe one specific variant of this architecture: the Hack computer. As usual in Nand to Tetris, we start with the abstraction, focusing on *what* the computer is designed to do. The computer's implementation—*how* it does it—is described later.

5.2.1 Overview

The Hack platform is a 16-bit von Neumann machine designed to execute programs written in the Hack machine language. In order to do so, the Hack platform consists 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*.

The Hack computer executes programs that reside in an instruction memory. In physical implementations of the Hack platform, the instruction memory can be implemented as a ROM (Read-Only Memory) chip that is preloaded with the required program. Software-based emulators of the Hack computer support this functionality by providing means for loading the instruction memory from a text file containing a program written in the Hack machine language.

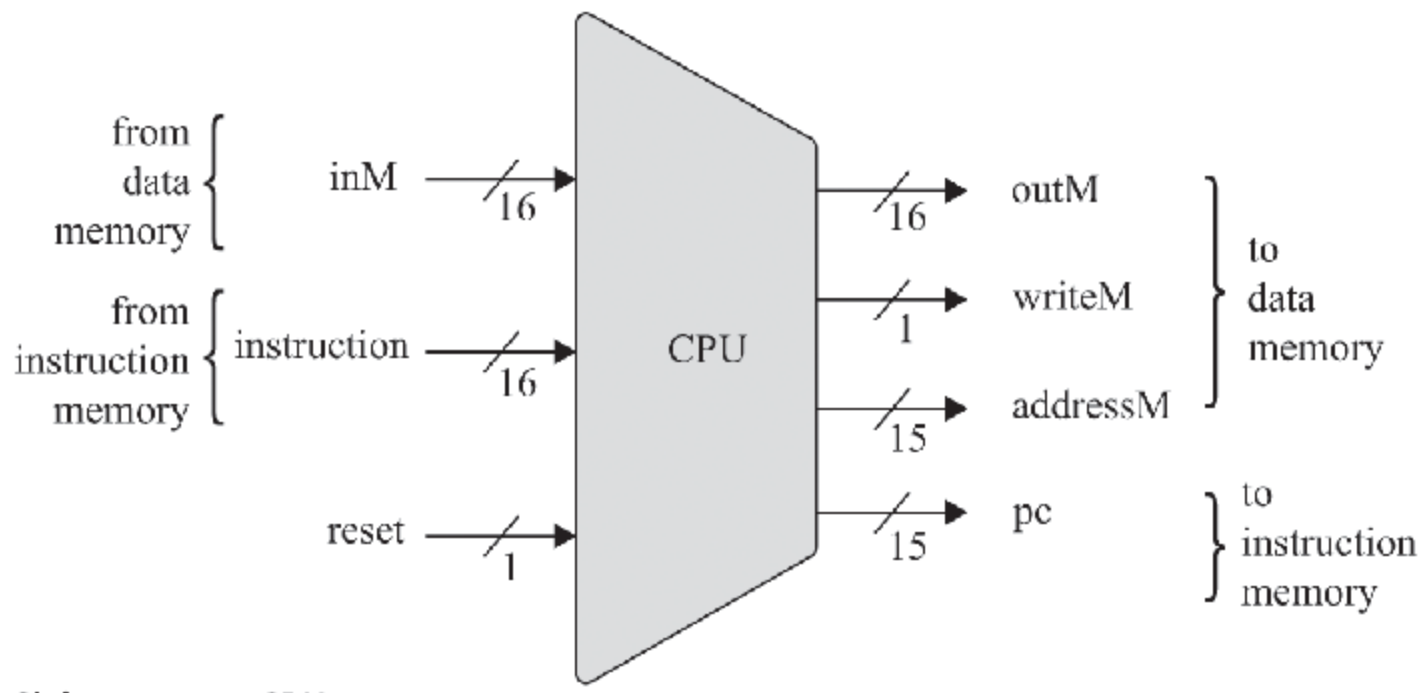
The Hack CPU consists of the ALU built in project 2 and three registers named *Data register* (D), *Address register* (A), and *Program Counter* (PC). The D register and the A register are identical to the Register chip built in project 3, and the program counter is identical to the PC chip built in project 3. While the D register is used solely for storing data values, the A register serves one of three different purposes, depending on the context in which it is used: storing a data value (like the D register), selecting an address in the instruction memory, or selecting an address in the data memory.

The Hack CPU is designed to execute instructions written in the Hack machine language. In case of an *A*-instruction, the 16 bits of the instruction are treated as a binary value which is loaded as is into the A register. In case of a *C*-instruction, the instruction is treated as a capsule of control bits that specify various micro-operations to be performed by various chip-parts

within the CPU. We now turn to describe how the CPU materializes these micro-codes into concrete actions.

5.2.2 Central Processing Unit

The Hack CPU interface is shown in figure 5.2. The CPU is designed to execute 16-bit instructions according to the Hack machine language specification presented in chapter 4. The CPU consists of an ALU, two registers named A and D, and a program counter named PC (these internal chip-parts are not seen outside the CPU). The CPU expects to be connected to an instruction memory, from which it fetches instructions for execution, and to a data memory, from which it can read, and into which it can write, data values. The inM input and the outM output hold the values referred to as M in the *C*-instruction syntax. The addressM output holds the address at which outM should be written.




```
Chip name: CPU
Input:
   instruction[16]   // Instruction to execute
   inM[16]           // The instruction's M input (contents of RAM[A])
   reset             // Signals whether to restart the program (if reset==1)
                     // or continue executing the program (if reset==0).
Output:
   outM[16]          // Written to RAM[addressM], the instruction's M output
   addressM[15]      // At which address to write?
   writeM            // Write to the memory?
   pc[15]            // Address of next instruction
```

Figure 5.2 The Hack Central Processing Unit (CPU) interface.

If the instruction input is an *A*-instruction, the CPU loads the 16-bit instruction value into the A register. If instruction is a *C*-instruction, then (i) the CPU causes the ALU to perform the computation specified by the instruction and (ii) the CPU causes this value to be stored in the subset of {A,D,M} destination registers specified by the instruction. If one of the destination registers is M, the CPU's outM output is set to the ALU output, and the CPU's writeM output is set to 1. Otherwise, writeM is set to 0, and any value may appear in outM.

