# 1 Boolean Logic

*Such simple things, And we make of them something so complex it defeats us, Almost.*
—John Ashbery (b. 1927), American poet

Every digital device—be it a personal computer, a cellular telephone, or a network router—is based on a set of chips designed to store and process information. Although these chips come in different shapes and forms, they are all made from the same building blocks: Elementary *logic gates*. The gates can be physically implemented in many different materials and fabrication technologies, but their logical behavior is consistent across all computers. In this chapter we start out with one primitive logic gate—Nand—and build all the other logic gates from it. The result is a rather standard set of gates, which will be later used to construct our computer's processing and storage chips. This will be done in chapters 2 and 3, respectively.

All the hardware chapters in the book, beginning with this one, have the same structure. Each chapter focuses on a well-defined task, designed to construct or integrate a certain family of chips. The prerequisite knowledge needed to approach this task is provided in a brief Background section. The next section provides a complete Specification of the chips' abstractions, namely, the various services that they should deliver, one way or another. Having presented the *what*, a subsequent Implementation section proposes guidelines and hints about *how* the chips can be actually implemented. A Perspective section rounds up the chapter with concluding comments about important topics that were left out from the discussion. Each chapter ends with a technical Project section. This section gives step-by-step instructions for actually building the chips on a personal computer, using the hardware simulator supplied with the book.

This being the first hardware chapter in the book, the Background section is somewhat lengthy, featuring a special section on *hardware description and simulation tools*.

## 1.1 Background

This chapter focuses on the construction of a family of simple chips called *Boolean gates*. Since Boolean gates are physical implementations of *Boolean functions*, we start with a brief treatment of Boolean algebra. We then show how Boolean gates implementing simple Boolean functions can be interconnected to deliver the functionality of more complex chips. We conclude the background section with a description of how hardware design is actually done in practice, using software simulation tools.

### 1.1.1 Boolean Algebra

Boolean algebra deals with Boolean (also called binary) values that are typically labeled true/false, 1/0, yes/no, on/off, and so forth. We will use 1 and 0. A Boolean function is a function that operates on binary inputs and returns binary outputs. Since computer hardware is based on the representation and manipulation of binary values, Boolean functions play a central role in the specification, construction, and optimization of hardware architectures. Hence, the ability to formulate and analyze Boolean functions is the first step toward constructing computer architectures.

**Truth Table Representation** The simplest way to specify a Boolean function is to enumerate all the possible values of the function's input variables, along with the function's output for each set of inputs. This is called the *truth table* representation of the function, illustrated in figure 1.1.

The first three columns of figure 1.1 enumerate all the possible binary values of the function's variables. For each one of the $2^n$ possible tuples $v_1 \ldots v_n$ (here $n = 3$), the last column gives the value of $f(v_1 \ldots v_n)$.

**Boolean Expressions** In addition to the truth table specification, a Boolean function can also be specified using Boolean operations over its input variables. The basic Boolean operators that are typically used are "And" ($x$ And $y$ is 1 exactly when both $x$ and $y$ are 1) "Or" ($x$ Or $y$ is 1 exactly when either $x$ or $y$ or both are 1), and "Not" (Not $x$ is 1 exactly when $x$ is 0). We will use a common arithmetic-like notation for these operations: $x \cdot y$ (or $xy$) means $x$ And $y$, $x + y$ means $x$ Or $y$, and $\bar{x}$ means Not $x$.

To illustrate, the function defined in figure 1.1 is equivalently given by the Boolean expression $f(x, y, z) = (x + y) \cdot \bar{z}$. For example, let us evaluate this expression on the

| $x$ | $y$ | $z$ | $f(x, y, z)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Figure 1.1** Truth table representation of a Boolean function (example).

inputs $x = 0,\ y = 1,\ z = 0$ (third row in the table). Since $y$ is 1, it follows that $x + y = 1$ and thus $1 \cdot \bar{0} = 1 \cdot 1 = 1$. The complete verification of the equivalence between the expression and the truth table is achieved by evaluating the expression on each of the eight possible input combinations, verifying that it yields the same value listed in the table's right column.

**Canonical Representation** As it turns out, every Boolean function can be expressed using at least one Boolean expression called the *canonical representation*. Starting with the function's truth table, we focus on all the rows in which the function has value 1. For each such row, we construct a term created by And-ing together *literals* (variables or their negations) that fix the values of all the row's inputs. For example, let us focus on the third row in figure 1.1, where the function's value is 1. Since the variable values in this row are $x = 0,\ y = 1,\ z = 0$, we construct the term $\bar{x}y\bar{z}$. Following the same procedure, we construct the terms $x\bar{y}\bar{z}$ and $xy\bar{z}$ for rows 5 and 7. Now, if we Or-together all these terms (for all the rows where the function has value 1), we get a Boolean expression that is equivalent to the given truth table. Thus the canonical representation of the Boolean function shown in figure 1.1 is $f(x, y, z) = \bar{x}y\bar{z} + x\bar{y}\bar{z} + xy\bar{z}$. This construction leads to an important conclusion: Every Boolean function, no matter how complex, can be expressed using three Boolean operators only: And, Or, and Not.

**Two-Input Boolean Functions** An inspection of figure 1.1 reveals that the number of Boolean functions that can be defined over $n$ binary variables is $2^{2^n}$. For example, the sixteen Boolean functions spanned by two variables are listed in figure 1.2. These functions were constructed systematically, by enumerating all the possible 4-wise combinations of binary values in the four right columns. Each function has a conventional

| Function | $x$ | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|
| | $y$ | 0 | 1 | 0 | 1 |
| Constant 0 | 0 | 0 | 0 | 0 | 0 |
| And | $x \cdot y$ | 0 | 0 | 0 | 1 |
| $x$ And Not $y$ | $x \cdot \bar{y}$ | 0 | 0 | 1 | 0 |
| $x$ | $x$ | 0 | 0 | 1 | 1 |
| Not $x$ And $y$ | $\bar{x} \cdot y$ | 0 | 1 | 0 | 0 |
| $y$ | $y$ | 0 | 1 | 0 | 1 |
| Xor | $x \cdot \bar{y} + \bar{x} \cdot y$ | 0 | 1 | 1 | 0 |
| Or | $x + y$ | 0 | 1 | 1 | 1 |
| Nor | $\overline{x + y}$ | 1 | 0 | 0 | 0 |
| Equivalence | $x \cdot y + \bar{x} \cdot \bar{y}$ | 1 | 0 | 0 | 1 |
| Not $y$ | $\bar{y}$ | 1 | 0 | 1 | 0 |
| If $y$ then $x$ | $x + \bar{y}$ | 1 | 0 | 1 | 1 |
| Not $x$ | $\bar{x}$ | 1 | 1 | 0 | 0 |
| If $x$ then $y$ | $\bar{x} + y$ | 1 | 1 | 0 | 1 |
| Nand | $\overline{x \cdot y}$ | 1 | 1 | 1 | 0 |
| Constant 1 | 1 | 1 | 1 | 1 | 1 |

**Figure 1.2** All the Boolean functions of two variables.

name that seeks to describe its underlying operation. Here are some examples: The name of the Nor function is shorthand for Not-Or: Take the Or of $x$ and $y$, then negate the result. The Xor function—shorthand for "exclusive or"—returns 1 when its two variables have opposing truth-values and 0 otherwise. Conversely, the Equivalence function returns 1 when the two variables have identical truth-values. The If-$x$-then-$y$ function (also known as $x \rightarrow y$, or "$x$ Implies $y$") returns 1 when $x$ is 0 or when both $x$ and $y$ are 1. The other functions are self-explanatory.

The Nand function (as well as the Nor function) has an interesting theoretical property: Each one of the operations And, Or, and Not can be constructed from it, and it alone (e.g., $x$ Or $y = (x$ Nand $x)$ Nand $(y$ Nand $y)$. And since every Boolean function can be constructed from And, Or, and Not operations using the canonical representation method, it follows that every Boolean function can be constructed from Nand operations alone. This result has far-reaching practical implications: Once we have in our disposal a physical device that implements Nand, we can use many copies of this device (wired in a certain way) to implement in hardware any Boolean function.

### 1.1.2 Gate Logic

A *gate* is a physical device that implements a Boolean function. If a Boolean function $f$ operates on $n$ variables and returns $m$ binary results (in all our examples so far, $m$ was 1), the gate that implements $f$ will have *n input pins* and *m output pins*. When we put some values $v_1 \ldots v_n$ in the gate's input pins, the gate's "logic"—its internal structure—should compute and output $f(v_1 \ldots v_n)$. And just like complex Boolean functions can be expressed in terms of simpler functions, complex gates are composed from more elementary gates. The simplest gates of all are made from tiny switching devices, called *transistors*, wired in a certain topology designed to effect the overall gate functionality.

Although most digital computers today use electricity to represent and transmit binary data from one gate to another, any alternative technology permitting switching and conducting capabilities can be employed. Indeed, during the last fifty years, researchers have built many hardware implementations of Boolean functions, including magnetic, optical, biological, hydraulic, and pneumatic mechanisms. Today, most gates are implemented as transistors etched in silicon, packaged as *chips*. In this book we use the words *chip* and *gate* interchangeably, tending to use the term *gates* for simple chips.

The availability of alternative switching technology options, on the one hand, and the observation that Boolean algebra can be used to abstract the behavior of *any* such technology, on the other, is extremely important. Basically, it implies that computer scientists don't have to worry about physical things like electricity, circuits, switches, relays, and power supply. Instead, computer scientists can be content with the abstract notions of Boolean algebra and gate logic, trusting that someone else (the physicists and electrical engineers—bless their souls) will figure out how to actually realize them in hardware. Hence, a *primitive gate* (see figure 1.3) can be viewed as a black box device that implements an elementary logical operation in one way or another—we don't care how. A hardware designer starts from such primitive gates and designs more complicated functionality by interconnecting them, leading to the construction of *composite* gates.

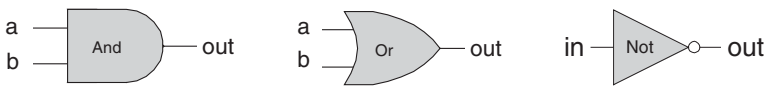



**Figure 1.3** Standard symbolic notation of some elementary logic gates.

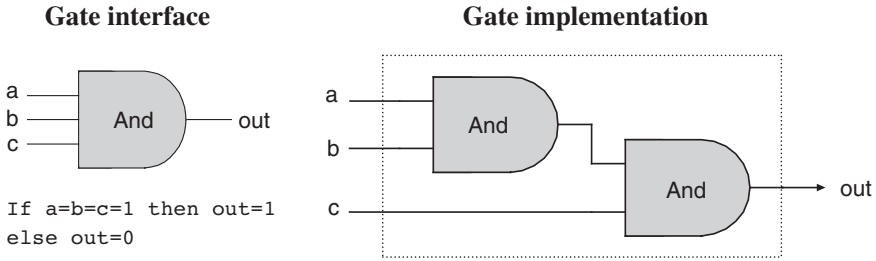



**Figure 1.4** Composite implementation of a three-way And gate. The rectangle on the right defines the conceptual boundaries of the gate interface.

**Primitive and Composite Gates** Since all logic gates have the same input and output semantics (0's and 1's), they can be chained together, creating *composite gates* of arbitrary complexity. For example, suppose we are asked to implement the 3-way Boolean function $\text{And}(a, b, c)$. Using Boolean algebra, we can begin by observing that $a \cdot b \cdot c = (a \cdot b) \cdot c$, or, using prefix notation, $\text{And}(a, b, c) = \text{And}(\text{And}(a, b), c)$. Next, we can use this result to construct the composite gate depicted in figure 1.4.

The construction described in figure 1.4 is a simple example of *gate logic*, also called *logic design*. Simply put, logic design is the art of interconnecting gates in order to implement more complex functionality, leading to the notion of *composite gates*. Since composite gates are themselves realizations of (possibly complex) Boolean functions, their "outside appearance" (e.g., left side of figure 1.4) looks just like that of primitive gates. At the same time, their internal structure can be rather complex.

We see that any given logic gate can be viewed from two different perspectives: external and internal. The right-hand side of figure 1.4 gives the gate's internal architecture, or *implementation*, whereas the left side shows only the gate *interface*, namely, the input and output pins that it exposes to the outside world. The former is relevant only to the gate designer, whereas the latter is the right level of detail for other designers who wish to use the gate as an abstract off-the-shelf component, without paying attention to its internal structure.

Let us consider another logic design example—that of a Xor gate. As discussed before, $\text{Xor}(a, b)$ is 1 exactly when either $a$ is 1 and $b$ is 0, or when $a$ is 0 and $b$ is 1. Said otherwise, $\text{Xor}(a, b) = \text{Or}(\text{And}(a, \text{Not}(b)), \text{And}(\text{Not}(a), b))$. This definition leads to the logic design shown in figure 1.5.

Note that the gate *interface* is unique: There is only one way to describe it, and this is normally done using a truth table, a Boolean expression, or some verbal specifica-

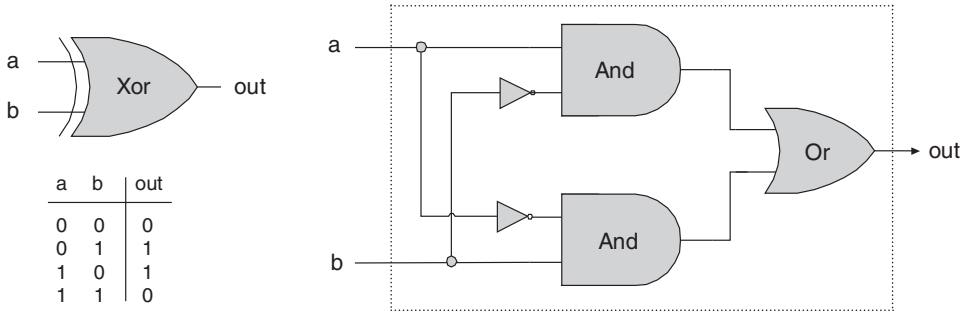



**Figure 1.5** Xor gate, along with a possible implementation.

tion. This interface, however, can be realized using many different *implementations*, some of which will be better than others in terms of cost, speed, and simplicity. For example, the Xor function can be implemented using four, rather than five, And, Or, and Not gates. Thus, from a functional standpoint, the fundamental requirement of logic design is that *the gate implementation will realize its stated interface, in one way or another*. From an efficiency standpoint, the general rule is to try to *do more with less*, that is, use as few gates as possible.

To sum up, the art of logic design can be described as follows: Given a gate specification (interface), find an efficient way to implement it using other gates that were already implemented. This, in a nutshell, is what we will do in the rest of this chapter.

### 1.1.3 Actual Hardware Construction

Having described the logic of composing complex gates from simpler ones, we are now in a position to discuss how gates are actually built. Let us start with an intentionally naïve example.

Suppose we open a chip fabrication shop in our home garage. Our first contract is to build a hundred Xor gates. Using the order's downpayment, we purchase a soldering gun, a roll of copper wire, and three bins labeled "And gates," "Or gates," and "Not gates," each containing many identical copies of these elementary logic gates. Each of these gates is sealed in a plastic casing that exposes some input and output pins, as well as a power supply plug. To get started, we pin figure 1.5 to our garage wall and proceed to realize it using our hardware. First, we take two And gates, two Not gates, and one Or gate, and mount them on a board according to the

figure's layout. Next, we connect the chips to one another by running copper wires among them and by soldering the wire ends to the respective input/output pins. Now, if we follow the gate diagram carefully, we will end up having three exposed wire ends. We then solder a pin to each one of these wire ends, seal the entire device (except for the three pins) in a plastic casing, and label it "Xor." We can repeat this assembly process many times over. At the end of the day, we can store all the chips that we've built in a new bin and label it "Xor gates." If we (or other people) are asked to construct some other chips in the future, we'll be able to use these Xor gates as elementary building blocks, just as we used the And, Or, and Not gates before.

As the reader has probably sensed, the garage approach to chip production leaves much to be desired. For starters, there is no guarantee that the given chip diagram is correct. Although we can prove correctness in simple cases like Xor, we cannot do so in many realistically complex chips. Thus, we must settle for empirical testing: Build the chip, connect it to a power supply, activate and deactivate the input pins in various configurations, and hope that the chip outputs will agree with its specifications. If the chip fails to deliver the desired outputs, we will have to tinker with its physical structure—a rather messy affair. Further, even if we will come up with the right design, replicating the chip assembly process many times over will be a time-consuming and error-prone affair. There must be a better way!

### 1.1.4 Hardware Description Language (HDL)

Today, hardware designers no longer build anything with their bare hands. Instead, they plan and optimize the chip architecture on a computer workstation, using structured modeling formalisms like *Hardware Description Language*, or HDL (also known as VHDL, where V stands for *Virtual*). The designer specifies the chip structure by writing an *HDL program*, which is then subjected to a rigorous battery of tests. These tests are carried out virtually, using computer simulation: A special software tool, called a *hardware simulator*, takes the HDL program as input and builds an image of the modeled chip in memory. Next, the designer can instruct the simulator to test the virtual chip on various sets of inputs, generating simulated chip outputs. The outputs can then be compared to the desired results, as mandated by the client who ordered the chip built.

In addition to testing the chip's correctness, the hardware designer will typically be interested in a variety of parameters such as speed of computation, energy consumption, and the overall cost implied by the chip design. All these param-

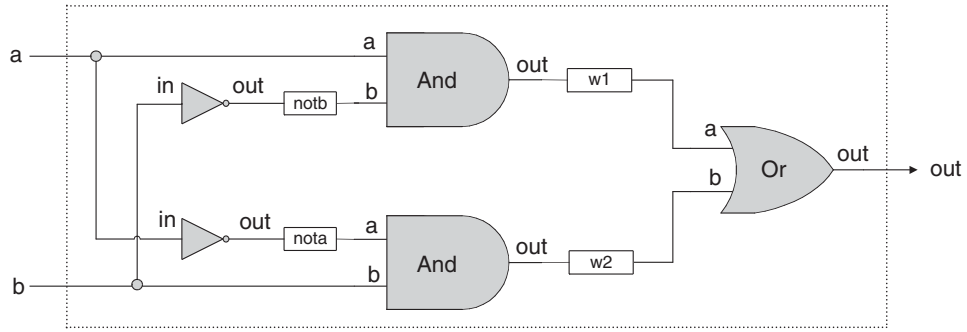
eters can be simulated and quantified by the hardware simulator, helping the designer optimize the design until the simulated chip delivers desired cost/performance levels.

Thus, using HDL, one can completely plan, debug, and optimize the entire chip before a single penny is spent on actual production. When the HDL program is deemed complete, that is, when the performance of the simulated chip satisfies the client who ordered it, the HDL program can become the blueprint from which many copies of the physical chip can be stamped in silicon. This final step in the chip life cycle—from an optimized HDL program to mass production—is typically outsourced to companies that specialize in chip fabrication, using one switching technology or another.

**Example: Building a Xor Gate** As we have seen in figures 1.2 and 1.5, one way to define *exclusive or* is $\mathrm{Xor}(a,b) = \mathrm{Or}(\mathrm{And}(a,\mathrm{Not}(b)),\mathrm{And}(\mathrm{Not}(a),b))$. This logic can be expressed either graphically, as a gate diagram, or textually, as an HDL program. The latter program is written in the HDL variant used throughout this book, defined in appendix A. See figure 1.6 for the details.

**Explanation** An HDL definition of a chip consists of a *header* section and a *parts* section. The header section specifies the chip *interface*, namely the chip name and the names of its input and output pins. The parts section describes the names and topology of all the lower-level parts (other chips) from which this chip is constructed. Each part is represented by a *statement* that specifies the part name and the way it is connected to other parts in the design. Note that in order to write such statements legibly, the HDL programmer must have a complete documentation of the underlying parts' *interfaces*. For example, figure 1.6 assumes that the input and output pins of the Not gate are labeled `in` and `out`, and those of And and Or are labeled `a`, `b` and `out`. This API-type information is not obvious, and one must have access to it before one can plug the chip parts into the present code.

Inter-part connections are described by creating and connecting *internal pins*, as needed. For example, consider the bottom of the gate diagram, where the output of a Not gate is piped into the input of a subsequent And gate. The HDL code describes this connection by the pair of statements `Not(...,out=nota)` and `And(a=nota,...)`. The first statement creates an internal pin (outbound wire) named `nota`, feeding `out` into it. The second statement feeds the value of `nota` into the `a` input of an And gate. Note that pins may have an unlimited fan out. For example, in figure 1.6, each input is simultaneously fed into two gates. In gate




***HDL program*** (`Xor.hdl`)

```
/* Xor (exclusive or) gate:
   If a<>b out=1 else out=0. */
CHIP Xor {
   IN a, b;
   OUT out;
   PARTS:
   Not(in=a, out=nota);
   Not(in=b, out=notb);
   And(a=a, b=notb, out=w1);
   And(a=nota, b=b, out=w2);
   Or(a=w1, b=w2, out=out);
}
```

***Test script*** (`Xor.tst`)

```
load Xor.hdl,
output-list a, b, out;
set a 0, set b 0,
eval, output;
set a 0, set b 1,
eval, output;
set a 1, set b 0,
eval, output;
set a 1, set b 1,
eval, output;
```

***Output file*** (`Xor.out`)

```
a | b | out
0 | 0 | 0
0 | 1 | 1
1 | 0 | 1
1 | 1 | 0
```

**Figure 1.6** HDL implementation of a Xor gate.

diagrams, multiple connections are described using forks. In HDL, the existence of forks is implied by the code.

**Testing** Rigorous quality assurance mandates that chips be tested in a specific, replicable, and well-documented fashion. With that in mind, hardware simulators are usually designed to run *test scripts*, written in some scripting language. For example, the test script in figure 1.6 is written in the scripting language understood by the hardware simulator supplied with the book. This scripting language is described fully in appendix B.

Let us give a brief description of the test script from figure 1.6. The first two lines of the test script instruct the simulator to load the `Xor.hdl` program and get ready to

print the values of selected variables. Next, the script lists a series of testing scenarios, designed to simulate the various contingencies under which the Xor chip will have to operate in "real-life" situations. In each scenario, the script instructs the simulator to bind the chip inputs to certain data values, compute the resulting output, and record the test results in a designated output file. In the case of simple gates like Xor, one can write an exhaustive test script that enumerates all the possible input values of the gate. The resulting output file (right side of figure 1.6) can then be viewed as a complete empirical proof that the chip is well designed. The luxury of such certitude is not feasible in more complex chips, as we will see later.

### 1.1.5 Hardware Simulation

Since HDL is a hardware construction *language*, the process of writing and debugging HDL programs is quite similar to software development. The main difference is that instead of writing code in a language like Java, we write it in HDL, and instead of using a compiler to translate and test the code, we use a *hardware simulator*. The hardware simulator is a computer program that knows how to parse and interpret HDL code, turn it into an executable representation, and test it according to the specifications of a given test script. There exist many commercial hardware simulators on the market, and these vary greatly in terms of cost, complexity, and ease of use. Together with this book we provide a simple (and free!) hardware simulator that is sufficiently powerful to support sophisticated hardware design projects. In particular, the simulator provides all the necessary tools for building, testing, and integrating all the chips presented in the book, leading to the construction of a general-purpose computer. Figure 1.7 illustrates a typical chip simulation session.

## 1.2 Specification

This section specifies a typical set of gates, each designed to carry out a common Boolean operation. These gates will be used in the chapters that follow to construct the full architecture of a typical modern computer. Our starting point is a single primitive Nand gate, from which all other gates will be derived recursively. Note that we provide only the gates' specifications, or interfaces, delaying implementation details until a subsequent section. Readers who wish to construct the specified gates in HDL are encouraged to do so, referring to appendix A as needed. All the gates can be built and simulated on a personal computer, using the hardware simulator supplied with the book.

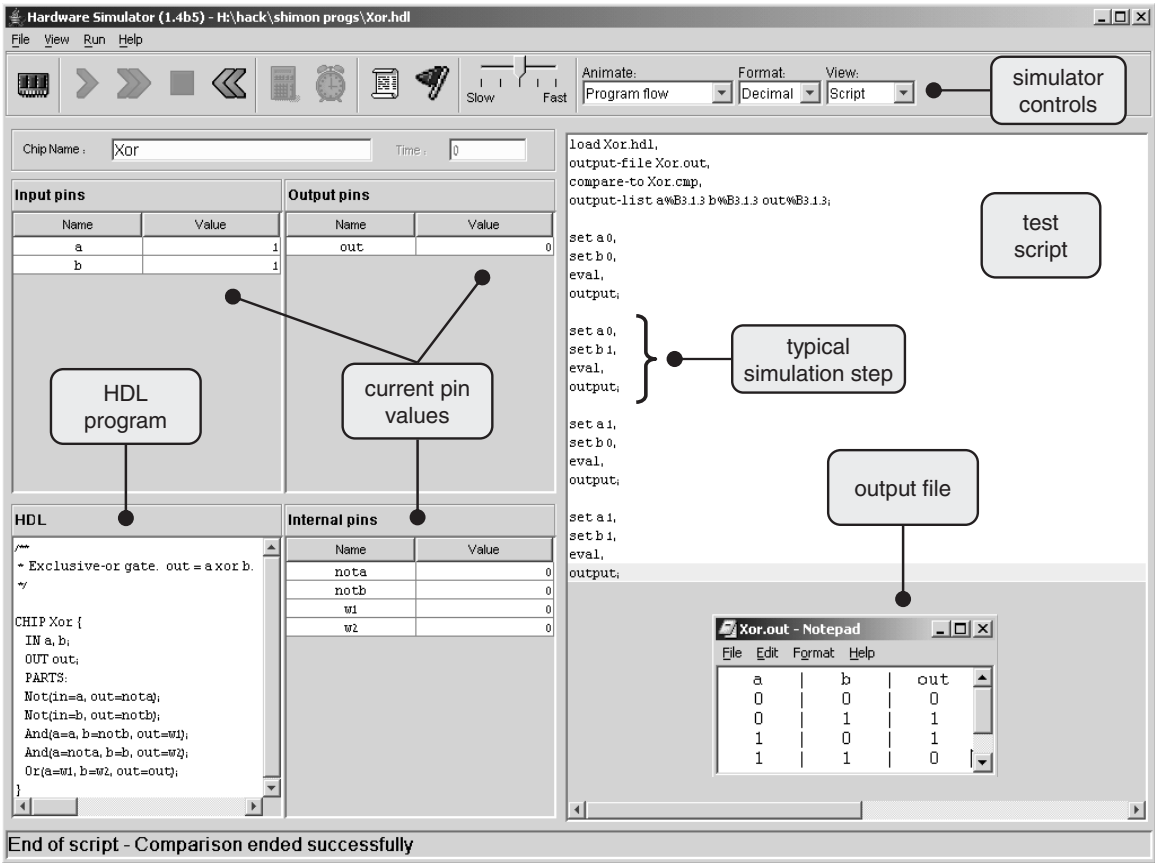



**Figure 1.7** A screen shot of simulating an Xor chip on the hardware simulator. The simulator state is shown just after the test script has completed running. The pin values correspond to the last simulation step ($a = b = 1$). Note that the *output file* generated by the simulation is consistent with the Xor truth table, indicating that the loaded HDL program delivers a correct Xor functionality. The *compare file*, not shown in the figure and typically specified by the chip's client, has exactly the same structure and contents as that of the output file. The fact that the two files agree with each other is evident from the status message displayed at the bottom of the screen.

### 1.2.1 The Nand Gate

The starting point of our computer architecture is the Nand gate, from which all other gates and chips are built. The Nand gate is designed to compute the following Boolean function:

| $a$ | $b$ | Nand$(a,b)$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Throughout the book, we use "chip API boxes" to specify chips. For each chip, the API specifies the chip name, the names of its input and output pins, the function or operation that the chip effects, and an optional comment.

```
Chip name: Nand
Inputs:    a, b
Outputs:   out
Function:  If a=b=1 then out=0 else out=1
Comment:   This gate is considered primitive and thus there is
           no need to implement it.
```

### 1.2.2 Basic Logic Gates

Some of the logic gates presented here are typically referred to as "elementary" or "basic." At the same time, every one of them can be composed from Nand gates alone. Therefore, they need not be viewed as primitive.

**Not** The single-input Not gate, also known as "converter," converts its input from 0 to 1 and vice versa. The gate API is as follows:

```
Chip name: Not
Inputs:    in
Outputs:   out
Function:  If in=0 then out=1 else out=0.
```

**And** The And function returns 1 when both its inputs are 1, and 0 otherwise.

```
Chip name: And
Inputs:    a, b
Outputs:   out
Function:  If a=b=1 then out=1 else out=0.
```

**Or** The Or function returns 1 when at least one of its inputs is 1, and 0 otherwise.

```
Chip name: Or
Inputs:    a, b
Outputs:   out
Function:  If a=b=0 then out=0 else out=1.
```

**Xor** The Xor function, also known as "exclusive or," returns 1 when its two inputs have opposing values, and 0 otherwise.

```
Chip name: Xor
Inputs:    a, b
Outputs:   out
Function:  If a≠b then out=1 else out=0.
```

**Multiplexor** A multiplexor (figure 1.8) is a three-input gate that uses one of the inputs, called "selection bit," to select and output one of the other two inputs, called "data bits." Thus, a better name for this device might have been *selector*. The name *multiplexor* was adopted from communications systems, where similar devices are used to serialize (multiplex) several input signals over a single output wire.

```
Chip name: Mux
Inputs:    a, b, sel
Outputs:   out
Function:  If sel=0 then out=a else out=b.
```

| a | b | sel | out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

| sel | out |
|---|---|
| 0 | a |
| 1 | b |

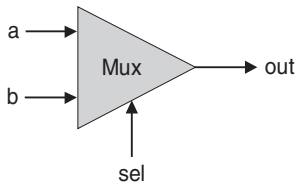



**Figure 1.8** Multiplexor. The table at the top right is an abbreviated version of the truth table on the left.

| sel | a | b |
|---|---|---|
| 0 | in | 0 |
| 1 | 0 | in |

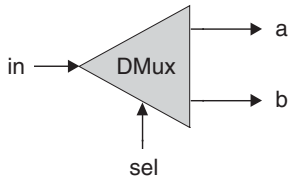



**Figure 1.9** Demultiplexor.

**Demultiplexor** A demultiplexor (figure 1.9) performs the opposite function of a multiplexor: It takes a single input and channels it to one of two possible outputs according to a selector bit that specifies which output to chose.

```
Chip name: DMux
Inputs:    in, sel
Outputs:   a, b
Function:  If sel=0 then {a=in, b=0} else {a=0, b=in}.
```

### 1.2.3 Multi-Bit Versions of Basic Gates

Computer hardware is typically designed to operate on multi-bit arrays called "buses." For example, a basic requirement of a 32-bit computer is to be able to compute (bit-wise) an And function on two given 32-bit buses. To implement this operation, we can build an array of 32 binary And gates, each operating separately

on a pair of bits. In order to enclose all this logic in one package, we can encapsulate the gates array in a single chip interface consisting of two 32-bit input buses and one 32-bit output bus.

This section describes a typical set of such multi-bit logic gates, as needed for the construction of a typical 16-bit computer. We note in passing that the architecture of *n*-bit logic gates is basically the same irrespective of *n*'s value.

When referring to individual bits in a bus, it is common to use an array syntax. For example, to refer to individual bits in a 16-bit bus named `data`, we use the notation `data[0], data[1],..., data[15]`.

**Multi-Bit Not** An *n*-bit Not gate applies the Boolean operation Not to every one of the bits in its *n*-bit input bus:

```
Chip name: Not16
Inputs:    in[16] // a 16-bit pin
Outputs:   out[16]
Function:  For i=0..15 out[i]=Not(in[i]).
```

**Multi-Bit And** An *n*-bit And gate applies the Boolean operation And to every one of the *n* bit-pairs arrayed in its two *n*-bit input buses:

```
Chip name: And16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  For i=0..15 out[i]=And(a[i],b[i]).
```

**Multi-Bit Or** An *n*-bit Or gate applies the Boolean operation Or to every one of the *n* bit-pairs arrayed in its two *n*-bit input buses:

```
Chip name: Or16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  For i=0..15 out[i]=Or(a[i],b[i]).
```

**Multi-Bit Multiplexor** An *n*-bit multiplexor is exactly the same as the binary multiplexor described in figure 1.8, except that the two inputs are each *n*-bit wide; the selector is a single bit.

```
Chip name: Mux16
Inputs:    a[16], b[16], sel
Outputs:   out[16]
Function:  If sel=0 then for i=0..15 out[i]=a[i]
           else for i=0..15 out[i]=b[i].
```

### 1.2.4 Multi-Way Versions of Basic Gates

Many 2-way logic gates that accept two inputs have natural generalization to multi-way variants that accept an arbitrary number of inputs. This section describes a set of multi-way gates that will be used subsequently in various chips in our computer architecture. Similar generalizations can be developed for other architectures, as needed.

**Multi-Way Or** An *n*-way Or gate outputs 1 when at least one of its *n* bit inputs is 1, and 0 otherwise. Here is the 8-way variant of this gate:

```
Chip name: Or8Way
Inputs:    in[8]
Outputs:   out
Function:  out=Or(in[0],in[1],...,in[7]).
```

**Multi-Way/Multi-Bit Multiplexor** An *m*-way *n*-bit multiplexor selects one of *m* *n*-bit input buses and outputs it to a single *n*-bit output bus. The selection is specified by a set of *k* control bits, where $k = \log_2 m$. Figure 1.10 depicts a typical example.

The computer platform that we develop in this book requires two variations of this chip: A 4-way 16-bit multiplexor and an 8-way 16-bit multiplexor:

| sel[1] | sel[0] | out |
|---|---|---|
| 0 | 0 | a |
| 0 | 1 | b |
| 1 | 0 | c |
| 1 | 1 | d |

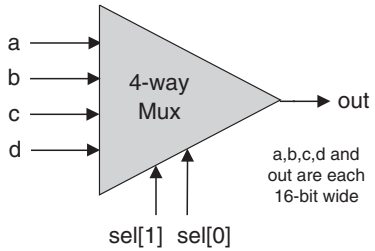



**Figure 1.10** 4-way multiplexor. The width of the input and output buses may vary.

```
Chip name: Mux4Way16
Inputs:    a[16], b[16], c[16], d[16], sel[2]
Outputs:   out[16]
Function:  If sel=00 then out=a else if sel=01 then out=b
           else if sel=10 then out=c else if sel=11 then out=d
Comment:   The assignment operations mentioned above are all
           16-bit. For example, "out=a" means "for i=0..15
           out[i]=a[i]".
```

```
Chip name: Mux8Way16
Inputs:    a[16],b[16],c[16],d[16],e[16],f[16],g[16],h[16],
           sel[3]
Outputs:   out[16]
Function:  If sel=000 then out=a else if sel=001 then out=b
           else if sel=010 out=c ... else if sel=111 then out=h
Comment:   The assignment operations mentioned above are all
           16-bit. For example, "out=a" means "for i=0..15
           out[i]=a[i]".
```

**Multi-Way/Multi-Bit Demultiplexor** An $m$-way $n$-bit demultiplexor (figure 1.11) channels a single $n$-bit input into one of $m$ possible $n$-bit outputs. The selection is specified by a set of $k$ control bits, where $k = \log_2 m$.