As long as the reset input is 0, the CPU uses the ALU output and the jump bits of the current instruction to decide which instruction to fetch next. If reset is 1, the CPU sets pc to 0. Later in the chapter we'll connect the CPU's pc output to the address input of the instruction memory chip, causing the latter to emit the next instruction. This configuration will realize the fetch step of the fetch-execute cycle.

The CPU's outM and writeM outputs are realized by *combinational* logic; thus, they are affected instantaneously by the instruction's execution. The addressM and pc outputs are *clocked*: although they are affected by the instruction's execution, they commit to their new values only in the next time step.

5.2.3 Instruction Memory

The Hack *instruction memory*, called ROM32K, is specified in figure 5.3.


```
Chip name:  ROM32K
Input:      address[15]
Output:     out[16]
Function:   Emits the 16-bit value stored in the address selected by the
            address input. It is assumed that the chip is preloaded
            with a program written in the Hack machine language.
```

Figure 5.3 The Hack instruction memory interface.

5.2.4 Input/Output

Access to the input/output devices of the Hack computer is made possible by the computer's *data memory*, a read/write RAM device consisting of 32K addressable 16-bit registers. In addition to serving as the computer's general-purpose data store, the data memory also interfaces between the CPU and the computer's input/output devices, as we now turn to describe.

The Hack platform can be connected to two peripheral devices: a *screen* and a *keyboard*. Both devices interact with the computer platform through designated memory areas called *memory maps*. Specifically, images can be drawn on the screen by writing 16-bit values in a designated memory segment called a *screen memory map*. Similarly, which key is presently pressed on the keyboard can be determined by probing a designated 16-bit memory register called a *keyboard memory map*.

The screen memory map and the keyboard memory map are continuously updated, many times per second, by peripheral refresh logic that is external to the computer. Thus, when one or more bits are changed in the screen memory map, the change is immediately reflected on the physical screen. Likewise, when a key is pressed on the physical keyboard, the character code of the pressed key immediately appears in the keyboard memory map. With that in mind, when a low-level program wants to read something from the keyboard, or write something on the screen, the program manipulates the respective memory maps of these I/O devices.

In the Hack computer platform, the screen memory map and the keyboard memory map are realized by two built-in chips named Screen and Keyboard. These chips behave like standard memory devices, with the additional side effects of continuously synchronizing between the I/O devices and their respective memory maps. We now turn to specify these chips in detail.

Screen: The Hack computer can interact with a physical screen consisting of 256 rows of 512 black-and-white pixels each, spanning a grid of 131,072 pixels. The computer interfaces with the physical screen via a memory map, implemented by an 8K memory chip of 16-bit registers. This chip, named Screen, behaves like a regular memory chip, meaning that it can be read and written to using the regular RAM interface. In addition, the Screen chip

features the side effect that the state of any one of its bits is continuously reflected by a respective pixel in the physical screen (1 = black, 0 = white).

The physical screen is a two-dimensional address space, where each pixel is identified by a row and a column. High-level programming languages typically feature a graphics library that allows accessing individual pixels by supplying (*row*,*column*) coordinates. However, the memory map that represents this two-dimensional screen at the low level is a one-dimensional sequence of 16-bit words, each identified by supplying an address. Therefore, individual pixels cannot be accessed directly. Rather, we have to figure out which word the target bit is located in and then access, and manipulate, the entire 16-bit word this pixel belongs to. The exact mapping between these two address spaces is specified in figure 5.4. This mapping will be realized by the screen driver of the operating system that we'll develop in part II of the book.

```
Chip name: Screen          // Screen memory map
Input:     in[16]          // What to write
           address[13]     // Where to read/write
           load            // Write-enable bit
Output:    out[16]         // Screen value at the given address
Function:  Exactly like a 16-bit, 8K RAM, plus a screen refresh side effect.
    Emits the value stored at the memory location specified by address.
    If load==1, then the memory location specified by address is set to the value of in.

    The loaded value will be emitted by out from the next time step onward.
    In addition, the chip continuously refreshes a physical screen, consisting of 256 rows
    and 512 columns of black and white pixels.

    The pixel at row r from the top and column c from the left (0 ≤ r ≤ 255, 0 ≤ c ≤ 511)
    is mapped onto the c%16 bit (counting from LSB to MSB) of the 16-bit word stored
    in Screen[r * 32 + c / 16].

    (Simulators of the Hack computer are expected to simulate the physical screen,
    the mapping, and the refresh contract).
```

Figure 5.4 The Hack Screen chip interface.

Keyboard: The Hack computer can interact with a physical keyboard, like that of a personal computer. The computer interfaces with the physical keyboard via a memory map, implemented by the Keyboard chip, whose interface is given in figure 5.5. The chip interface is identical to that of a read-only, 16-bit register. In addition, the Keyboard chip has the side effect of reflecting the state of the physical keyboard: When a key is pressed on the

physical keyboard, the 16-bit code of the respective character is emitted by the output of the Keyboard chip. When no key is pressed, the chip outputs 0. The character set supported by the Hack computer is given in appendix 5, along with the code of each character.

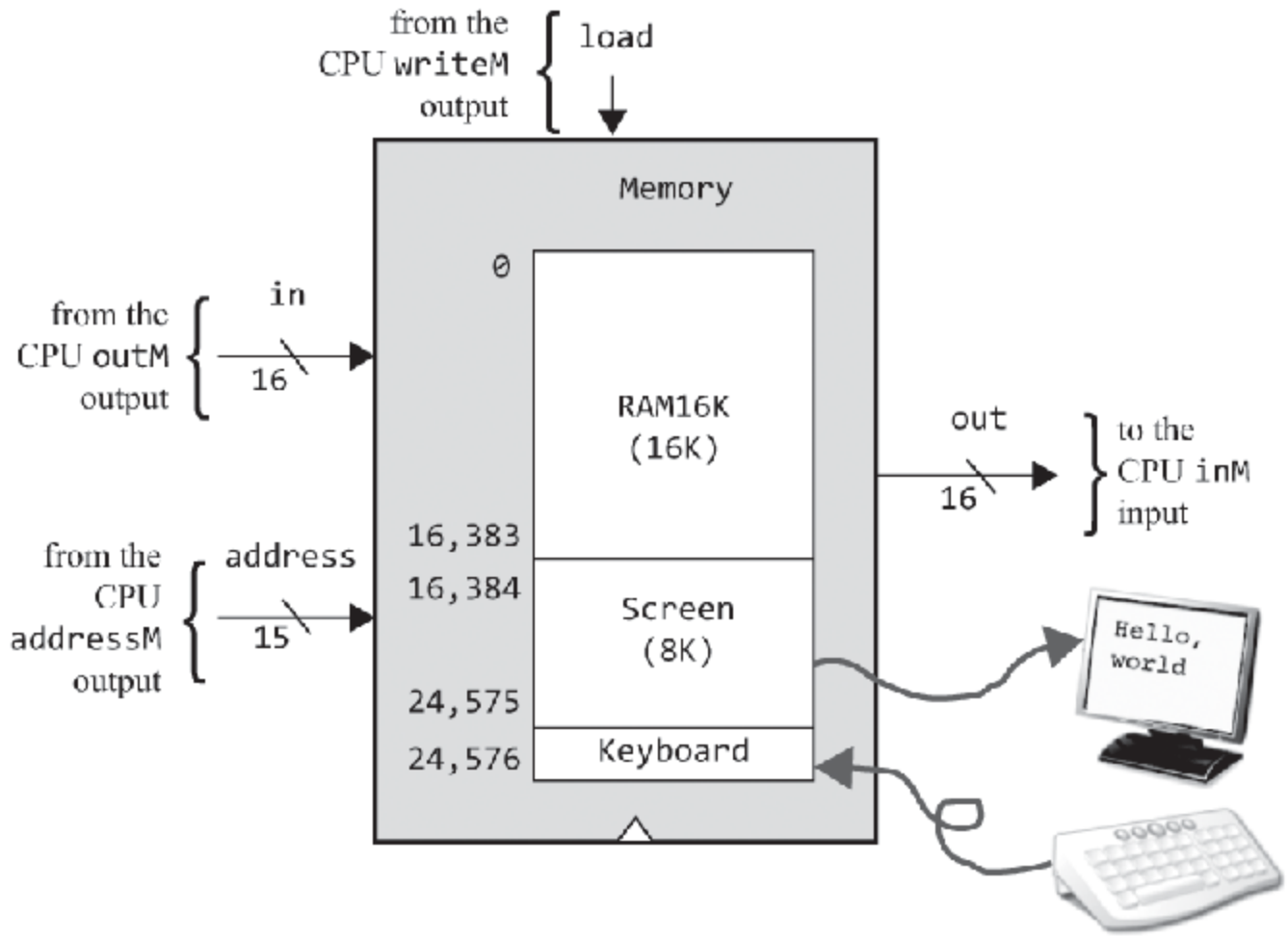
```
Chip name: Keyboard    // Keyboard memory map
Output:    out[16]
Function:  Emits the 16-bit character code of the currently pressed
           key on the physical keyboard or 0 if no key is pressed.
```

(Simulators of the Hack computer are expected to simulate this refresh contract).

Figure 5.5 The Hack Keyboard chip interface.

5.2.5 Data Memory

The overall address space of the Hack *data memory* is realized by a chip named Memory. This chip is essentially a package of three 16-bit chip-parts: RAM16K (a RAM chip of 16K registers, serving as a general-purpose data store), Screen (a built-in RAM chip of 8K registers, acting as the screen memory map), and Keyboard (a built-in register chip, acting as the keyboard memory map). The complete specification is given in figure 5.6.

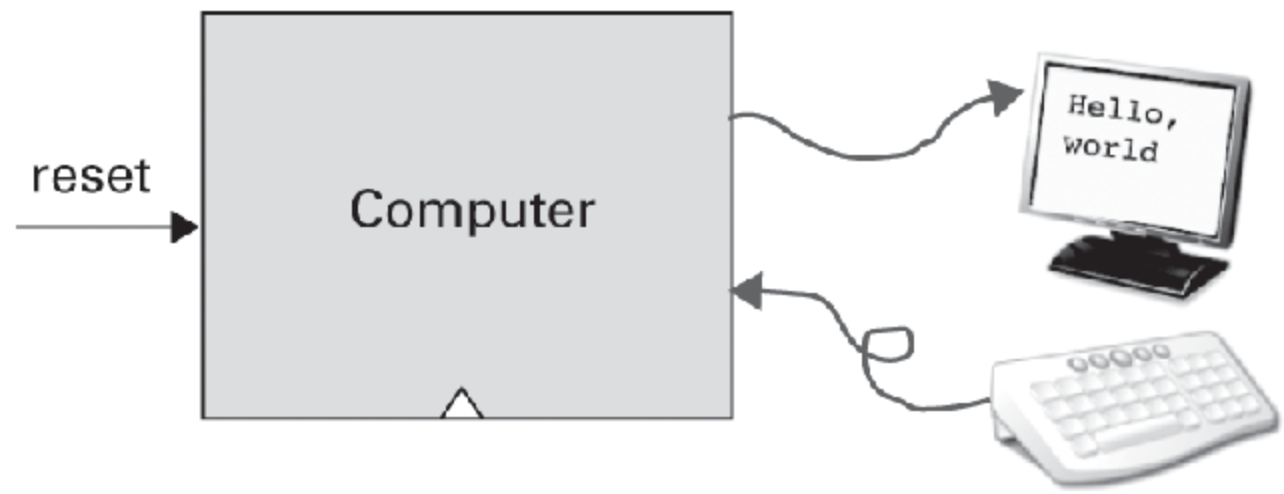



```
Chip name:  Memory          // Data memory
Input:      in[16]          // What to write
            address[15]     // Where to read/write
            load            // Write-enable bit
Output:     out[16]         // Value at the given address
Function:
    The complete address space of the Hack computer's data memory.
    Only the top 16K+8K+1 words of the address space are used.
    Accessing an address in the range 0 - 16383 results in accessing RAM16K;
    Accessing an address in the range 16384 - 24575 results in accessing Screen;
    Accessing the address 24576 results in accessing Keyboard;
    Accessing any other address is invalid.
```

Figure 5.6 The Hack data memory interface. Note that the decimal values 16384 and 24576 are 4000 and 6000 in hexadecimal.