The specific computer platform that we will build requires two variations of this chip: A 4-way 1-bit demultiplexor and an 8-way 1-bit multiplexor, as follows.

| sel[1] | sel[0] | a | b | c | d |
|---|---|---|---|---|---|
| 0 | 0 | in | 0 | 0 | 0 |
| 0 | 1 | 0 | in | 0 | 0 |
| 1 | 0 | 0 | 0 | in | 0 |
| 1 | 1 | 0 | 0 | 0 | in |

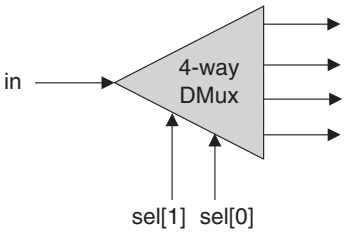



**Figure 1.11** 4-way demultiplexor.

```
Chip name: DMux4Way
Inputs:    in, sel[2]
Outputs:   a, b, c, d
Function:  If sel=00 then      {a=in, b=c=d=0}
           else if sel=01 then {b=in, a=c=d=0}
           else if sel=10 then {c=in, a=b=d=0}
           else if sel=11 then {d=in, a=b=c=0}.
```

```
Chip name: DMux8Way
Inputs:    in, sel[3]
Outputs:   a, b, c, d, e, f, g, h
Function:  If sel=000 then      {a=in, b=c=d=e=f=g=h=0}
           else if sel=001 then {b=in, a=c=d=e=f=g=h=0}
           else if sel=010 ...
           ...
           else if sel=111 then {h=in, a=b=c=d=e=f=g=0}.
```

## 1.3 Implementation

Similar to the role of *axioms* in mathematics, *primitive gates* provide a set of elementary building blocks from which everything else can be built. Operationally, primitive gates have an "off-the-shelf" implementation that is supplied externally. Thus, they can be used in the construction of other gates and chips without worrying about their internal design. In the computer architecture that we are now beginning

to build, we have chosen to base all the hardware on one primitive gate only: Nand. We now turn to outlining the first stage of this bottom-up hardware construction project, one gate at a time.

Our implementation guidelines are intentionally partial, since we want you to discover the actual gate architectures yourself. We reiterate that each gate can be implemented in more than one way; the simpler the implementation, the better.

**Not:** The implementation of a unary Not gate from a binary Nand gate is simple. Tip: Think positive.

**And:** Once again, the gate implementation is simple. Tip: Think negative.

**Or/Xor:** These functions can be defined in terms of some of the Boolean functions implemented previously, using some simple Boolean manipulations. Thus, the respective gates can be built using previously built gates.

**Multiplexor/Demultiplexor:** Likewise, these gates can be built using previously built gates.

**Multi-Bit Not/And/Or Gates:** Since we already know how to implement the elementary versions of these gates, the implementation of their *n*-ary versions is simply a matter of constructing arrays of *n* elementary gates, having each gate operate separately on its bit inputs. This implementation task is rather boring, but it will carry its weight when these multi-bit gates are used in more complex chips, as described in subsequent chapters.

**Multi-Bit Multiplexor:** The implementation of an *n*-ary multiplexor is simply a matter of feeding the same selection bit to every one of *n* binary multiplexors. Again, a boring task resulting in a very useful chip.

**Multi-Way Gates:** Implementation tip: Think forks.

## 1.4 Perspective

This chapter described the first steps taken in an applied digital design project. In the next chapter we will build more complicated functionality using the gates built here.

Although we have chosen to use Nand as our basic building block, other approaches are possible. For example, one can build a complete computer platform using Nor gates alone, or, alternatively, a combination of And, Or, and Not gates. These constructive approaches to logic design are theoretically equivalent, just as all theorems in geometry can be founded on different sets of axioms as alternative points of departure. The theory and practice of such constructions are covered in standard textbooks about *digital design* or *logic design*.

Throughout the chapter, we paid no attention to efficiency considerations such as the number of elementary gates used in constructing a composite gate or the number of wire crossovers implied by the design. Such considerations are critically important in practice, and a great deal of computer science and electrical engineering expertise focuses on optimizing them. Another issue we did not address at all is the physical implementation of gates and chips using the laws of physics, for example, the role of transistors embedded in silicon. There are of course several such implementation options, each having its own characteristics (speed, power requirements, production cost, etc.). Any nontrivial coverage of these issues requires some background in electronics and physics.

## 1.5 Project

**Objective** Implement all the logic gates presented in the chapter. The only building blocks that you can use are primitive Nand gates and the composite gates that you will gradually build on top of them.

**Resources** The only tool that you need for this project is the hardware simulator supplied with the book. All the chips should be implemented in the HDL language specified in appendix A. For each one of the chips mentioned in the chapter, we provide a skeletal `.hdl` program (text file) with a missing implementation part. In addition, for each chip we provide a `.tst` script file that tells the hardware simulator how to test it, along with the correct output file that this script should generate, called `.cmp` or "compare file." Your job is to complete the missing implementation parts of the supplied `.hdl` programs.

**Contract** When loaded into the hardware simulator, your chip design (modified `.hdl` program), tested on the supplied `.tst` file, should produce the outputs listed in the supplied `.cmp` file. If that is not the case, the simulator will let you know.

**Tips** The Nand gate is considered primitive and thus there is no need to build it: Whenever you use Nand in one of your HDL programs, the simulator will automatically invoke its built-in `tools/builtIn/Nand.hdl` implementation. We recommend implementing the other gates in this project in the order in which they appear in the chapter. However, since the `builtIn` directory features working versions of all the chips described in the book, you can always use these chips without defining them first: The simulator will automatically use their built-in versions.

For example, consider the skeletal `Mux.hdl` program supplied in this project. Suppose that for one reason or another you did not complete this program's implementation, but you still want to use Mux gates as internal parts in other chip designs. This is not a problem, thanks to the following convention. If our simulator fails to find a `Mux.hdl` file in the current directory, it automatically invokes a built-in Mux implementation, pre-supplied with the simulator's software. This built-in implementation—a Java class stored in the `builtIn` directory—has the same interface and functionality as those of the Mux gate described in the book. Thus, if you want the simulator to ignore one or more of your chip implementations, simply move the corresponding `.hdl` files out of the current directory.

**Steps** We recommend proceeding in the following order:

0. The *hardware simulator* needed for this project is available in the `tools` directory of the book's software suite.
1. Read appendix A, sections A1–A6 only.
2. Go through the *hardware simulator tutorial*, parts I, II, and III only.
3. Build and simulate all the chips specified in the `projects/01` directory.

# 2 Boolean Arithmetic

*Counting is the religion of this generation, its hope and salvation.*
—Gertrude Stein (1874–1946)

In this chapter we build gate logic designs that represent numbers and perform arithmetic operations on them. Our starting point is the set of logic gates built in chapter 1, and our ending point is a fully functional Arithmetic Logical Unit. The ALU is the centerpiece chip that executes all the arithmetic and logical operations performed by the computer. Hence, building the ALU functionality is an important step toward understanding how the Central Processing Unit (CPU) and the overall computer work.

As usual, we approach this task gradually. The first section gives a brief Background on how binary codes and Boolean arithmetic can be used, respectively, to represent and add signed numbers. The Specification section describes a succession of *adder chips*, designed to add two bits, three bits, and pairs of *n*-bit binary numbers. This sets the stage for the ALU specification, which is based on a sophisticated yet simple logic design. The Implementation and Project sections provide tips and guidelines on how to build the adder chips and the ALU on a personal computer, using the hardware simulator supplied with the book.

Binary addition is a simple operation that runs deep. Remarkably, most of the operations performed by digital computers can be reduced to elementary additions of binary numbers. Therefore, constructive understanding of binary addition holds the key to the implementation of numerous computer operations that depend on it, one way or another.

## 2.1 Background

**Binary Numbers** Unlike the decimal system, which is founded on base 10, the binary system is founded on base 2. When we are given a certain binary pattern, say "10011," and we are told that this pattern is supposed to represent an integer number, the decimal value of this number is computed by convention as follows:

$$(10011)_{two} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19 \tag{1}$$

In general, let $x = x_n x_{n-1} \ldots x_0$ be a string of digits. The *value* of $x$ in base $b$, denoted $(x)_b$, is defined as follows:

$$(x_n x_{n-1} \ldots x_0)_b = \sum_{i=0}^{n} x_i \cdot b^i \tag{2}$$

The reader can verify that in the case of $(10011)_{two}$, rule (2) reduces to calculation (1).

The result of calculation (1) happens to be 19. Thus, when we press the keyboard keys labeled '1', '9' and ENTER while running, say, a spreadsheet program, what ends up in some register in the computer's memory is the binary code 10011. More precisely, if the computer happens to be a 32-bit machine, what gets stored in the register is the bit pattern 00000000000000000000000000010011.

**Binary Addition** A pair of binary numbers can be added digit by digit from right to left, according to the same elementary school method used in decimal addition. First, we add the two right-most digits, also called the *least significant bits* (LSB) of the two binary numbers. Next, we add the resulting carry bit (which is either 0 or 1) to the sum of the next pair of bits up the significance ladder. We continue the process until the two *most significant bits* (MSB) are added. If the last bit-wise addition generates a carry of 1, we can report overflow; otherwise, the addition completes successfully:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | | (carry) | 1 | 1 | 1 | 1 | | |
| | **1** | **0** | **0** | **1** | $x$ | | **1** | **0** | **1** | **1** |
| + | **0** | **1** | **0** | **1** | $y$ | + | **0** | **1** | **1** | **1** |
| 0 | 1 | 1 | 1 | 0 | $x + y$ | 1 | 0 | 0 | 1 | 0 |
| no overflow | | | | | | overflow | | | | | |

We see that computer hardware for binary addition of two $n$-bit numbers can be built from logic gates designed to calculate the sum of three bits (pair of bits plus carry bit). The transfer of the resulting carry bit forward to the addition of the next significant pair of bits can be easily accomplished by proper wiring of the 3-bit adder gates.

**Signed Binary Numbers** A binary system with $n$ digits can generate a set of $2^n$ different bit patterns. If we have to represent signed numbers in binary code, a natural solution is to split this space into two equal subsets. One subset of codes is assigned to represent positive numbers, and the other negative numbers. Ideally, the coding scheme should be chosen in such a way that the introduction of signed numbers would complicate the hardware implementation as little as possible.

This challenge has led to the development of several coding schemes for representing signed numbers in binary code. The method used today by almost all computers is called the *2's complement* method, also known as *radix complement*. In a binary system with $n$ digits, the 2's complement of the number $x$ is defined as follows:

$$\bar{x} = \begin{cases} 2^n - x & \text{if } x \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

For example, in a 5-bit binary system, the 2's complement representation of $-2$ or "minus$(00010)_{two}$" is $2^5 - (00010)_{two} = (32)_{ten} - (2)_{ten} = (30)_{ten} = (11110)_{two}$. To check the calculation, the reader can verify that $(00010)_{two} + (11110)_{two} = (00000)_{two}$. Note that in the latter computation, the sum is actually $(100000)_{two}$, but since we are dealing with a 5-bit binary system, the left-most sixth bit is simply ignored. As a rule, when the 2's complement method is applied to $n$-bit numbers, $x + (-x)$ always sums up to $2^n$ (i.e., 1 followed by $n$ 0's)—a property that gives the method its name. Figure 2.1 illustrates a 4-bit binary system with the 2's complement method.

An inspection of figure 2.1 suggests that an $n$-bit binary system with 2's complement representation has the following properties:

| Positive numbers | | Negative numbers | |
|---|---|---|---|
| 0 | 0000 | | |
| 1 | 0001 | 1111 | −1 |
| 2 | 0010 | 1110 | −2 |
| 3 | 0011 | 1101 | −3 |
| 4 | 0100 | 1100 | −4 |
| 5 | 0101 | 1011 | −5 |
| 6 | 0110 | 1010 | −6 |
| 7 | 0111 | 1001 | −7 |
| | | 1000 | −8 |

**Figure 2.1** 2's complement representation of signed numbers in a 4-bit binary system.

- The system can code a total of $2^n$ signed numbers, of which the maximal and minimal numbers are $2^{n-1} - 1$ and $-2^{n-1}$, respectively.
- The codes of all positive numbers begin with a 0.
- The codes of all negative numbers begin with a 1.
- To obtain the code of $-x$ from the code of $x$, leave all the trailing (least significant) 0's and the first least significant 1 intact, then flip all the remaining bits (convert 0's to 1's and vice versa). An equivalent shortcut, which is easier to implement in hardware, is to flip all the bits of $x$ and add 1 to the result.

A particularly attractive feature of this representation is that addition of any two signed numbers in 2's complement is exactly the same as addition of positive numbers. Consider, for example, the addition operation $(-2) + (-3)$. Using 2's complement (in a 4-bit representation), we have to add, in binary, $(1110)_{two} + (1101)_{two}$. Without paying any attention to which numbers (positive or negative) these codes represent, bit-wise addition will yield 1011 (after throwing away the overflow bit). As figure 2.1 shows, this indeed is the 2's complement representation of $-5$.

In short, we see that the 2's complement method facilitates the addition of any two signed numbers without requiring special hardware beyond that needed for simple bit-wise addition. What about subtraction? Recall that in the 2's complement method, the arithmetic negation of a signed number $x$, that is, computing $-x$, is achieved by negating all the bits of $x$ and adding 1 to the result. Thus subtraction can be easily handled by $x - y = x + (-y)$. Once again, hardware complexity is kept to a minimum.

The material implications of these theoretical results are significant. Basically, they imply that a single chip, called *Arithmetic Logical Unit*, can be used to encapsulate all the basic arithmetic and logical operators performed in hardware. We now turn to specify one such ALU, beginning with the specification of an *adder* chip.

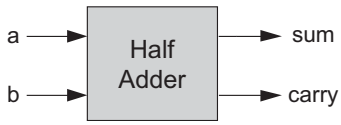
## 2.2 Specification

### 2.2.1 Adders

We present a hierarchy of three adders, leading to a multi-bit adder chip:

- *Half-adder:* designed to add two bits
- *Full-adder:* designed to add three bits
- *Adder:* designed to add two $n$-bit numbers

| Inputs | | Outputs | |
|---|---|---|---|
| a | b | carry | sum |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |




```
Chip name: HalfAdder
Inputs:    a, b
Outputs:   sum, carry
Function:  sum   = LSB of a + b
           carry = MSB of a + b
```

**Figure 2.2** Half-adder, designed to add 2 bits.

We also present a special-purpose adder, called *incrementer*, designed to add 1 to a given number.

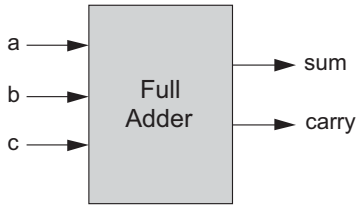
**Half-Adder** The first step on our way to adding binary numbers is to be able to add two bits. Let us call the least significant bit of the addition `sum`, and the most significant bit `carry`. Figure 2.2 presents a chip, called half-adder, designed to carry out this operation.

**Full-Adder** Now that we know how to add two bits, figure 2.3 presents a *full-adder* chip, designed to add three bits. Like the half-adder case, the full-adder chip produces two outputs: the least significant bit of the addition, and the carry bit.

**Adder** Memory and register chips represent integer numbers by *n*-bit patterns, *n* being 16, 32, 64, and so forth—depending on the computer platform. The chip whose job is to add such numbers is called a multi-bit adder, or simply *adder*. Figure 2.4 presents a 16-bit adder, noting that the same logic and specifications scale up as is to any *n*-bit adder.

**Incrementer** It is convenient to have a special-purpose chip dedicated to adding the constant 1 to a given number. Here is the specification of a 16-bit incrementer:

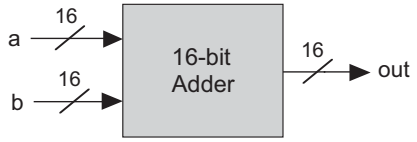
| a | b | c | carry | sum |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |




```
Chip name: FullAdder
Inputs:    a, b, c
Outputs:   sum, carry
Function:  sum = LSB of a + b + c
           carry = MSB of a + b + c
```

**Figure 2.3** Full-adder, designed to add 3 bits.

```
...  1  0  1  1  a
...  0  0  1  0  b   +
-------------------
...  1  1  0  1  out
```




```
Chip name: Add16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  out = a + b
Comment:   Integer 2's complement addition.
           Overflow is neither detected nor handled.
```

**Figure 2.4** 16-bit adder. Addition of two *n*-bit binary numbers for any *n* is "more of the same."

```
Chip name: Inc16
Inputs:    in[16]
Outputs:   out[16]
Function:  out=in+1
Comment:   Integer 2's complement addition.
           Overflow is neither detected nor handled.
```

### 2.2.2 The Arithmetic Logic Unit (ALU)

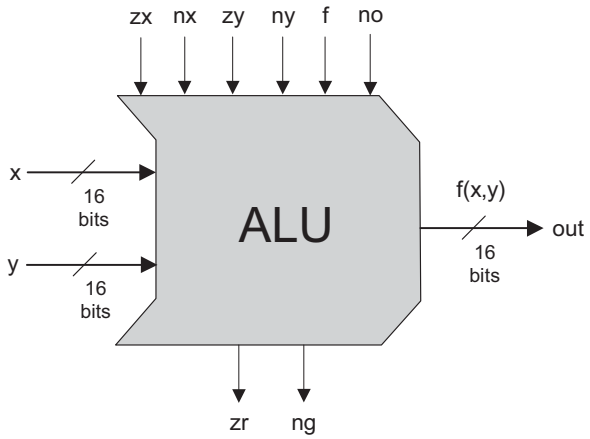
The specifications of the adder chips presented so far were generic, meaning that they hold for any computer. In contrast, this section describes an ALU that will later become the centerpiece of a specific computer platform called *Hack*. At the same time, the principles underlying the design of our ALU are rather general. Further, our ALU architecture achieves a great deal of functionality using a minimal set of internal parts. In that respect, it provides a good example of an efficient and elegant logic design.

The Hack ALU computes a fixed set of functions $out = f_i(x, y)$ where $x$ and $y$ are the chip's two 16-bit inputs, *out* is the chip's 16-bit output, and $f_i$ is an arithmetic or logical function selected from a fixed repertoire of eighteen possible functions. We instruct the ALU which function to compute by setting six input bits, called *control bits*, to selected binary values. The exact input-output specification is given in figure 2.5, using pseudo-code.

Note that each one of the six control bits instructs the ALU to carry out a certain elementary operation. Taken together, the combined effects of these operations cause the ALU to compute a variety of useful functions. Since the overall operation is driven by six control bits, the ALU can potentially compute $2^6 = 64$ different functions. Eighteen of these functions are documented in figure 2.6.

We see that programming our ALU to compute a certain function $f(x, y)$ is done by setting the six control bits to the code of the desired function. From this point on, the internal ALU logic specified in figure 2.5 should cause the ALU to output the value $f(x, y)$ specified in figure 2.6. Of course, this does not happen miraculously, it's the result of careful design.

For example, let us consider the twelfth row of figure 2.6, where the ALU is instructed to compute the function `x-1`. The `zx` and `nx` bits are 0, so the `x` input is neither zeroed nor negated. The `zy` and `ny` bits are 1, so the `y` input is first zeroed, and then negated bit-wise. Bit-wise negation of zero, $(000\ldots00)_{\text{two}}$, gives $(111\ldots11)_{\text{two}}$, the 2's complement code of $-1$. Thus the ALU inputs end up being `x`




```
Chip name: ALU
Inputs:    x[16], y[16],    // Two 16-bit data inputs
           zx,              // Zero the x input
           nx,              // Negate the x input
           zy,              // Zero the y input
           ny,              // Negate the y input
           f,               // Function code: 1 for Add, 0 for And
           no               // Negate the out output
Outputs:   out[16],         // 16-bit output
           zr,              // True iff out=0
           ng               // True iff out<0
Function:  if zx then x = 0       // 16-bit zero constant
           if nx then x = !x      // Bit-wise negation
           if zy then y = 0       // 16-bit zero constant
           if ny then y = !y      // Bit-wise negation
           if f then out = x + y  // Integer 2's complement addition
                else out = x & y  // Bit-wise And
           if no then out = !out  // Bit-wise negation
           if out=0 then zr = 1 else zr = 0  // 16-bit eq. comparison
           if out<0 then ng = 1 else ng = 0  // 16-bit neg. comparison
Comment:   Overflow is neither detected nor handled.
```

**Figure 2.5** The Arithmetic Logic Unit.

| These bits instruct how to preset the x input | | These bits instruct how to preset the y input | | This bit selects between + / And | This bit inst. how to postset out | Resulting ALU output |
|---|---|---|---|---|---|---|
| zx | nx | zy | ny | f | no | out= |
| if zx then x=0 | if nx then x=!x | if zy then y=0 | if ny then y=!y | if f then out=x+y else out=x&y | if no then out=!out | f(x,y)= |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | -1 |
| 0 | 0 | 1 | 1 | 0 | 0 | x |
| 1 | 1 | 0 | 0 | 0 | 0 | y |
| 0 | 0 | 1 | 1 | 0 | 1 | !x |
| 1 | 1 | 0 | 0 | 0 | 1 | !y |
| 0 | 0 | 1 | 1 | 1 | 1 | -x |
| 1 | 1 | 0 | 0 | 1 | 1 | -y |
| 0 | 1 | 1 | 1 | 1 | 1 | x+1 |
| 1 | 1 | 0 | 1 | 1 | 1 | y+1 |
| 0 | 0 | 1 | 1 | 1 | 0 | x-1 |
| 1 | 1 | 0 | 0 | 1 | 0 | y-1 |
| 0 | 0 | 0 | 0 | 1 | 0 | x+y |
| 0 | 1 | 0 | 0 | 1 | 1 | x-y |
| 0 | 0 | 0 | 1 | 1 | 1 | y-x |
| 0 | 0 | 0 | 0 | 0 | 0 | x&y |
| 0 | 1 | 0 | 1 | 0 | 1 | x\|y |

**Figure 2.6** The ALU truth table. Taken together, the binary operations coded by the first six columns effect the function listed in the right column (we use the symbols !, &, and | to represent the operators Not, And, and Or, respectively, performed bit-wise). The complete ALU truth table consists of sixty-four rows, of which only the eighteen presented here are of interest.

and −1. Since the `f`-bit is 1, the selected operation is *arithmetic addition*, causing the ALU to calculate `x+(-1)`. Finally, since the `no` bit is 0, the output is not negated but rather left as is. To conclude, the ALU ends up computing `x-1`, which was our goal.

Does the ALU logic described in figure 2.6 compute every one of the other seventeen functions listed in the figure's right column? To verify that this is indeed the case, the reader can pick up some other rows in the table and prove their respective ALU operation. We note that some of these computations, beginning with the

function $f(x, y) = 1$, are not trivial. We also note that there are some other useful functions computed by the ALU but not listed in the figure.

It may be instructive to describe the thought process that led to the design of this particular ALU. First, we made a list of all the primitive operations that we wanted our computer to be able to perform (right column in figure 2.6). Next, we used backward reasoning to figure out how `x`, `y`, and `out` can be manipulated in binary fashion in order to carry out the desired operations. These processing requirements, along with our objective to keep the ALU logic as simple as possible, have led to the design decision to use six control bits, each associated with a straightforward binary operation. The resulting ALU is simple and elegant. And in the hardware business, simplicity and elegance imply inexpensive and powerful computer systems.

## 2.3 Implementation

Our implementation guidelines are intentionally partial, since we want you to discover the actual chip architectures yourself. As usual, each chip can be implemented in more than one way; the simpler the implementation, the better.

**Half-Adder** An inspection of figure 2.2 reveals that the functions $\text{sum}(a, b)$ and $\text{carry}(a, b)$ happen to be identical to the standard $\text{Xor}(a, b)$ and $\text{And}(a, b)$ Boolean functions. Thus, the implementation of this adder is straightforward, using previously built gates.

**Full-Adder** A full adder chip can be implemented from two half adder chips and one additional simple gate. A direct implementation is also possible, without using half-adder chips.

**Adder** The addition of two signed numbers represented by the 2's complement method as two $n$-bit buses can be done bit-wise, from right to left, in $n$ steps. In step 0, the least significant pair of bits is added, and the carry bit is fed into the addition of the next significant pair of bits. The process continues until in step $n - 1$ the most significant pair of bits is added. Note that each step involves the addition of three bits. Hence, an $n$-bit adder can be implemented by creating an array of $n$ full-adder chips and propagating the carry bits up the significance ladder.

**Incrementer** An $n$-bit incrementer can be implemented trivially from an $n$-bit adder.

**ALU** Note that our ALU was carefully planned to effect all the desired ALU operations *logically*, using simple Boolean operations. Therefore, the *physical* implementation of the ALU is reduced to implementing these simple Boolean operations, following their pseudo-code specifications. Your first step will likely be to create a logic circuit that manipulates a 16-bit input according to the `nx` and `zx` control bits (i.e., the circuit should conditionally zero and negate the 16-bit input). This logic can be used to manipulate the `x` and `y` inputs, as well as the `out` output. Chips for bitwise And-ing and addition have already been built in this and in the previous chapter. Thus, what remains is to build logic that chooses between them according to the `f` control bit. Finally, you will need to build logic that integrates all the other chips into the overall ALU. (When we say "build logic," we mean "write HDL code").

## 2.4 Perspective

The construction of the multi-bit adder presented in this chapter was standard, although no attention was paid to efficiency. In fact, our suggested adder implementation is rather inefficient, due to the long delays incurred while the carry bit propagates from the least significant bit pair to the most significant bit pair. This problem can be alleviated using logic circuits that effect so-called carry look-ahead techniques. Since addition is one of the most prevalent operations in any given hardware platform, any such low-level improvement can result in dramatic and global performance gains throughout the computer.

In any given computer, the overall functionality of the hardware/software platform is delivered jointly by the ALU and the operating system that runs on top of it. Thus, when designing a new computer system, the question of how much functionality the ALU should deliver is essentially a cost/performance issue. The general rule is that hardware implementations of arithmetic and logical operations are usually more costly, but achieve better performance. The design trade-off that we have chosen in this book is to specify an ALU hardware with a limited functionality and then implement as many operations as possible in software. For example, our ALU features neither multiplication nor division nor floating point arithmetic. We will implement some of these operations (as well as more mathematical functions) at the operating system level, described in chapter 12.

Detailed treatments of Boolean arithmetic and ALU design can be found in most computer architecture textbooks.

## 2.5 Project

**Objective** Implement all the chips presented in this chapter. The only building blocks that you can use are the chips that you will gradually build and the chips described in the previous chapter.

**Tip** When your HDL programs invoke chips that you may have built in the previous project, we recommend that you use the built-in versions of these chips instead. This will ensure correctness and speed up the operation of the hardware simulator. There is a simple way to accomplish this convention: Make sure that your project directory includes only the `.hdl` files that belong to the present project.

The remaining instructions for this project are identical to those of the project from the previous chapter, except that the last step should be replaced with "Build and simulate all the chips specified in the `projects/02` directory."

# 3 Sequential Logic

*It's a poor sort of memory that only works backward.*
—Lewis Carroll (1832–1898)

All the Boolean and arithmetic chips that we built in chapters 1 and 2 were *combinational*. Combinational chips compute functions that depend solely on *combinations* of their input values. These relatively simple chips provide many important processing functions (like the ALU), but they cannot *maintain state*. Since computers must be able to not only compute values but also store and recall values, they must be equipped with memory elements that can preserve data over time. These memory elements are built from *sequential chips*.

The implementation of memory elements is an intricate art involving synchronization, clocking, and feedback loops. Conveniently, most of this complexity can be embedded in the operating logic of very low-level sequential gates called *flip-flops*. Using these flip-flops as elementary building blocks, we will specify and build all the memory devices employed by typical modern computers, from binary cells to registers to memory banks and counters. This effort will complete the construction of the chip set needed to build an entire computer—a challenge that we take up in the chapter 5.

Following a brief overview of clocks and flip-flops, the Background section introduces all the memory chips that we will build on top of them. The next two sections describe the chips Specification and Implementation, respectively. As usual, all the chips mentioned in the chapter can be built and tested using the hardware simulator supplied with the book, following the instructions given in the final Project section.

## 3.1 Background

The act of "remembering something" is inherently time-dependent: You remember *now* what has been committed to memory *before*. Thus, in order to build chips that "remember" information, we must first develop some standard means for representing the progression of time.

**The Clock** In most computers, the passage of time is represented by a master clock that delivers a continuous train of alternating signals. The exact hardware implementation is usually based on an oscillator that alternates continuously between two phases labeled 0–1, *low-high*, *tick-tock*, etc. The elapsed time between the beginning of a "tick" and the end of the subsequent "tock" is called *cycle*, and each clock cycle is taken to model one discrete time unit. The current clock phase (*tick* or *tock*) is represented by a binary signal. Using the hardware's circuitry, this signal is simultaneously broadcast to every sequential chip throughout the computer platform.

**Flip-Flops** The most elementary sequential element in the computer is a device called a *flip-flop*, of which there are several variants. In this book we use a variant called a *data flip-flop*, or DFF, whose interface consists of a single-bit data input and a single-bit data output. In addition, the DFF has a *clock* input that continuously changes according to the master clock's signal. Taken together, the data and the clock inputs enable the DFF to implement the time-based behavior $out(t) = in(t-1)$, where *in* and *out* are the gate's input and output values and $t$ is the current clock cycle. In other words, the DFF simply outputs the input value from the previous time unit.

As we now show, this elementary behavior can form the basis of all the hardware devices that computers use to *maintain state*, from binary cells to registers to arbitrarily large random access memory (RAM) units.

**Registers** A *register* is a storage device that can "store," or "remember," a value over time, implementing the classical storage behavior $out(t) = out(t-1)$. A DFF, on the other hand, can only output its previous input, namely, $out(t) = in(t-1)$. This suggests that a register can be implemented from a DFF by simply feeding the output of the latter back into its input, creating the device shown in the middle of figure 3.1. Presumably, the output of this device at any time $t$ will echo its output at time $t-1$, yielding the classical function expected from a storage unit.

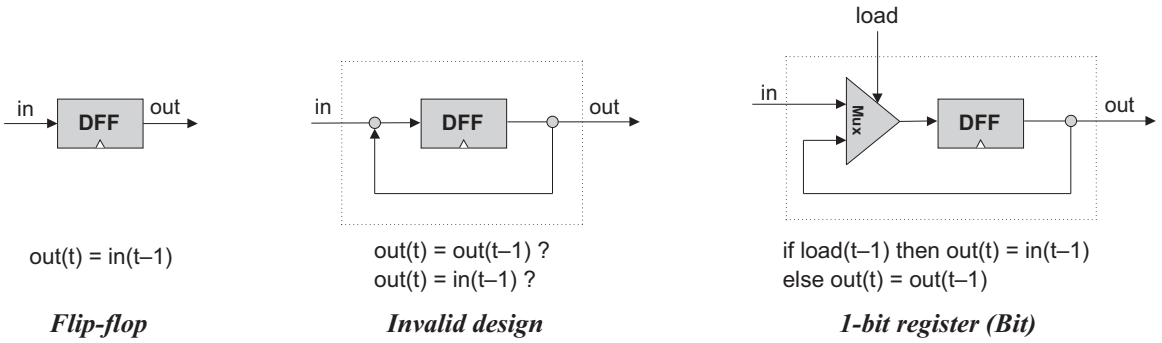


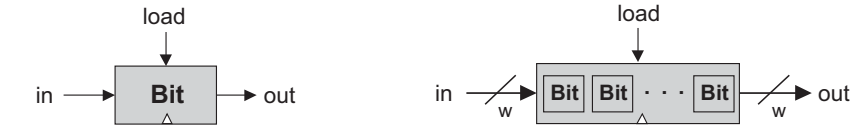

**Figure 3.1** From a DFF to a single-bit register. The small triangle represents the clock input. This icon is used to state that the marked chip, as well as the overall chip that encapsulates it, is time-dependent.

Well, not so. The device shown in the middle of figure 3.1 is invalid. First, it is not clear how we'll ever be able to load this device with a new data value, since there are no means to tell the DFF when to draw its input from the `in` wire and when from the `out` wire. More generally, the rules of chip design dictate that internal pins must have a fan-in of 1, meaning that they can be fed from a single source only.

The good thing about this thought experiment is that it leads us to the correct and elegant solution shown in the right side of figure 3.1. In particular, a natural way to resolve our input ambiguity is to introduce a multiplexor into the design. Further, the "select bit" of this multiplexor can become the "load bit" of the overall register chip: If we want the register to start storing a new value, we can put this value in the `in` input and set the `load` bit to 1; if we want the register to keep storing its internal value until further notice, we can set the `load` bit to 0.

Once we have developed the basic mechanism for remembering a single bit over time, we can easily construct arbitrarily wide registers. This can be achieved by forming an array of as many single-bit registers as needed, creating a register that holds multi-bit values (figure 3.2). The basic design parameter of such a register is its *width*—the number of bits that it holds—e.g., 16, 32, or 64. The multi-bit contents of such registers are typically referred to as *words*.

**Memories** Once we have the basic ability to represent words, we can proceed to build memory banks of arbitrary length. As figure 3.3 shows, this can be done by stacking together many registers to form a *Random Access Memory* RAM unit. The term *random access memory* derives from the requirement that read/write operations




if load(t–1) then out(t) = in(t–1)
else out(t) = out(t–1)

***Binary cell (Bit)***

if load(t–1) then out(t) = in(t–1)
else out(t) = out(t–1)

***w-bit register***

**Figure 3.2** From single-bit to multi-bit registers. A multi-bit register of width $w$ can be constructed from an array of $w$ 1-bit chips. The operating functions of both chips is exactly the same, except that the "=" assignments are single-bit and multi-bit, respectively.

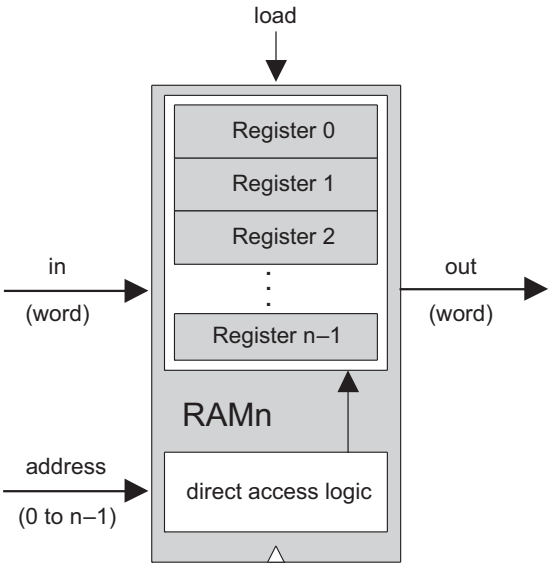



**Figure 3.3** RAM chip (conceptual). The width and length of the RAM can vary.

on a RAM should be able to access randomly chosen words, with no restrictions on the order in which they are accessed. That is to say, we require that any word in the memory—irrespective of its physical location—be accessed directly, in equal speed.

This requirement can be satisfied as follows. First, we assign each word in the $n$-register RAM a unique *address* (an integer between 0 to $n - 1$), according to which it will be accessed. Second, in addition to building an array of $n$ registers, we build a gate logic design that, given an address $j$, is capable of selecting the individual register whose address is $j$. Note however that the notion of an "address" is not an explicit part of the RAM design, since the registers are not "marked" with addresses in any physical sense. Rather, as we will see later, the chip is equipped with direct access logic that implements the notion of addressing using logical means.

In sum, a classical RAM device accepts three inputs: a data input, an address input, and a load bit. The *address* specifies which RAM register should be accessed in the current time unit. In the case of a read operation (`load=0`), the RAM's output immediately emits the value of the selected register. In the case of a write operation (`load=1`), the selected memory register commits to the input value in the next time unit, at which point the RAM's output will start emitting it.

The basic design parameters of a RAM device are its data *width*—the width of each one of its words, and its *size*—the number of words in the RAM. Modern computers typically employ 32- or 64-bit-wide RAMs whose sizes are up to hundreds of millions.

**Counters** A counter is a sequential chip whose state is an integer number that increments every time unit, effecting the function $out(t) = out(t - 1) + c$, where $c$ is typically 1. Counters play an important role in digital architectures. For example, a typical CPU includes a *program counter* whose output is interpreted as the address of the instruction that should be executed next in the current program.

A counter chip can be implemented by combining the input/output logic of a standard register with the combinatorial logic for adding a constant. Typically, the counter will have to be equipped with some additional functionality, such as possibilities for resetting the count to zero, loading a new counting base, or decrementing instead of incrementing.