5.2.6 Computer

The topmost element in the Hack hardware hierarchy is a computer-on-a-chip named Computer (figure 5.7). The Computer chip can be connected to a screen and a keyboard. The user sees the screen, the keyboard, and a single bit input named reset. When the user sets this bit to 1 and then to 0, the computer starts executing the currently loaded program. From this point onward, the user is at the mercy of the software.




```
Chip name: Computer
Input:     reset
Function:
```

When `reset==0`, the program stored in the computer executes.
When `reset==1`, the execution of the program restarts.
To start the program's execution, set `reset` to `1`, and then to `0`.
(It is assumed that the computer's instruction memory is loaded with a program written in the Hack machine language).

Figure 5.7 Interface of Computer, the topmost chip in the Hack hardware platform.

This startup logic realizes what is sometimes referred to as "*booting* the computer." For example, when you boot up a PC or a cell phone, the device is set up to run a ROM-resident program. This program, in turn, loads the operating system's kernel (also a program) into the RAM and starts executing it. The kernel then executes a process (yet another program) that listens to the computer's input devices, that is, keyboard, mouse, touch screen, microphone, and so on. At some point the user will do something, and the OS will respond by running another process or invoking some program.

In the Hack computer, the software consists of a binary sequence of 16-bit instructions, written in the Hack machine language and stored in the computer's instruction memory. Typically, this binary code will be the low-level version of a program written in some high-level language and translated by a *compiler* into the Hack machine language. The compilation process will be discussed and implemented in part II of the book.

5.3 Implementation

This section describes a hardware implementation that realizes the Hack computer specified in the previous section. As usual, we don't give exact building instructions. Rather, we expect readers to discover and complete the implementation details on their own. All the chips described below can be built in HDL and simulated on a personal computer using the supplied hardware simulator.

5.3.1 The Central Processing Unit

The implementation of the Hack CPU entails building a logic gate architecture capable of (i) executing a given Hack instruction and (ii) determining which instruction should be fetched and executed next. In order to do so, we will use gate logic for decoding the current instruction, an Arithmetic Logic Unit (ALU) for computing the function specified by the instruction, a set of registers for storing the resulting value, as specified by the instruction, and a program counter for keeping track of which instruction should be fetched and executed next. Since all the underlying building blocks (ALU, registers, PC, and elementary logic gates) were already built in previous chapters, the key question that we now face is how to connect these chip-parts judiciously in a way that realizes the desired CPU operation. One possible configuration is illustrated in figure 5.8 and explained in the following pages.