**Time Matters** All the chips described so far in this chapter are *sequential*. Simply stated, a sequential chip is a chip that embeds one or more DFF gates, either directly or indirectly. Functionally speaking, the DFF gates endow sequential chips with the ability to either maintain state (as in memory units) or operate on state (as in

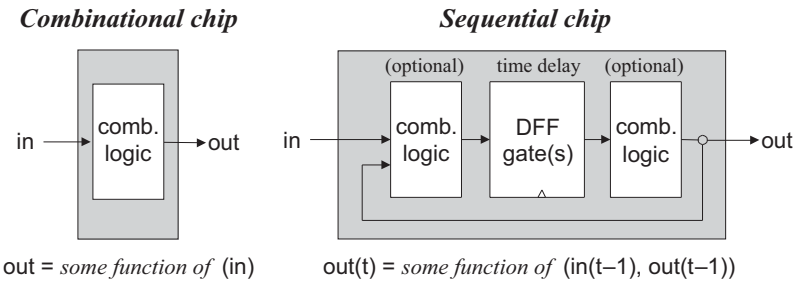



**Figure 3.4** Combinational versus sequential logic (in and out stand for one or more input and output variables). Sequential chips always consist of a layer of DFFs sandwiched between optional combinational logic layers.

counters). Technically speaking, this is done by forming feedback loops inside the sequential chip (see figure 3.4). In combinational chips, where time is neither modeled nor recognized, the introduction of feedback loops is problematic: The output would depend on the input, which itself would depend on the output, and thus the output would depend on itself. On the other hand, there is no difficulty in feeding the output of a sequential chip back into itself, since the DFFs introduce an inherent time delay: The output at time $t$ does not depend on itself, but rather on the output at time $t-1$. This property guards against the uncontrolled "data races" that would occur in combinational chips with feedback loops.

Recall that the outputs of combinational chips change when their inputs change, irrespective of time. In contrast, the inclusion of the DFFs in the sequential architecture ensures that their outputs change only at the point of transition from one clock cycle to the next, and not within the cycle itself. In fact, we allow sequential chips to be in unstable states *during* clock cycles, requiring only that at the beginning of the next cycle they output correct values.

This "discretization" of the sequential chips' outputs has an important side effect: It can be used to synchronize the overall computer architecture. To illustrate, suppose we instruct the arithmetic logic unit (ALU) to compute $x+y$ where $x$ is the value of a nearby register and $y$ is the value of a remote RAM register. Because of various physical constraints (distance, resistance, interference, random noise, etc.) the electric signals representing $x$ and $y$ will likely arrive at the ALU at different times. However, being a *combinational chip*, the ALU is insensitive to the concept of time—it continuously adds up whichever data values happen to lodge in its inputs. Thus, it will take some time before the ALU's output stabilizes to the correct $x+y$ result. Until then, the ALU will generate garbage.

How can we overcome this difficulty? Well, since the output of the ALU is always routed to some sort of a sequential chip (a register, a RAM location, etc.), *we don't really care*. All we have to do is ensure, when we build the computer's clock, that the length of the clock cycle will be slightly longer that the time it takes a bit to travel the longest distance from one chip in the architecture to another. This way, we are guaranteed that by the time the sequential chip updates its state (at the beginning of the next clock cycle), the inputs that it receives from the ALU will be valid. This, in a nutshell, is the trick that synchronizes a set of stand-alone hardware components into a well-coordinated system, as we shall see in chapter 5.

## 3.2 Specification

This section specifies a hierarchy of sequential chips:

- Data-flip-flops (DFFs)
- Registers (based on DFFs)
- Memory banks (based on registers)
- Counter chips (also based on registers)

### 3.2.1 Data-Flip-Flop

The most elementary sequential device that we present—the basic component from which all memory elements will be designed—is the *data flip-flop* gate. A DFF gate has a single-bit input and a single-bit output, as follows:


```
Chip name: DFF
Inputs:    in
Outputs:   out
Function:  out(t)=in(t-1)
Comment:   This clocked gate has a built-in
           implementation and thus there is
           no need to implement it.
```

Like Nand gates, DFF gates enter our computer archtecture at a very low level. Specifically, all the sequential chips in the computer (registers, memory, and

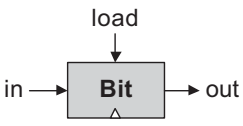
counters) are based on numerous DFF gates. All these DFFs are connected to the same master clock, forming a huge distributed "chorus line." At the beginning of each clock cycle, the outputs of *all* the DFFs in the computer commit to their inputs during the previous time unit. At all other times, the DFFs are "latched," meaning that changes in their inputs have no immediate effect on their outputs. This conduction operation effects any one of the millions of DFF gates that make up the system, about a billion times per second (depending on the computer's clock frequency).

Hardware implementations achieve this time dependency by simultaneously feeding the master clock signal to all the DFF gates in the platform. Hardware simulators emulate the same effect in software. As far as the computer architect is concerned, the end result is the same: The inclusion of a DFF gate in the design of any chip ensures that the overall chip, as well as all the chips up the hardware hierarchy that depend on it, will be inherently time-dependent. These chips are called *sequential*, by definition.

The physical implementation of a DFF is an intricate task, and is based on connecting several elementary logic gates using feedback loops (one classic design is based on Nand gates alone). In this book we have chosen to abstract away this complexity, treating DFFs as primitive building blocks. Thus, our hardware simulator provides a built-in DFF implementation that can be readily used by other chips.

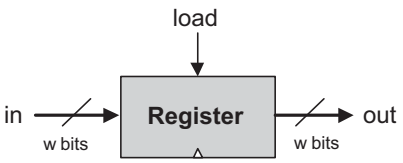
### 3.2.2 Registers

A single-bit register, which we call Bit, or *binary cell*, is designed to store a single bit of information (0 or 1). The chip interface consists of an input pin that carries a data bit, a `load` pin that enables the cell for writes, and an output pin that emits the current state of the cell. The interface diagram and API of a binary cell are as follows:




```
Chip name: Bit
Inputs:    in, load
Outputs:   out
Function:  If load(t-1) then out(t)=in(t-1)
           else out(t)=out(t-1)
```

The API of the Register chip is essentially the same, except that the input and output pins are designed to handle multi-bit values:




```
Chip name: Register
Inputs:    in[16], load
Outputs:   out[16]
Function:  If load(t-1) then out(t)=in(t-1)
           else out(t)=out(t-1)
Comment:   "=" is a 16-bit operation.
```

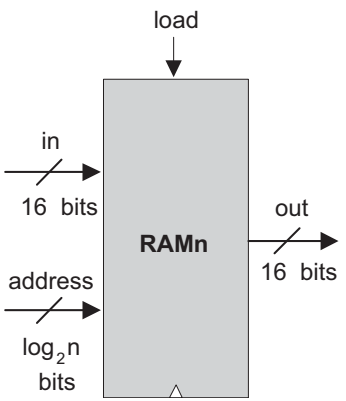
The Bit and Register chips have exactly the same read/write behavior:

*Read:* To read the contents of a register, we simply probe its output.

*Write:* To write a new data value *d* into a register, we put *d* in the `in` input and assert (set to 1) the `load` input. In the next clock cycle, the register commits to the new data value, and its output starts emitting *d*.

### 3.2.3 Memory

A direct-access memory unit, also called RAM, is an array of *n* *w*-bit registers, equipped with direct access circuitry. The number of registers (*n*) and the width of each register (*w*) are called the memory's *size* and *width*, respectively. We will now set out to build a hierarchy of such memory devices, all 16 bits wide, but with varying sizes: RAM8, RAM64, RAM512, RAM4K, and RAM16K units. All these memory chips have precisely the same API, and thus we describe them in one parametric diagram:




```
Chip name: RAMn // n and k are listed below
Inputs:    in[16], address[k], load
Outputs:   out[16]
Function:  out(t)=RAM[address(t)](t)
           If load(t-1) then
              RAM[address(t-1)](t)=in(t-1)
Comment:   "=" is a 16-bit operation.
```

**The specific RAM chips needed for the Hack platform are:**

| Chip name | n | k |
|---|---|---|
| RAM8 | 8 | 3 |
| RAM64 | 64 | 6 |
| RAM512 | 512 | 9 |
| RAM4K | 4096 | 12 |
| RAM16K | 16384 | 14 |

*Read:* To read the contents of register number *m*, we put *m* in the `address` input. The RAM's direct-access logic will select register number *m*, which will then emit its output value to the RAM's output pin. This is a combinational operation, independent of the clock.

*Write:* To write a new data value *d* into register number *m*, we put *m* in the `address` input, *d* in the `in` input, and assert the `load` input bit. This causes the RAM's direct-access logic to select register number *m*, and the `load` bit to enable it. In the next clock cycle, the selected register will commit to the new value (*d*), and the RAM's output will start emitting it.

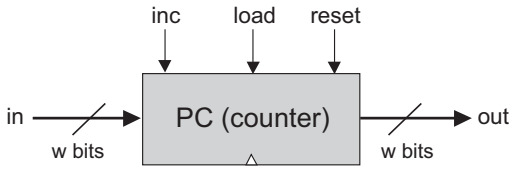
### 3.2.4 Counter

Although a *counter* is a stand-alone abstraction in its own right, it is convenient to motivate its specification by saying a few words about the context in which it is normally used. For example, consider a counter chip designed to contain the address of the instruction that the computer should fetch and execute next. In most cases, the counter has to simply increment itself by 1 in each clock cycle, thus causing the computer to fetch the next instruction in the program. In other cases, for example, in "jump to execute instruction number *n*," we want to be able to set the counter to *n*, then have it continue its default counting behavior with $n + 1$, $n + 2$, and so forth. Finally, the program's execution can be restarted anytime by resetting the counter to 0, assuming that that's the address of the program's first instruction. In short, we need a loadable and resettable counter.

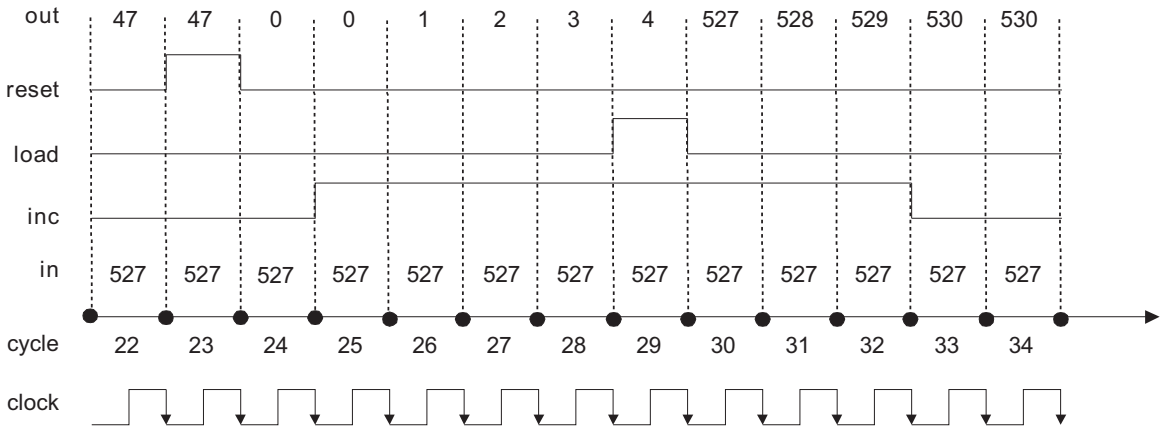
With that in mind, the interface of our Counter chip is similar to that of a register, except that it has two additional control bits labeled `reset` and `inc`. When `inc=1`, the counter increments its state in every clock cycle, emitting the value `out(t)=out(t-1)+1`. If we want to reset the counter to 0, we assert the `reset` bit; if we want to initialize it to some other counting base *d*, we put *d* in the `in` input and assert the `load` bit. The details are given in the counter API, and an example of its operation is depicted in figure 3.5.

## 3.3 Implementation

**Flip-Flop** DFF gates can be implemented from lower-level logic gates like those built in chapter 1. However, in this book we treat DFFs as primitive gates, and thus they can be used in hardware construction projects without worrying about their internal implementation.




```
Chip name: PC  // 16-bit counter
Inputs:    in[16], inc, load, reset
Outputs:   out[16]
Function:  If reset(t-1) then out(t)=0
              else if load(t-1) then out(t)=in(t-1)
                   else if inc(t-1) then out(t)=out(t-1)+1
                        else out(t)=out(t-1)
Comment:   "=" is 16-bit assignment.
           "+" is 16-bit arithmetic addition.
```




We assume that we start tracking the counter in time unit 22, when its input and output happen to be 527 and 47, respectively. We also assume that the counter's control bits (reset, load, inc) start at 0—all arbitrary assumptions.

**Figure 3.5** Counter simulation. At time 23 a *reset* signal is issued, causing the counter to emit 0 in the following time unit. The 0 persists until an *inc* signal is issued at time 25, causing the counter to start incrementing, one time unit later. The counting continues until, at time 29, the load bit is asserted. Since the counter's input holds the number 527, the counter is reset to that value in the next time unit. Since inc is still asserted, the counter continues incrementing until time 33, when inc is de-asserted.

**1-Bit Register (Bit)** The implementation of this chip was given in figure 3.1.

**Register** The construction of a *w*-bit Register chip from 1-bit registers is straightforward. All we have to do is construct an array of *w* Bit gates and feed the register's `load` input to every one of them.

**8-Register Memory (RAM8)** An inspection of figure 3.3 may be useful here. To implement a RAM8 chip, we line up an array of eight registers. Next, we have to build combinational logic that, given a certain `address` value, takes the RAM8's `in` input and loads it into the selected register. In a similar fashion, we have to build combinational logic that, given a certain `address` value, selects the right register and pipes its `out` value to the RAM8's `out` output. Tip: This combinational logic was already implemented in chapter 1.

***n*-Register Memory** A memory bank of arbitrary length (a power of 2) can be built recursively from smaller memory units, all the way down to the single register level. This view is depicted in figure 3.6. Focusing on the right-hand side of the figure, we note that a 64-register RAM can be built from an array of eight 8-register RAM chips. To select a particular register from the RAM64 memory, we use a 6-bit address, say *xxxyyy*. The MSB *xxx* bits select one of the RAM8 chips, and the LSB *yyy* bits select one of the registers within the selected RAM8. The RAM64 chip should be equipped with logic circuits that effect this hierarchical addressing scheme.

**Counter** A *w*-bit counter consists of two main elements: a regular *w*-bit register, and combinational logic. The combinational logic is designed to (a) compute the counting function, and (b) put the counter in the right operating mode, as mandated by the values of its three control bits. Tip: Most of this logic was already built in chapter 2.

## 3.4 Perspective

The cornerstone of all the memory systems described in this chapter is the flip-flop—a gate that we treated here as an atomic, or primitive, building block. The usual approach in hardware textbooks is to construct flip-flops from elementary combinatorial gates (e.g., Nand gates) using appropriate feedback loops. The standard con-

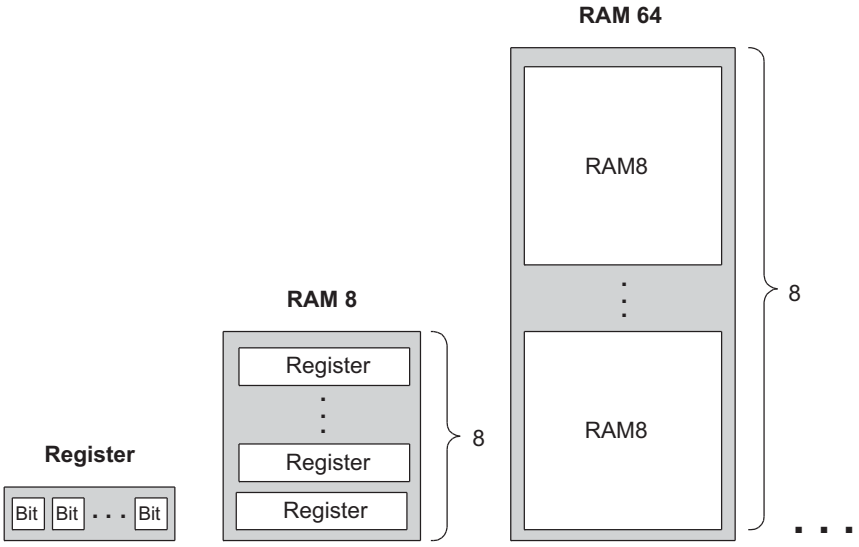



**Figure 3.6** Gradual construction of memory banks by recursive ascent. A *w*-bit register is an array of *w* binary cells, an 8-register RAM is an array of eight *w*-bit registers, a 64-register RAM is an array of eight RAM8 chips, and so on. Only three more similar construction steps are necessary to build a 16K RAM chip.

struction begins by building a simple (non-clocked) flip-flop that is bi-stable, namely, that can be set to be in one of two states. Then a clocked flip-flop is obtained by cascading two such simple flip-flops, the first being set when the clock *ticks* and the second when the clock *tocks*. This "master-slave" design endows the overall flip-flop with the desired clocked synchronization functionality.

These constructions are rather elaborate, requiring an understating of delicate issues like the effect of feedback loops on combinatorial circuits, as well as the implementation of clock cycles using a two-phase binary clock signal. In this book we have chosen to abstract away these low-level considerations by treating the flip-flop as an atomic gate. Readers who wish to explore the internal structure of flip-flop gates can find detailed descriptions in most logic design and computer architecture textbooks.

In closing, we should mention that memory devices of modern computers are not always constructed from standard flip-flops. Instead, modern memory chips are usually very carefully optimized, exploiting the unique physical properties of the underlying storage technology. Many such alternative technologies are available

today to computer designers; as usual, which technology to use is a cost-performance issue.

Aside from these low-level considerations, all the other chip constructions in this chapter—the registers and memory chips that were built on top of the flip-flop gates—were standard.

## 3.5 Project

**Objective** Build all the chips described in the chapter. The only building blocks that you can use are primitive DFF gates, chips that you will build on top of them, and chips described in previous chapters.

**Resources** The only tool that you need for this project is the hardware simulator supplied with the book. All the chips should be implemented in the HDL language specified in appendix A. As usual, for each chip we supply a skeletal `.hdl` program with a missing implementation part, a `.tst` script file that tells the hardware simulator how to test it, and a `.cmp` compare file. Your job is to complete the missing implementation parts of the supplied `.hdl` programs.

**Contract** When loaded into the hardware simulator, your chip design (modified `.hdl` program), tested on the supplied `.tst` file, should produce the outputs listed in the supplied `.cmp` file. If that is not the case, the simulator will let you know.

**Tip** The Data Flip-Flop (DFF) gate is considered primitive and thus there is no need to build it: When the simulator encounters a DFF gate in an HDL program, it automatically invokes the built-in `tools/builtIn/DFF.hdl` implementation.

**The Directory Structure of This Project** When constructing RAM chips from smaller ones, we recommend using built-in versions of the latter. Otherwise, the simulator may run very slowly or even out of (real) memory space, since large RAM chips contain tens of thousands of lower-level chips, and all these chips are kept in memory (as software objects) by the simulator. For this reason, we have placed the `RAM512.hdl`, `RAM4K.hdl`, and `RAM16K.hdl` programs in a separate directory. This way, the recursive descent construction of the RAM4K and RAM16K chips stops with the RAM512 chip, whereas the lower-level chips from which the latter chip

is made are bound to be built-in (since the simulator does not find them in this directory).

**Steps** We recommend proceeding in the following order:

0. The hardware simulator needed for this project is available in the `tools` directory of the book's software suite.
1. Read appendix A, focusing on sections A.6 and A.7.
2. Go through the *hardware simulator tutorial*, focusing on parts IV and V.
3. Build and simulate all the chips specified in the `projects/03` directory.

# 4 Machine Language

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

A computer can be described *constructively*, by laying out its hardware platform and explaining how it is built from low-level chips. A computer can also be described *abstractly*, by specifying and demonstrating its machine language capabilities. And indeed, it is convenient to get acquainted with a new computer system by first seeing some low-level programs written in its machine language. This helps us understand not only how to program the computer to do useful things, but also why its hardware was designed in a certain way. With that in mind, this chapter focuses on low-level programming in machine language. This sets the stage for chapter 5, where we complete the construction of a general-purpose computer designed to run machine language programs. This computer will be constructed from the chip set built in chapters 1–3.

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

Machine language is the most profound interface in the overall computer enterprise—the fine line where hardware and software meet. This is the point where the abstract thoughts of the programmer, as manifested in symbolic instructions, are turned into physical operations performed in silicon. Thus, machine language can

be construed as both a programming tool and an integral part of the hardware platform. In fact, just as we say that the machine language is designed to exploit a given hardware platform, we can say that the hardware platform is designed to fetch, interpret, and execute instructions written in the given machine language.

The chapter begins with a general introduction to machine language programming. Next, we give a detailed specification of the Hack machine language, covering both its binary and its symbolic assembly versions. The project that ends the chapter engages you in writing a couple of machine language programs. This project offers a hands-on appreciation of low-level programming and prepares you for building the computer itself in the next chapter.

Although most people will never write programs directly in machine language, the study of low-level programming is a prerequisite to a complete understanding of computer architectures. Also, it is rather fascinating to realize how the most sophisticated software systems are, at bottom, long series of elementary instructions, each specifying a very simple and primitive operation on the underlying hardware. As usual, this understanding is best achieved constructively, by writing some low-level code and running it directly on the hardware platform.

## 4.1 Background

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

### 4.1.1 Machines

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

**Memory** The term *memory* refers loosely to the collection of hardware devices that store data and instructions in a computer. From the programmer's standpoint, all memories have the same structure: A continuous array of cells of some fixed width, also called *words* or *locations*, each having a unique *address*. Hence, an individual word (representing either a data item or an instruction) is specified by supplying its

address. In what follows we will refer to such individual words using the equivalent notation Memory[address], RAM[address], or M[address] for brevity.

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

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

### 4.1.2 Languages

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

Since binary codes are rather cryptic, machine languages are normally specified using both binary codes and symbolic mnemonics (a *mnemonic* is a symbolic label whose name hints at what it stands for—in our case hardware elements and binary operations). For example, the language designer can decide that the operation code `1010` will be represented by the mnemonic `add` and that the registers of the machine will be symbolically referred to using the symbols `R0`, `R1`, `R2`, and so forth. Using these conventions, one can specify machine language instructions either directly, as `1010001100011001`, or symbolically, as, say, `ADD R3,R1,R9`.

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

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

### 4.1.3 Commands

**Arithmetic and Logic Operations** Every computer is required to perform basic arithmetic operations like addition and subtraction as well as basic Boolean operations like bit-wise negation, bit shifting, and so forth. Here are some examples, written in typical machine language syntax:

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

ADD R2,R1,foo // R2←R1+foo where foo stands for the
              // value of the memory location pointed
              // at by the user-defined label foo.

AND R1,R1,R2  // R1←bit wise And of R1 and R2
```

**Memory Access** Memory access commands fall into two categories. First, as we have just seen, arithmetic and logical commands are allowed to operate not only on registers, but also on selected memory locations. Second, all computers feature explicit *load* and *store* commands, designed to move data between registers and memory. These memory access commands may use several types of *addressing modes*—ways of specifying the address of the required memory word. As usual, different computers offer different possibilities and different notations, but the following three memory access modes are almost always supported:

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

```
LOAD R1,67      //  R1←Memory[67]

// Or, assuming that bar refers to memory address 67:

LOAD R1,bar     //  R1←Memory[67]
```

- *Immediate addressing* This form of addressing is used to load constants—namely, load values that appear in the instruction code: Instead of treating the numeric field that appears in the instruction as an address, we simply load the value of the field itself into the register, as follows:

```
LOADI R1,67     //  R1←67
```

- *Indirect addressing* In this addressing mode the address of the required memory location is not hard-coded into the instruction; instead, the instruction specifies a memory location that holds the required address. This addressing mode is used to handle *pointers*. For example, consider the high-level command `x=foo[j]`, where `foo` is an array variable and `x` and `j` are integer variables. What is the machine language equivalent of this command? Well, when the array `foo` is declared and initialized in the high-level program, the compiler allocates a memory segment to hold the array data and makes the symbol `foo` refer to the *base address* of that segment.

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

```
// Translation of x=foo[j] or x=*(foo+j):
ADD R1,foo,j    // R1←foo+j
LOAD* R2,R1     // R2←Memory[R1]
STR R2,x        // x←R2
```

**Flow of Control** While programs normally execute in a linear fashion, one command after the other, they also include occasional branches to locations other than the next command. Branching serves several purposes including *repetition* (jump

**High-level**

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

**Low-level**

```
// Typical translation:
beginWhile:
  JNG R1,endWhile // If R1<0 goto endWhile
  // Translation of code segment 1 comes here
  JMP beginWhile  // Goto beginWhile
endWhile:
  // Translation of code segment 2 comes here
```

**Figure 4.1** High- and low-level branching logic. The syntax of *goto commands* varies from one language to another, but the basic idea is the same.

backward to the beginning of a loop), *conditional execution* (if a Boolean condition is false, jump forward to the location after the "if-then" clause), and *subroutine calling* (jump to the first command of some other code segment). In order to support these programming constructs, every machine language features the means to jump to selected locations in the program, both conditionally and unconditionally. In assembly languages, locations in the program can also be given symbolic names, using some syntax for specifying labels. Figure 4.1 illustrates a typical example.

*Unconditional jump* commands like `JMP beginWhile` specify only the address of the target location. *Conditional jump* commands like `JNG R1,endWhile` must also specify a Boolean condition, expressed in some way. In some languages the condition is an explicit part of the command, while in others it is a by-product of executing a previous command.

This ends our informal introduction to machine languages and the generic operations that they normally support. The next section gives a formal description of one specific machine language—the native code of the computer that we will build in chapter 5.

## 4.2 Hack Machine Language Specification

### 4.2.1 Overview

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

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

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

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

First, the A register can be used to facilitate direct access to the data memory (which, from now on, will be often referred to as "memory"). Since Hack instructions are 16-bit wide, and since addresses are specified using 15 bits, it is impossible to pack both an operation code and an address in one instruction. Thus, the syntax of the Hack language mandates that memory access instructions operate on an implicit memory location labeled "`M`", for example, `D=M+1`. In order to resolve this address, the convention is that `M` always refers to the memory word whose address is the current value of the A register. For example, if we want to effect the operation D = Memory[516] − 1, we have to use one instruction to set the A register to 516, and a subsequent instruction to specify `D=M-1`.

In addition, the hardworking A register is also used to facilitate direct access to the instruction memory. Similar to the memory access convention, Hack jump instructions do not specify a particular address. Instead, the convention is that any jump operation always effects a jump to the instruction located in the memory word addressed by A. Thus, if we want to effect the operation *goto* 35, we use one instruction to set A to 35, and a second instruction to code a *goto* command, without specifying an address. This sequence causes the computer to fetch the instruction located in InstructionMemory[35] in the next clock cycle.

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

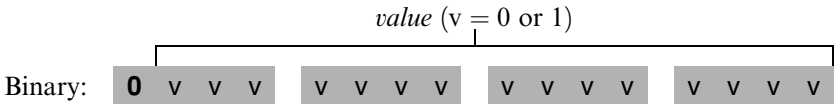
And now to the example: Suppose we want to add the integers 1 to 100, using repetitive addition. Figure 4.2 gives a C language solution and a possible compilation into the Hack language.

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

### 4.2.2 The *A*-Instruction

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

*A*-instruction: *@value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.




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

The *A*-instruction is used for three different purposes. First, it provides the only way to enter a constant into the computer under program control. Second, it sets the stage for a subsequent *C*-instruction designed to manipulate a certain data memory location, by first setting A to the address of that location. Third, it sets the stage for a subsequent *C*-instruction that specifies a jump, by first loading the address of the jump destination to the A register. These uses are demonstrated in figure 4.2.

**C language**

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

**Hack machine language**

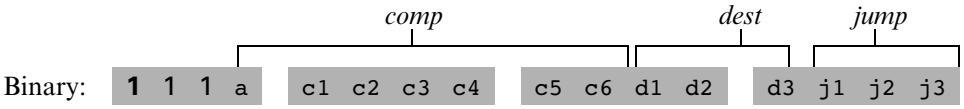
```
// Adds 1+...+100.
       @i     // i refers to some mem. location.
       M=1    // i=1
       @sum   // sum refers to some mem. location.
       M=0    // sum=0
 (LOOP)
       @i
       D=M    // D=i
       @100
       D=D-A  // D=i-100
       @END
       D;JGT  // If (i-100)>0 goto END
      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

**Figure 4.2** C and assembly versions of the same program. The infinite loop at the program's end is our standard way to "terminate" the execution of Hack programs.

### 4.2.3 The *C*-Instruction

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

*C*-instruction: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is omitted;
// If *jump* is empty, the ";" is omitted.




The leftmost bit is the *C*-instruction code, which is 1. The next two bits are not used. The remaining bits form three fields that correspond to the three parts of the instruction's symbolic representation. The overall semantics of the symbolic instruction *dest = comp;jump* is as follows. The *comp* field instructs the ALU what to compute. The *dest* field instructs where to store the computed value (ALU output). The *jump* field specifies a jump condition, namely, which command to fetch and execute next. We now describe the format and semantics of each of the three fields.

**The Computation Specification** The Hack ALU is designed to compute a fixed set of functions on the D, A, and M registers (where M stands for Memory[A]). The computed function is specified by the `a`-bit and the six `c`-bits comprising the instruction's *comp* field. This 7-bit pattern can potentially code 128 different functions, of which only the 28 listed in figure 4.3 are documented in the language specification.

Recall that the format of the *C*-instruction is `111a cccc ccdd djjj`. Suppose we want to have the ALU compute `D-1`, the current value of the D register minus 1. According to figure 4.3, this can be done by issuing the instruction `111`**`0 0011 10`**`00 0000` (the 7-bit operation code is in bold). To compute the value of `D|M`, we issue the instruction `111`**`1 0101 01`**`00 0000`. To compute the constant −1, we issue the instruction `111`**`0 1110 10`**`00 0000`, and so on.

**The Destination Specification** The value computed by the *comp* part of the *C*-instruction can be stored in several destinations, as specified by the instruction's 3-bit

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

**Figure 4.3** The *compute* field of the *C*-instruction. D and A are names of registers. M refers to the memory location addressed by A, namely, to Memory[A]. The symbols + and − denote 16-bit 2's complement addition and subtraction, while !, |, and & denote the 16-bit bit-wise Boolean operators Not, Or, and And, respectively. Note the similarity between this instruction set and the ALU specification given in figure 2.6.

*dest* part (see figure 4.4). The first and second d-bits code whether to store the computed value in the A register and in the D register, respectively. The third d-bit codes whether to store the computed value in M (i.e., in Memory[A]). One, more than one, or none of these bits may be asserted.

Recall that the format of the *C*-instruction is `111a cccc ccdd djjj`. Suppose we want the computer to increment the value of Memory[7] by 1 and to also store the result in the D register. According to figures 4.3 and 4.4, this can be accomplished by the following instructions:

```
0000 0000 0000 0111    // @7
1111 1101 1101 1000    // MD=M+1
```

| d1 | d2 | d3 | *Mnemonic* | *Destination (where to store the computed value)* |
|---|---|---|---|---|
| 0 | 0 | 0 | null | The value is not stored anywhere |
| 0 | 0 | 1 | M | Memory[A] (memory register addressed by A) |
| 0 | 1 | 0 | D | D register |
| 0 | 1 | 1 | MD | Memory[A] and D register |
| 1 | 0 | 0 | A | A register |
| 1 | 0 | 1 | AM | A register and Memory[A] |
| 1 | 1 | 0 | AD | A register and D register |
| 1 | 1 | 1 | AMD | A register, Memory[A], and D register |

**Figure 4.4** The *dest* field of the *C*-instruction.

The first instruction causes the computer to select the memory register whose address is 7 (the so-called M register). The second instruction computes the value of M + 1 and stores the result in both M and D.

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

Whether or not a jump should actually materialize depends on the three `j`-bits of the *jump* field and on the ALU output value (computed according to the *comp* field). The first `j`-bit specifies whether to jump in case this value is negative, the second `j`-bit in case the value is zero, and the third `j`-bit in case it is positive. This gives eight possible jump conditions, shown in figure 4.5.

The following example illustrates the jump commands in action:

*Logic*

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

*Implementation*

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

| j1 ($out < 0$) | j2 ($out = 0$) | j3 ($out > 0$) | Mnemonic | Effect |
|---|---|---|---|---|
| 0 | 0 | 0 | null | No jump |
| 0 | 0 | 1 | JGT | If $out > 0$ jump |
| 0 | 1 | 0 | JEQ | If $out = 0$ jump |
| 0 | 1 | 1 | JGE | If $out \geq 0$ jump |
| 1 | 0 | 0 | JLT | If $out < 0$ jump |
| 1 | 0 | 1 | JNE | If $out \neq 0$ jump |
| 1 | 1 | 0 | JLE | If $out \leq 0$ jump |
| 1 | 1 | 1 | JMP | Jump |

**Figure 4.5** The *jump* field of the *C*-instruction. *Out* refers to the ALU output (resulting from the instruction's *comp* part), and *jump* implies "continue execution with the instruction addressed by the A register."

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

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

### 4.2.4 Symbols

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

- *Predefined symbols:* A special subset of RAM addresses can be referred to by any assembly program using the following predefined symbols:
- *Virtual registers:* To simplify assembly programming, the symbols `R0` to `R15` are predefined to refer to RAM addresses 0 to 15, respectively.
- *Predefined pointers:* The symbols `SP`, `LCL`, `ARG`, `THIS`, and `THAT` are predefined to refer to RAM addresses 0 to 4, respectively. Note that each of these memory

locations has two labels. For example, address 2 can be referred to using either `R2` or `ARG`. This syntactic convention will come to play in the implementation of the virtual machine, discussed in chapters 7 and 8.

- *I/O pointers:* The symbols `SCREEN` and `KBD` are predefined to refer to RAM addresses 16384 (0x4000) and 24576 (0x6000), respectively, which are the base addresses of the screen and keyboard memory maps. The use of these I/O devices is explained later.

- *Label symbols:* These user-defined symbols, which serve to label destinations of *goto* commands, are declared by the pseudo-command "`(Xxx)`". This directive defines the symbol `Xxx` to refer to the instruction memory location holding the next command in the program. A label can be defined only once and can be used anywhere in the assembly program, even before the line in which it is defined.

- *Variable symbols:* Any user-defined symbol `Xxx` appearing in an assembly program that is not predefined and is not defined elsewhere using the "`(Xxx)`" command is treated as a *variable*, and is assigned a unique memory address by the assembler, starting at RAM address 16 (0x0010).

### 4.2.5 Input/Output Handling

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

**Screen** The Hack computer includes a black-and-white screen organized as 256 rows of 512 pixels per row. The screen's contents are represented by an 8K memory map that starts at RAM address 16384 (0x4000). Each row in the physical screen, starting at the screen's top left corner, is represented in the RAM by 32 consecutive 16-bit words. Thus the pixel at row $r$ from the top and column $c$ from the left is mapped on the $c\%16$ bit (counting from LSB to MSB) of the word located at $\mathrm{RAM}[16384 + r \cdot 32 + c/16]$. To write or read a pixel of the physical screen, one reads or writes the corresponding bit in the RAM-resident memory map (1 = black, 0 = white). Example:

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

**Keyboard** The Hack computer interfaces with the physical keyboard via a single-word memory map located in RAM address 24576 (0x6000). Whenever a key is pressed on the physical keyboard, its 16-bit ASCII code appears in RAM[24576]. When no key is pressed, the code 0 appears in this location. In addition to the usual ASCII codes, the Hack keyboard recognizes the keys shown in figure 4.6.

### 4.2.6 Syntax Conventions and File Format

**Binary Code Files** A binary code file is composed of text lines. Each line is a sequence of sixteen "0" and "1" ASCII characters, coding a single machine language instruction. Taken together, all the lines in the file represent a machine language program. The contract is such that when a machine language program is loaded into the computer's instruction memory, the binary code represented by the file's *n*th line is stored in address *n* of the instruction memory (the count of both program lines and memory addresses starts at 0). By convention, machine language programs are stored in text files with a "`hack`" extension, for example, `Prog.hack`.

**Assembly Language Files** By convention, assembly language programs are stored in text files with an "`asm`" extension, for example, `Prog.asm`. An assembly language

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

**Figure 4.6** Special keyboard codes in the Hack platform.

file is composed of text lines, each representing either an *instruction* or a *symbol declaration:*

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

(The remaining conventions in this section pertain to assembly programs only.)

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

**Comments** Text beginning with two slashes (`//`) and ending at the end of the line is considered a comment and is ignored.

**White Space** Space characters are ignored. Empty lines are ignored.

**Case Conventions** All the assembly mnemonics must be written in uppercase. The rest (user-defined labels and variable names) is case sensitive. The convention is to use uppercase for labels and lowercase for variable names.