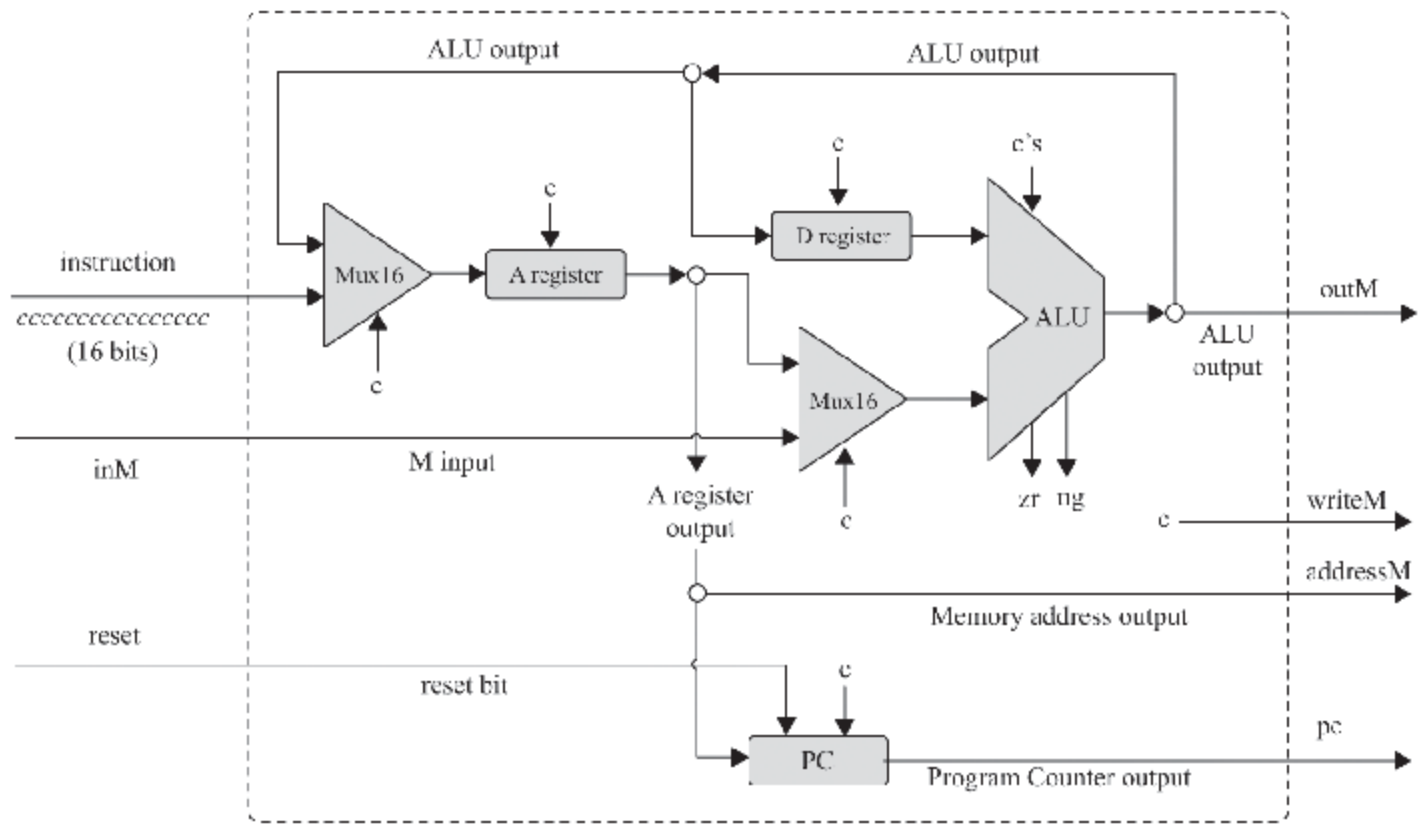


Figure 5.8 Proposed implementation of the Hack CPU, showing an incoming 16-bit instruction. We use the instruction notation *cccccccccccccccc* to emphasize that in the case of a *C*-instruction, the instruction is treated as a capsule of control bits, designed to control different CPU chip-parts. In this diagram, every *c* symbol entering a chip-part stands for some control bit extracted from the instruction (in the case of the ALU, the *c*'s input stands for the six control bits that instruct the ALU what to compute). Taken together, the distributed behavior induced by these control bits ends up executing the instruction. We don't specify which bits go where, since we want readers to answer these questions themselves.

Instruction decoding: Let's start by focusing on the CPU's instruction input. This 16-bit value represents either an *A*-instruction (when the leftmost bit is 0) or a *C*-instruction (when the leftmost bit is 1). In case of an *A*-instruction, the instruction bits are interpreted as a binary value that should be loaded into the A register. In case of a *C*-instruction, the instruction is treated as a capsule of control bits 1xxaccccccdddjjj, as follows. The a and cccccc bits code the *comp* part of the instruction; the ddd bits code the *dest* part of the instruction; the jjj bits code the *jump* part of the instruction. The xx bits are ignored.

Instruction execution: In case of an *A*-instruction, the 16 bits of the instruction are loaded as is into the A register (actually, this is a 15-bit value, since the MSB is the op-code 0). In case of a *C*-instruction, the a-bit determines whether the ALU input will be fed from the A register value or from the incoming M value. The cccccc bits determine which function the ALU will compute. The ddd bits determine which registers should accept the ALU output. The jjj bits are used for determining which instruction to fetch next.

The CPU architecture should extract the control bits described above from the instruction input and route them to their chip-part destinations, where they instruct the chip-parts what to do in order to partake in the instruction's execution. Note that every one of these chip-parts is already designed to carry out its intended function. Therefore, the CPU design is mostly a matter of connecting existing chips in a way that realizes this execution model.

Instruction fetching: As a side effect of executing the current instruction, the CPU determines, and outputs, the address of the instruction that should

be fetched and executed next. The key element in this subtask is the *Program Counter*—a CPU chip-part whose role is to always store the address of the next instruction.

According to the Hack computer specification, the current program is stored in the instruction memory, starting at address 0. Hence, if we wish to start (or restart) a program's execution, we should set the Program Counter to 0. That's why in figure 5.8 the reset input of the CPU is fed directly into the reset input of the PC chip-part. If we assert this bit, we'll effect PC=0, causing the computer to fetch and execute the first instruction in the program.

What should we do next? Normally, we'd like to execute the next instruction in the program. Therefore, and assuming that the reset input had been set "back" to 0, the default operation of the program counter is PC++.

But what if the current instruction includes a jump directive? According to the language specification, execution always branches to the instruction whose address is the current value of A. Thus, the CPU implementation must realize the following Program Counter behavior: if *jump* then PC=A else PC++.

How can we effect this behavior using gate logic? The answer is hinted in figure 5.8. Note that the output of the A register feeds into the input of the PC register. Thus, if we assert the PC's load-bit, we'll enable the operation PC=A rather than the default operation PC++. We should do this only if we have to effect a jump. Which leads to the next question: How do we know if we have to effect a jump? The answer depends on the three j-bits of the current instruction and the two ALU output bits zr and ng. Taken together, these bits can be used to determine whether the jump condition is satisfied or not.

We'll stop here, lest we rob readers of the pleasure of completing the CPU implementation themselves. We hope that as they do so, they will savor the clockwork elegance of the Hack CPU.

5.3.2 Memory

The Memory chip of the Hack computer is an aggregate of three chip-parts: RAM16K, Screen, and Keyboard. This modularity, though, is implicit: Hack

machine language programs see a *single address space*, ranging from address 0 to address 24576 (in hexadecimal: 6000).

The Memory chip interface is shown in figure 5.6. The implementation of this interface should realize the continuum effect just described. For example, if the address input of the Memory chip happens to be 16384, the implementation should access address 0 in the Screen chip, and so on. Once again, we prefer not to provide too many details and let you figure out the rest of the implementation yourself.

5.3.3 Computer

We have reached the end of our hardware journey. The topmost Computer chip can be realized using three chip-parts: the CPU, the data Memory chip, and the instruction memory chip, ROM32K. Figure 5.9 gives the details.