## 4.3 Perspective

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

In terms of syntax, we have chosen to give Hack a somewhat different look-and-feel than the mechanical nature of most assembly languages. In particular, we have chosen a high-level language-like syntax for the *C*-command, for example, `D=M` and `D=D+M` instead of the more traditional `LOAD` and `ADD` directives. The reader

should note, however, that these are just syntactic details. For example, the + character plays no algebraic role whatsoever in the command `D=D+M`. 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 appear in a single command. In this respect, Hack may be described as a "$\frac{1}{2}$ address machine": Since there is no room to pack both an instruction code and a 15-bit address in the 16-bit instruction format, operations involving memory access will normally be specified in Hack using two instructions: an *A*-instruction to specify the address and a *C*-instruction to specify the operation. In comparison, most machine languages can directly specify at least one address in every machine instruction.

Indeed, Hack assembly code typically ends up being (mostly) an alternating sequence of *A*- and *C*-instructions, for example, `@xxx` followed by `D=D+M`, `@YYY` followed by `0;JMP`, and so on. If you find this coding style tedious or even peculiar, you should note that friendlier *macro commands* like `D=D+M[xxx]` and `GOTO YYY` can easily be introduced into the language, causing Hack assembly code to be more readable as well as about 50 percent shorter. The trick is to have the assembler translate these macro commands into binary code effecting `@xxx` followed by `D=D+M`, `@YYY` followed by `0;JMP`, and so on.

The *assembler*, mentioned several times in this chapter, is the program responsible for translating symbolic assembly programs into executable programs written in binary code. In addition, the assembler is responsible for managing all the system- and user-defined symbols found in the assembly program, and for replacing them with physical memory addresses, as needed. We return to this translation task in chapter 6, in which we build an assembler for the Hack language.

## 4.4 Project

**Objective** Get a taste of low-level programming in machine language, and get acquainted with the Hack computer platform. In the process of working on this project, you will also become familiar with the assembly process, and you will appreciate visually how the translated binary code executes on the target hardware.

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

**Contract** Write and test the two programs described in what follows. When executed on the CPU emulator, your programs should generate the results mandated by the test scripts supplied in the project directory.

- *Multiplication Program* (`Mult.asm`): The inputs of this program are the current values stored in `R0` and `R1` (i.e., the two top RAM locations). The program computes the product `R0*R1` and stores the result in `R2`. We assume (in this program) that `R0>=0`, `R1>=0`, and `R0*R1<32768`. Your program need not test these conditions, but rather assume that they hold. The supplied `Mult.tst` and `Mult.cmp` scripts will test your program on several representative data values.
- *I/O-Handling Program* (`Fill.asm`): This program runs an infinite loop that listens to the keyboard input. When a key is pressed (any key), the program blackens the screen, namely, writes "black" in every pixel. When no key is pressed, the screen should be cleared. You may choose to blacken and clear the screen in any spatial order, as long as pressing a key continuously for long enough will result in a fully blackened screen and not pressing any key for long enough will result in a cleared screen. This program has a test script (`Fill.tst`) but no compare file—it should be checked by visibly inspecting the simulated screen.

**Steps** We recommend proceeding as follows:

0. The assembler and CPU emulator programs needed for this project are available in the `tools` directory of the book's software suite. Before using them, go through the assembler tutorial and the CPU emulator tutorial.

1. Use a plain text editor to write the first program in assembly, and save it as `projects/04/mult/Mult.asm`.

2. Use the supplied assembler (in either batch or interactive mode) to translate your program. If you get syntax errors, go to step 1. If there are no syntax errors, the assembler will produce a file called `projects/04/mult/Mult.hack`, containing binary machine instructions.

3. Use the supplied CPU emulator to test the resulting `Mult.hack` code. This can be done either interactively, or batch-style using the supplied `Mult.tst` script. If you get run-time errors, go to step 1.

4. Repeat stages 1–3 for the second program (`Fill.asm`), using the `projects/04/fill` directory.

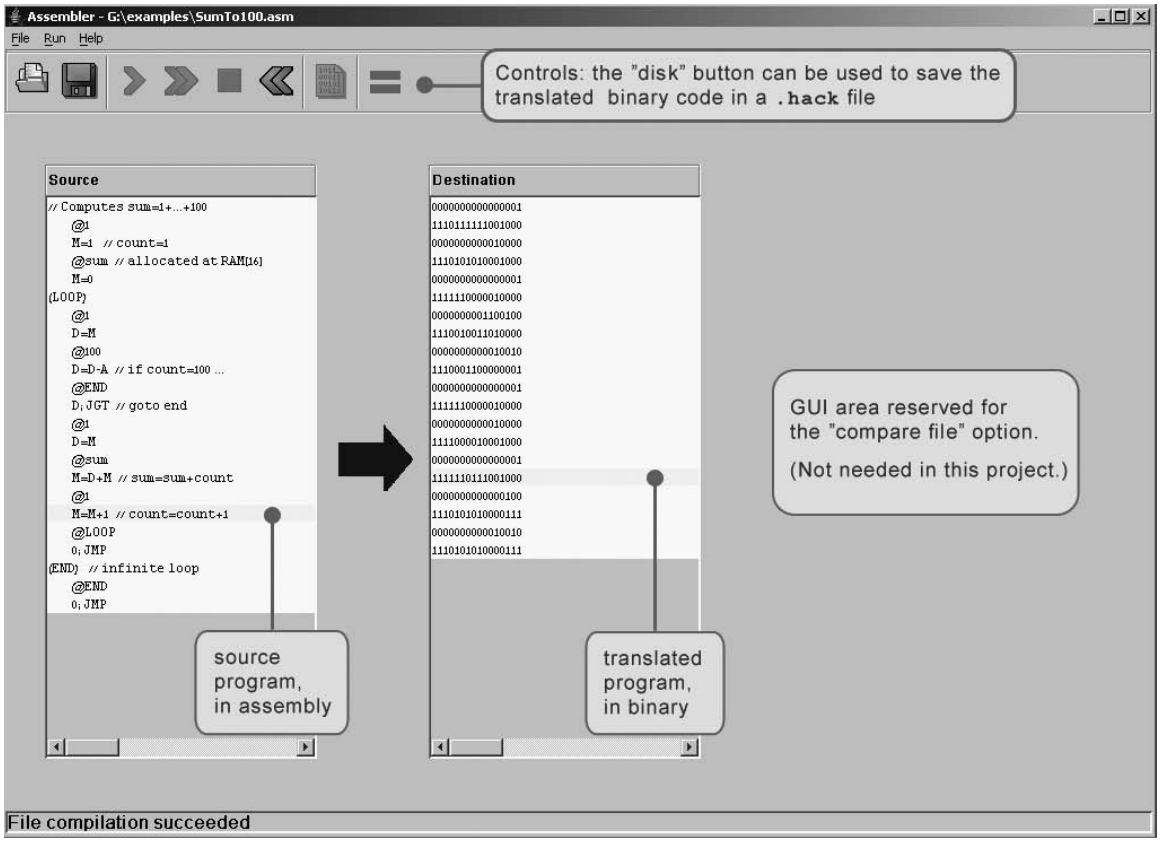



**Figure 4.7** The visual assembler supplied with the book.

**Debugging Tip** The Hack language is case sensitive. A common error occurs when one writes, say, `@foo` and `@Foo` in different parts of the program, thinking that both commands refer to the same variable. In fact, the assembler treats these symbols as two completely different identifiers.

**The Supplied Assembler** The book's software suite includes a Hack assembler that can be used in either command mode or GUI mode. The latter mode of operation allows observing the translation process in a visual and step-wise fashion, as shown in figure 4.7.

The machine language programs produced by the assembler can be tested in two different ways. First, one can run the `.hack` program in the CPU emulator.

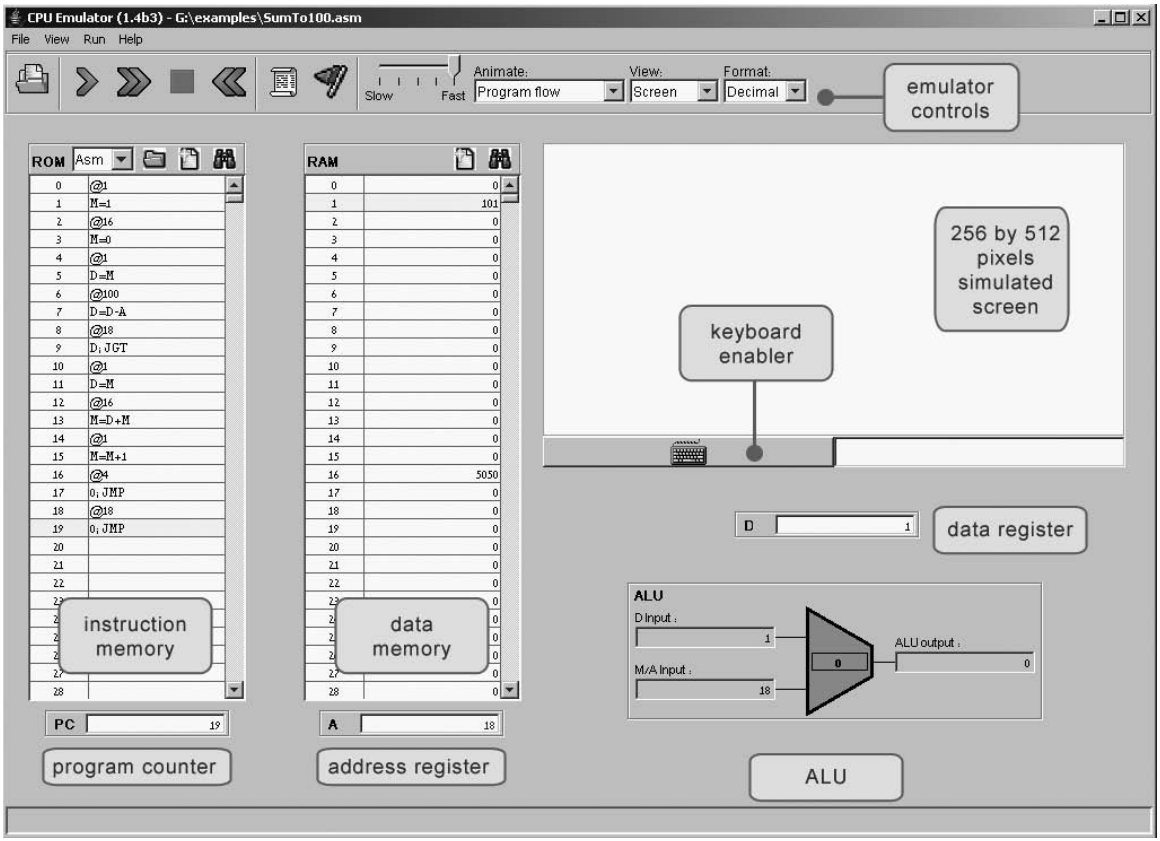



**Figure 4.8** The CPU emulator supplied with the book. The loaded program can be displayed either in symbolic notation (as shown in this screen shot) or in binary code. The screen and the keyboard are not used by this particular program.

Alternatively, one can run the same program directly on the hardware, by loading it into the computer's instruction memory using the hardware simulator. Since we will finish building the hardware platform only in the next chapter, the former option makes more sense at this stage.

**The Supplied CPU Emulator** This program simulates the Hack computer platform. It allows loading a Hack program into the simulated ROM and visually observing its execution on the simulated hardware, as shown in figure 4.8.

For ease of use, the CPU emulator enables loading binary `.hack` files as well as symbolic `.asm` files. In the latter case, the emulator translates the assembly program into binary code on the fly. This utility seems to render the supplied assembler unnecessary, but this is not the case. First, the supplied assembler shows the translation process visually, for instructive purposes. Second, the binary files generated by the assembler can be executed directly on the hardware platform. To do so, load the Computer chip (built in chapter 5's project) into the hardware simulator, then load the `.hack` file generated by the assembler into the computer's ROM chip.

# 5 Computer Architecture[1]

This chapter is the pinnacle of the hardware part of our journey. We are now ready to take all the chips that we've 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 we will build, called *Hack*, has two important virtues. On the one hand, Hack is a simple machine that can be constructed in just a few hours, using previously built chips and the hardware simulator supplied with the book. 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 an excellent understanding of how modern computers work at the low hardware and software levels.

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 systems built in project 3. Section 5.4 compares the Hack machine with industrial-strength computers, and emphasizes the critical role that optimization plays in the latter. Section 5.5 gives an overview of the computer construction project.

The computer that will emerge from this project will be as simple as possible, but not simpler. On the one hand, the computer will be based on a minimal and compact hardware configuration. On the other hand, this configuration will be sufficiently powerful for executing programs written in a Java-like programming language, delivering a reasonable performance and a satisfying user experience.

---

[1] Chapter 5 from *The Elements of Computing Systems* by Noam Nisan and Shimon Schocken, Second Edition, MIT Press, forthcoming 2017.

## 5.1 Computer Architecture Fundamentals

### 5.1.1 The Stored Program Concept

Compared to all the other machines around us, the most unique feature of the digital computer is its amazing versatility. Here is a machine with finite hardware that can perform an infinite number of tasks, from playing games to publishing books to designing airplanes. This remarkable versatility—a boon that we have come to take for granted—is the fruit of a brilliant 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 remarkably simple. The computer is based on a fixed hardware platform, capable of executing a fixed repertoire of very 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 it was in mechanical computers predating 1930. Instead, the program's code is temporarily stored and manipulated in the computer's memory, *just 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 a key element of many abstract and practical computer models, most notably the *universal 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 computational

problems and solutions. In contrast, the von Neumann machine is a practical architecture and the conceptual blueprint of almost all computer platforms today.

The von Neumann architecture, shown in diagram 5.1, is based on a *central processing unit* (CPU), interacting with a *memory* device, receiving data from some *input* device, and sending 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 very instructions that tell the computer what to do. Let us explore this architecture in some detail.

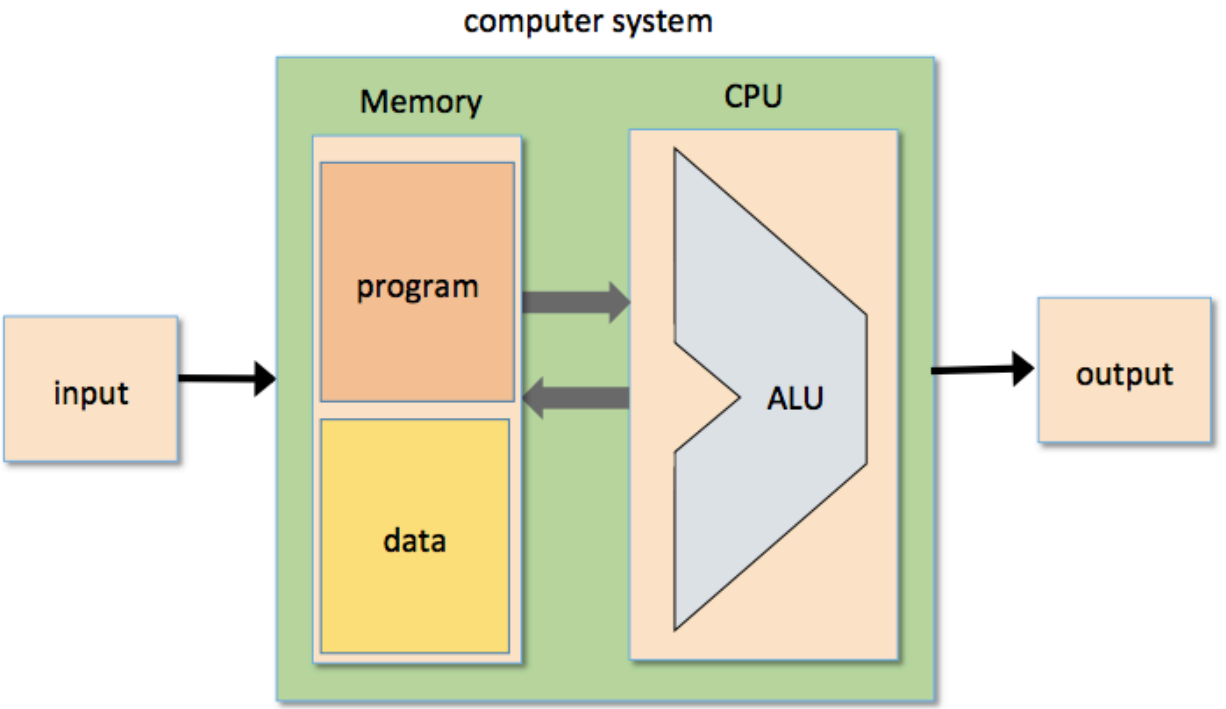



Diagram 5.1: the von Neumann architecture

### 5.1.3 Memory

Like other hardware elements, the memory unit can be discussed from a physical perspective and from a logical perspective. Physically, the memory is a linear sequence of addressable *registers*, each having a unique address and a value, which is a fixed-size word of information. Logically, the memory is divided into two areas. One area is dedicated for storing data, e.g. the arrays and objects of programs that are presently executing, while the other area is dedicated for storing the programs' instructions. Although all these "data words" and "instruction words" look exactly the same physically, they serve very different purposes.

In some von Neumann architecture variants, the data memory and the instruction memory are managed within the same physical memory unit, as was just explained. In other variants, the data memory and the instruction memory are kept in separate physical memory units that have distinct address spaces. This setting, sometimes referred to as "Harvard architecture", is also the architecture of our Hack computer. Both variants have certain advantages that will be discussed later in the chapter.

All the memory registers—irrespective of their roles—are accessed in the same way: in order to manipulate a particular memory register, one must first select the register by supplying an address. This action provides an immediate access to the register's data. The term *Random Access Memory* (RAM) is often used to denote the important fact that each randomly selected register can be reached in the same access time, irrespective of the memory size and the register's location in it.

**Data Memory**: High-level programs manipulate abstract artifacts like variables, arrays, and objects. After the programs are translated into machine language, these data abstractions become binary codes, stored in the computer's memory. Once an individual register has been

selected from the memory by specifying its address, its contents can be either *read* or *written* to. In the former case, we retrieve the value of the selected register. In the latter case, we store a new value in the selected register, overriding the previous value. Such memories are sometimes referred to as "read/write" memories.