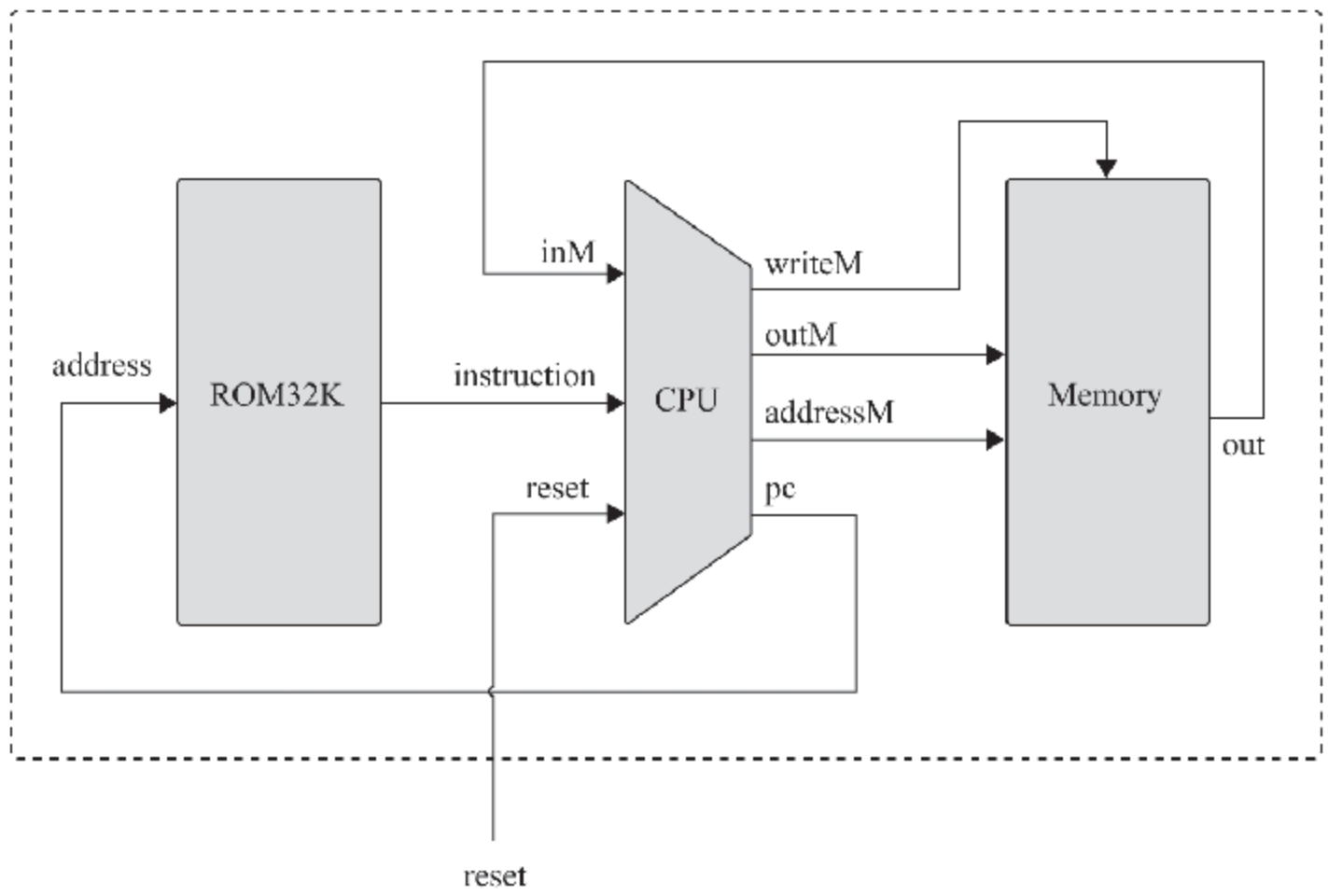


Figure 5.9 Proposed implementation of Computer, the topmost chip in the Hack platform.

The Computer implementation is designed to realize the following fetch-execute cycle: When the user asserts the reset input, the CPU's pc output emits 0, causing the instruction memory (ROM32K) to emit the first instruction in the program. The instruction will be executed by the CPU, and this execution may involve reading or writing a data memory register.

In the process of executing the instruction, the CPU figures out which instruction to fetch next, and emits this address through its pc output. The CPU's pc output feeds the address input of the instruction memory, causing the latter to output the instruction that ought to be executed next. This output, in turn, feeds the instruction input of the CPU, closing the fetch-execute cycle.

5.4 Project

Objective: Build the Hack computer, culminating in the topmost Computer chip.

Resources: All the chips described in this chapter should be written in HDL and tested on the supplied hardware simulator, using the test programs described below.

Contract: Build a hardware platform capable of executing programs written in the Hack machine language. Demonstrate the platform's operations by having your Computer chip run the three supplied test programs.

Test programs: A natural way to test the overall Computer chip implementation is to have it execute sample programs written in the Hack machine language. In order to run such a test, one can write a test script that loads the Computer chip into the hardware simulator, loads a program from an external text file into the ROM32K chip-part (the instruction memory), and then runs the clock enough cycles to execute the program. We provide three such test programs, along with corresponding test scripts and compare files:

- Add.hack: Adds the two constants 2 and 3, and writes the result in RAM[0].
- Max.hack: Computes the maximum of RAM[0] and RAM[1] and writes the result in RAM[2].
- Rect.hack: Draws on the screen a rectangle of RAM[0] rows of 16 pixels each. The rectangle's top-left corner is located at the top-left corner of the screen.

Before testing your Computer chip on any one of these programs, review the test script associated with the program, and be sure to understand the instructions given to the simulator. If needed, consult appendix 3 ("Test Description Language").

Steps: Implement the computer in the following order:

Memory: This chip can be built along the general outline given in figure 5.6, using three chip-parts: RAM16K, Screen, and Keyboard. Screen and Keyboard are available as built-in chips; there is no need to build them. Although the RAM16K chip was built in project 3, we recommend using its built-in version instead.

CPU: The central processing unit can be built according to the proposed implementation given in figure 5.8. In principle, the CPU can use the ALU built in project 2, the Register and PC chips built in project 3, and logic gates from project 1, as needed. However, we recommend using the built-in versions of all these chips (in particular, use the built-in registers ARegister, DRegister, and PC). The built-in chips have exactly the same functionality as the memory chips built in previous projects, but they feature GUI side effects that make the testing and simulation of your work easier.

In the course of implementing the CPU, you may be tempted to specify and build internal ("helper") chips of your own. Be advised that there is no need to do so; the Hack CPU can be implemented elegantly and efficiently using only the chip-parts that appear in figure 5.8, plus some elementary logic gates built in project 1 (of which it is best to use their built-in versions).

Instruction memory: Use the built-in ROM32K chip.

Computer: The computer can be built according to the proposed implementation given in figure 5.9.

Hardware simulator: All the chips in this project, including the topmost Computer chip, can be implemented and tested using the supplied hardware simulator. Figure 5.10 is a screenshot of testing the Rect.hack program on a Computer chip implementation.

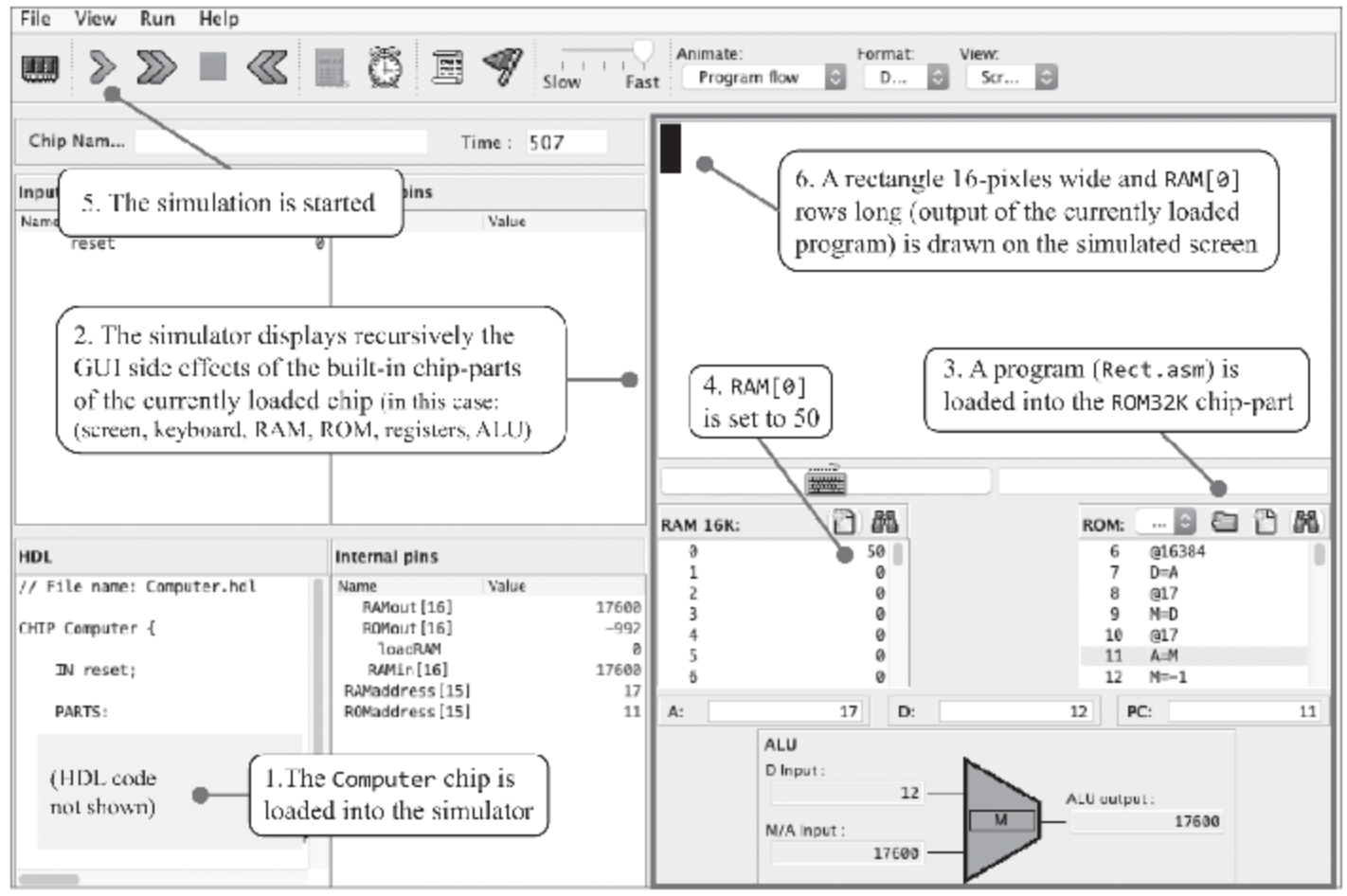


Figure 5.10 Testing the Computer chip on the supplied hardware simulator. The stored program is Rect, which draws a rectangle of RAM[0] rows of 16 pixels each, all black, at the top-left of the screen.

A web-based version of project 5 is available at www.nand2tetris.org.

5.5 Perspective

Following the Nand to Tetris spirit, the architecture of the Hack computer is minimal. Typical computer platforms feature more registers, more data types, more powerful ALUs, and richer instruction sets. Most of these differences, though, are quantitative. From a qualitative standpoint, almost all digital computers, including Hack, are based on the same conceptual architecture: the von Neumann machine.

In terms of *purpose*, computer systems can be classified into two categories: *general-purpose computers* and *single-purpose computers*. General-purpose computers, like PCs and cell phones, typically interact with a user. They are designed to execute many programs and easily switch from one program to another. *Single-purpose computers* are usually embedded in other systems like automobiles, cameras, media streamers, medical devices, industrial controllers, and so on. For any particular application, a single program is burned into the dedicated computer's ROM

(Read-Only Memory). For example, in some game consoles, the game software resides in an external cartridge, which is a replaceable ROM module encased in a fancy package. Although general-purpose computers are typically more complex and versatile than dedicated computers, they all share the same basic architectural ideas: stored programs, fetch-decode-execute logic, CPU, registers, and counters.

Most general-purpose computers use a single address space for storing both programs and data. Other computers, like Hack, use two separate address spaces. The latter configuration, which for historical reasons is called *Harvard architecture*, is less flexible in terms of ad hoc memory utilization but has distinct advantages. First, it is easier and cheaper to build. Second, it is often faster than the single address space configuration. Finally, if the size of the program that the computer has to run is known in advance, the size of the instruction memory can be optimized and fixed accordingly. For these reasons, the Harvard architecture is the architecture of choice in many dedicated, single-purpose, embedded computers.

Computers that use the same address space for storing instructions and data face the following challenge: How can we feed the address of the instruction, and the address of the data register on which the instruction has to operate, into the same address input of the shared memory device? Clearly, we cannot do it at the same time. The standard solution is to base the computer operation on a two-cycle logic. During the *fetch cycle*, the instruction address is fed to the address input of the memory, causing it to immediately emit the current instruction, which is then stored in an *instruction register*. In the subsequent *execute cycle*, the instruction is decoded, and the data address on which it has to operate is fed to the same address input. In contrast, computers that use separate instruction and data memories, like Hack, benefit from a single-cycle fetch-execute logic, which is faster and easier to handle. The price is having to use separate data and instruction memory units, although there is no need to use an instruction register.

The Hack computer interacts with a screen and a keyboard. General-purpose computers are typically connected to multiple I/O devices like printers, storage devices, network connections, and so on. Also, typical display devices are much fancier than the Hack screen, featuring more pixels, more colors, and faster rendering performance. Still, the basic

principle that each pixel is driven by a memory-resident binary value is maintained: instead of a single bit controlling the pixel's black or white color, typically 8 bits are devoted to controlling the brightness level of each of several primary colors that, taken together, produce the pixel's ultimate color. The result is millions of possible colors, more than the human eye can discern.

The mapping of the Hack screen on the computer's main memory is simplistic. Instead of having memory bits drive pixels directly, many computers allow the CPU to send high-level graphic instructions like "draw a line" or "draw a circle" to a dedicated graphics chip or a standalone graphics processing unit, also known as GPU. The hardware and low-level software of these dedicated graphical processors are especially optimized for rendering graphics, animation, and video, offloading from the CPU and the main computer the burden of handling these voracious tasks directly.

Finally, it should be stressed that much of the effort and creativity in designing computer hardware is invested in achieving better performance. Many hardware architects devote their work to speeding up memory access, using clever caching algorithms and data structures, optimizing access to I/O devices, and applying pipelining, parallelism, instruction prefetching, and other optimization techniques that were completely sidestepped in this chapter.

Historically, attempts to accelerate processing performance have led to two main camps of CPU design. Advocates of the *Complex Instruction Set Computing* (CISC) approach argued for achieving better performance by building more powerful processors featuring more elaborate instruction sets. Conversely, the *Reduced Instruction Set Computing* (RISC) camp built simpler processors and tighter instruction sets, arguing that these actually deliver faster performance in benchmark tests. The Hack computer does not enter this debate, featuring neither a strong instruction set nor special hardware acceleration techniques.

6 Assembler

What's in a name? That which we call a rose by any other name would smell as sweet.

—Shakespeare, *Romeo and Juliet*

In the previous chapters, we completed the development of a hardware platform designed to run programs in the Hack machine language. We presented two versions of this language—symbolic and binary—and explained that symbolic programs can be translated into binary code using a program called an *assembler*. In this chapter we describe how assemblers work, and how they are built. This will lead to the construction of a *Hack assembler*—a program that translates programs written in the Hack symbolic language into binary code that can execute on the barebone Hack hardware.

Since the relationship between symbolic instructions and their corresponding binary codes is straightforward, implementing an assembler using a high-level programming language is not a difficult task. One complication arises from allowing assembly programs to use symbolic references to memory addresses. The assembler is expected to manage these symbols and resolve them to physical memory addresses. This task is normally done using a *symbol table*—a commonly used data structure.

Implementing the assembler is the first in a series of seven software development projects that accompany part II of the book. Developing the assembler will equip you with a basic set of general skills that will serve you well throughout all these projects and beyond: handling command-line arguments, handling input and output text files, parsing instructions, handling white space, handling symbols, generating code, and many other techniques that come into play in many software development projects.

If you have no programming experience, you can develop a paper-based assembler. This option is described in the web-based version of project 6, available at www.nand2tetris.org.

6.1 Background

Machine languages are typically specified in two flavors: *binary* and *symbolic*. A binary instruction, for example, 1100001000000011000000000000111, is an agreed-upon package of micro-codes designed to be decoded and executed by some target hardware platform. For example, the instruction's leftmost 8 bits, 11000010, can represent an operation like "load." The next 8 bits, 00000011, can represent a register, say R3. The remaining 16 bits, 0000000000000111, can represent a value, say 7. When we set out to build a hardware architecture and a machine language, we can decide that this particular 32-bit instruction will cause the hardware to effect the operation "load the constant 7 into register R3." Modern computer platforms support hundreds of such possible operations. Thus, machine languages can be complex, involving many operation codes, memory addressing modes, and instruction formats.

Clearly, specifying these operations in binary code is a pain. A natural solution is using an agreed-upon equivalent symbolic syntax, say, "load R3,7". The load operation code is sometimes called a *mnemonic*, which in Latin means a pattern of letters designed to help with remembering something. Since the translation from mnemonics and symbols to binary code is straightforward, it makes sense to write low-level programs directly in symbolic notation and have a computer program translate them into binary code. The symbolic language is called *assembly*, and the translator program *assembler*. The assembler parses each assembly instruction into its underlying fields, for example, load, R3, and 7, translates each field into its equivalent binary code, and finally assembles the generated bits into a binary instruction that can be executed by the hardware. Hence the name *assembler*.

Symbols: Consider the symbolic instruction goto 312. Following translation, this instruction causes the computer to fetch and execute the instruction

stored in address 312, which may be the beginning of some loop. Well, if it's the beginning of a loop, why not mark this point in the assembly program with a descriptive label, say LOOP, and use the command goto LOOP instead of goto 312? All we have to do is record somewhere that LOOP stands for 312. When we translate the program into binary code, we replace each occurrence of LOOP with 312. That's a small price to pay for the gain in program readability and portability.

In general, assembly languages use symbols for three purposes:

- *Labels*: Assembly programs can declare and use symbols that mark various locations in the code, for example, LOOP and END.
- *Variables*: Assembly programs can declare and use symbolic variables, for example, i and sum.
- *Predefined symbols*: Assembly programs can refer to special addresses in the computer's memory using agreed-upon symbols, for example, SCREEN and KBD.

Of course, there is no free lunch. Someone must be responsible for managing all these symbols. In particular, someone must remember that SCREEN stands for 16384, that LOOP stands for 312, that sum stands for some other address, and so on. This symbol-handling task is one of the most important functions of the assembler.

Example: Figure 6.1 lists two versions of the same program written in the Hack machine language. The symbolic version includes all sorts of things that humans are fond of seeing in computer programs: comments, white space, indentation, symbolic instructions, and symbolic references. None of these embellishments concern computers, which understand one thing only: bits. The agent that bridges the gap between the symbolic code convenient for humans and the binary code understood by the computer is the assembler.