**Instruction Memory**: Before high-level programs can be executed on the computer, they must be translated into machine language. Following this translation, each high-level statement becomes a series of one or more machine language instructions. These instructions are stored in the computer's *instruction memory* as binary codes. In each step of a program's execution, the CPU *fetches* (i.e., *reads*) a binary machine instruction from a selected register in the instruction memory, decodes it, executes the specified instruction, and figures out which instruction to fetch and execute next.

We see that before executing a particular program, we must first load the program's code into the instruction memory, typically from some peripheral mass storage device like a disk. Given the compact and highly focused perspective of a von Neumann machine, *how* a program is loaded into the computer's instruction 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 memory, one way or another. As you saw in chapter 4, the act of loading a program into the instruction memory from an external text file is supported by the supplied CPU emulator.

### 5.1.4 Central Processing Unit

The CPU—the centerpiece of the computer's architecture—is in charge of executing the instructions of the currently loaded program. These instructions tell the CPU which calculation it has to perform, which registers is has to read from or write to, and which instruction it has to fetch and execute next. The CPU executes these tasks using three main hardware 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. For example, a typical ALU can add two numbers, compute a bitwise And function on two numbers, compare two numbers, and so on. How much functionality an ALU should have is a matter of need, budget, energy, and similar cost-effectiveness considerations. Any function not supported by the ALU as a primitive hardware operation can be later realized by the computer's system software (yielding a slower implementation, of course).

**Registers**: Since the CPU is the computer's centerpiece, it must perform as efficiently as possible. In order to boost performance, it is desirable to store the intermediate results that computer programs generate locally, close to the ALU, rather than ship them in and out of the CPU chip and store them in some remote and separate RAM chip. Thus, a CPU is typically equipped with a small set of 2 up to 32 resident high-speed *registers*, each capable of holding a single word.

**Control Unit**: A computer instruction is represented as a binary code, typically 16, 32, or 64 bits wide. Before such an instruction can be executed, it must be decoded, and the information embedded in it must be used to signal various hardware devices (ALU, registers, memory) how to execute the instruction. The instruction decoding is done by some *control unit*, which is also responsible for figuring out which instruction to fetch and execute next.

The CPU operation can now be described as a repeated loop: decode the current instruction, execute it, figure out which instruction to execute next, fetch it, decode it, and so on. This process is sometimes referred to as the "fetch-execute cycle".

### 5.1.5 Registers

When talking about computer hardware, the term "register" is used quite liberally to refer to any device capable of storing a chunk of bits that represents some stand-alone value like a variable

value, an instruction, or an address. According to this broad definition, any memory location is in fact a register, and so of course are the registers that reside inside the CPU. This section is dedicated to a discussion of these CPU-resident registers.

Suppose we didn't have any CPU-resident registers. This would imply that any CPU operation that requires inputs or outputs would have to rely on memory access. Let us consider what each such memory access entails. First, some address value travels from the CPU to the RAM's address input. Next, the RAM's direct-access logic uses the supplied address to select a specific memory register. Finally, the register's contents either travels back to the CPU (a read operation), or is replaced by some additional value that travels from the CPU (a write operation).

Note that this elaborate process involves at least two separate chips, an address bus, and a data bus, resulting in an expensive and time-consuming operation. This stands in sharp contract to the ALU, which is a lean and mean combinational machine. Thus we have a lightning fast calculator that depends on a sluggish data store for supplying inputs and consuming outputs. The result may well lead to what is sometimes called *starvation*, which is what happens when a processor is denied the resources it needs to complete its work.

Clearly, if we could have placed a few high-speed registers inside the CPU itself, right next to the ALU, we could have saved ourselves a great deal of time and overhead. There is another, subtle but critically important advantage for using CPU-resident registers. In order to specify an instruction that includes a memory register, like `Memory[`*addr*`]=`*value*, we must supply a memory address, which typically requires many bits. In the 16-bit Hack platform, this technical detail alone forces us to use *two* machine instructions, and *two* clock cycles, even for preforming mundane tasks like `Memory[`*addr*`]=0` or `Memory[`*addr*`]=1`.

In contrast, since there are normally only a few CPU-resident registers, identifying each one of them requires only a few bits. Therefore, an operation like *someCPURegister*`=0` or

*someCPURegister*=1 requires only one machine instruction, and one cycle. To sum up, CPU-resident registers save unnecessary memory access, and allow using thinner instruction formats, resulting in faster throughput. The remainder of this section describes the registers that CPU's typically use.

***Data registers***: These registers give the CPU short-term memory services. For example, if a program wants to calculate $(a - b) \cdot c$, we must first compute and remember the value of $(a - b)$. In principle, this temporary result can be stored in some memory register. Clearly, a much more sensible solution is to store it locally inside the CPU, using a *data register*. Typically, CPU's use at least one and up to 32 data registers.

***Address registers****:* Many machine language instructions involve memory access: reading data, writing data, and fetching instructions. In any one of these operations, we must specify which memory register we wish to operate on. This is done by supplying an address. In some cases, the address is coded as part of the instruction, while in other cases the address is specified, or computed, by some previous instruction. In the latter case, the address must be stored somewhere. This is done using a CPU-resident chip called *address register*.

Unlike regular registers, the output of an address register is typically connected to the address input of a memory device. Therefore, placing a value in the address register has the side effect of selecting a particular memory register, and this register makes itself available to subsequent instructions designed to manipulate it. For example, suppose we wish to set `Memory[17]` to `1`. In the Hack language, this can be done using the pair of instructions `@17` (which sets `A=17` and makes the `M` mnemonic stand for `Memory[17]`), followed by `M=1` (which sets the selected memory register to 1).

In addition to supporting this fundamental addressing operation, an address register is, well, a register. Therefore, if needed, it can be used as yet another data register. For example,

suppose we wish to set the `D` register to 17. This can be done using the pair of instructions `@17`, followed by `D=A`. Here we use `A` not as an address register, but rather as a data register. The fact that `Memory[17]` was selected as a side effect of `@17` is completely ignored.

***Program counter***: When executing a program, the CPU must always keep track of the address of the instruction that must be fetched and executed next. This address is kept in a special register called *program counter*, or PC. The contents of the PC is computed and updated as a side effect of executing the current instruction, as we elaborate later in the chapter.

### 5.1.6 Input and Output

Computers interact with their external environments using a diverse array of input and output (I/O) devices. These include screens, keyboards, disks, printers, scanners, network interface cards, and so on, not to mention the bewildering array of proprietary components that embedded computers are called to control in automobiles, cameras, medical devices, and so on. There are two reasons why we don't concern ourselves here with the low-level architecture of these various devices. First, every one of them represents a unique piece of machinery requiring a unique knowledge of engineering. Second, and for this very same reason, computer scientists have devised clever schemes to make all these different devices look exactly the same to the computer. The key trick in managing this complexity is called *memory-mapped* I/O.

The basic idea is to create a binary emulation of the I/O device, making it "look" to the CPU as if it were a regular memory segment. In particular, each I/O device is allocated an exclusive area in memory, becoming 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 certain pixel turns on or off on the screen. The I/O devices are "refreshed" from the memory (and vice versa) several times per second, so the response time from the user's perspective is almost instantaneous. Programmatically, the key implication is that computer programs can access any I/O device by simply manipulating selected registers in their designated memory areas.

Obviously, this arrangement is based on several agreed-upon contracts. First, the data that drives each I/O device must be serialized, or "mapped", on the computer's memory, hence the name "memory map". For example, the screen, which can be viewed as a 2-dimensional grid of pixels, must be mapped on a 1-dimensional vector of fixed-size memory registers. Second, each I/O device is required to support some agreed-upon interaction protocol, so that programs will be able to access it in a predictable manner. For example, it should be decided, and agreed-upon, which binary codes should represent which keys on the keyboard. As a side comment, given the multitude of computer platforms, I/O devices, and different hardware and software vendors, one can appreciate the crucial role that *standards* play in determining these low-level interaction contracts.

The practical implications of a memory-mapped I/O architecture are significant: The design of the CPU and the overall platform can be totally independent of the number, nature, or make of the I/O devices that interact, or will interact, with the computer. 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 its base address (these one-time configurations are typically done by the operating system). From this point onward, any program that wants to manipulate this I/O device can do so—all it needs to do is manipulate selected registers in the memory map designated to represent the device.

The architectural framework described thus far in the chapter is characteristic of any general-purpose computer system. We now turn to describe one specific example of this architecture: the Hack computer.

## 5.2 The Hack Hardware Platform: Specification

### 5.2.1 Overview

The Hack platform is a 16-bit von Neumann machine, designed to execute programs written in the Hack machine language presented in chapter 4. 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, this memory can be implemented using a ROM chip that is pre-loaded with the required program. Software-based simulators of the Hack computer are expected to 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 called *data register* (`D`), *address register* (`A`), and *program counter* (`PC`), identical to the 16-bit registers built in project 3. While the `D`-register is used solely for storing data values, the `A`-register serves three different purposes, depending on the context in which it is used: storing a data value (just like the `D`-register), pointing at an address in the instruction memory, or pointing at an address in the data memory. More about this, later.

The Hack CPU is designed to execute instructions written in the Hack machine language. These instructions have the 16-bit format "`ixxaccccccdddjjj`". The `i`-bit (also known as *opcode*) codes the instruction *type*, which is either `0` for an *A*-instruction or `1` for a *C*-instruction. In case of an *A*-instruction, the instruction is treated as a 16-bit binary value which is loaded

into the A register. In case of a *C*-instruction, the instruction is treated as a sequence of control bits that determine which function the ALU should computer, and in which registers the computed value should be stored. In the course of executing any one of these instructions, the CPU also figures out which instruction in the program should be fetched and executed next.

We now turn to specify the various components of the Hack hardware platform.

### 5.2.2 Central Processing Unit (CPU)

The CPU of the Hack platform is designed to execute 16-bit instructions according to the Hack machine language specification presented in chapter 4. The Hack CPU expects to be connected to two separate memory modules: an instruction memory, from which it fetches instructions for execution, and a data memory, from which it can read, and into which it can write, data values. Diagram 5.2 gives the complete CPU specification.

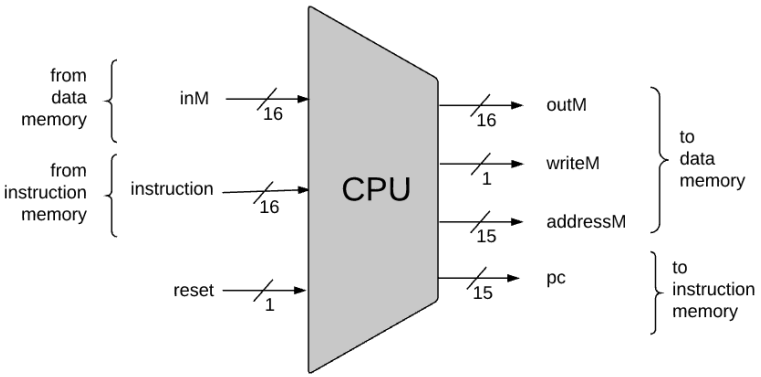
### 5.2.3 Instruction Memory

The Hack *instruction memory* is implemented in a direct-access *Read-Only Memory* device, also called ROM. The Hack ROM consists of 32K addressable 16-bit registers, as shown in diagram 5.3.

### 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 specify.

In order to facilitate interaction with a user, the Hack platform can be connected to two peripheral devices: a *screen* and a *keyboard*. Both devices interact with the computer platform through *memory-mapped* buffers. Specifically, screen images can be drawn and probed by

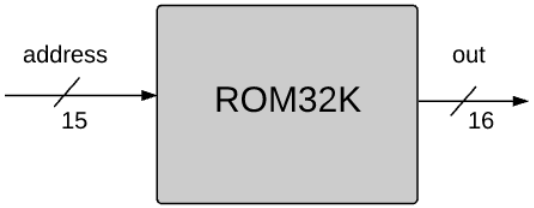



```
/** The Hack Central Processing Unit consists of an ALU, two registers named A and D, and a program
counter named PC (these internal chip-parts are not shown in the diagram). The inM
input and outM output hold the values referred to as "M" in the Hack instruction syntax. The addressM
output holds the memory address to which outM should be written.

The CPU is designed to fetch and execute instructions written in the Hack machine language. If
instruction is an A-instruction, the CPU loads the 16-bit constant that the instruction represents 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} registers
specified by the instruction. If one of these registers is M, the CPU asserts the writeM control bit output
(when writeM is 0, any value may appear in outM).

When the reset input is 0, the CPU uses the ALU output and the jump directive specified by the
instruction to compute the address of the next instruction, and emits this address to the pc output. If the
reset input is 1, the CPU sets pc to 0.

The outM and writeM outputs are combinational, and 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. */

CHIP CPU

IN
   instruction[16], // Instruction to execute.
   inM[16],         // Value of Mem[A], the instruction's M input
   reset;           // Signals whether to continue executing the current program
                    // (reset==1) or restart the current program (reset==0).

OUT
   outM[16],      // Value to write to Mem[addressM], the instruction's M output
   addressM[15],  // In which address to write?
   writeM,        // Write to the Memory?
   pc[15];        // address of next instruction
```

Diagram 5.2: The Hack Central Processing Unit (CPU) interface

writing and reading, respectively, 16-bit values in a designated memory segment called *screen memory map*. Similarly, which key is presently pressed on the keyboard can be determined by probing a designated memory register called *keyboard memory map*.

The memory maps interact with their respective I/O devices via peripheral logic that resides outside the computer. The contract is as follows: When a bit is changed in the screen's memory map, a respective black and white pixel is drawn on the physical screen. When a key is pressed on the physical keyboard, the respective scan-code of this key appears in the keyboard's memory map.




/** The instruction memory of the Hack computer, implemented as a read-only memory of 32K registers, each 16-bit wide.
Performs the operation `out = ROM32K[address]`.
In words: outputs the 16-bit value stored in the register selected by the `address` input. This value is taken to be the *current instruction*.
It is assumed that the chip is preloaded with a program written in the Hack machine language.
Software-based simulators of the Hack computer are expected to provide means for loading the chip with a Hack program, either interactively, or using a test script. */

```
CHIP ROM32K

IN  address[15];

OUT out[16];
```

**Diagram 5.3: The Hack Instruction Memory interface.**

We now turn to specifying the built-in chips that interface between the hardware platform and the I/O devices. This will set the stage for specifying the complete memory module that embeds these chips.

**Screen**: The Hack computer can interact with a physical screen consisting of 256 rows of 512 black-and-white pixels each. The computer interfaces with the physical screen via a memory map, implemented by a `RAM` chip called `Screen`. This chip behaves like regular memory, meaning that it can be read and written to. In addition, it features the side effect that any bit written to it is reflected as a pixel on the physical screen (`1` = black, `0` = white). The exact mapping between the memory map and the physical screen is specified in diagram 5.4.

/** The `Screen` (memory map) functions exactly like a 16-bit, 8K RAM:

(1) `out(t)` = `Screen[address(t)](t)`

(2) `if load(t) then Screen[address(t)](t+1)` = `in(t)`

The chip implementation has the side effect of continuously refreshing a physical screen. The physical screen consists of 256 rows and 512 columns of black and white pixels (simulators of the Hack computer are expected to simulate this screen).

Each row in the physical screen, starting at the top left corner, is represented in the `Screen` memory map by 32 consecutive 16-bit words. Thus the pixel at row *r* from the top and column *c* from the left ($0 \leq r \leq 255$, $0 \leq c \leq 511$) is mapped on the c%16 bit (counting from LSB to MSB) of the 16-bit word stored in `Screen[`*r* * 32 + *c* / 16`]`. */

```
CHIP Screen

IN
    in[16],      // what to write
    address[13]; // where to write (or read)
    load,        // write-enable bit

OUT
    out[16];     // Screen value at the given address
```

Diagram 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 chip called `Keyboard`. When a key is pressed on the physical keyboard, a unique 16-bit scan-code is emitted to the `output` of the `Keyboard` chip. When no key is pressed, the chip outputs `0`. The character set of the Hack platform is given in Appendix C, along with the scan-code of each character.

/** The `Keyboard` (memory map) is connected to a standard, physical keyboard. It is made to output the 16-bit scan-code associated with the key which is presently pressed on the physical keyboard, or `0` if no key is pressed.

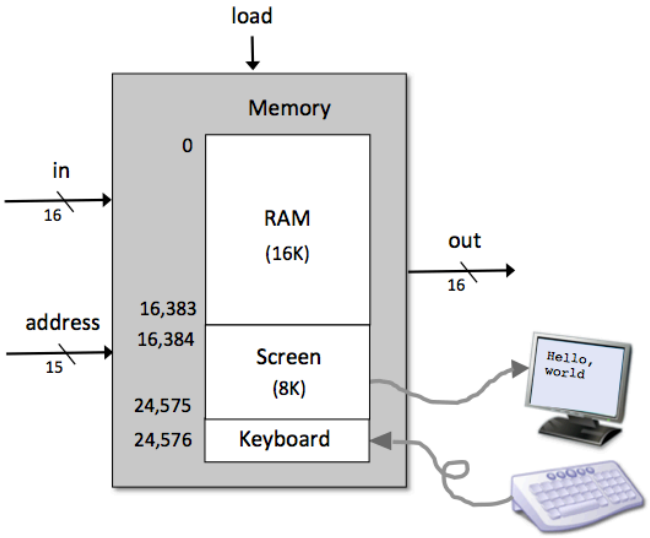
The keyboard scan-codes are given in Appendix C of the book.
Simulators of the Hack computer are expected to implement the contract described above. */

```
CHIP Keyboard

OUT out[16];   // The scan-code of the pressed key,
               // or 0 if no key is currently pressed.
```

Diagram 5.5: The Hack `Keyboard` chip interface

### 5.2.6 Data Memory

The overall address space known as the Hack *data memory* is realized by a chip called `Memory`. This chip is essentially a package of three 16-bit storage devices: a `RAM` (16K registers, for regular data storage), a `Screen` (8K registers, acting as the screen memory map), and a `Keyboard` (1 register, acting as the keyboard memory map). The complete specification is given in diagram 5.6.




/** The complete address space of the Hack computer's data memory, including RAM and memory-mapped I/O. Facilitates read and write operations, as follows:

Read: `out(t) = Mem[address(t)](t)`

Write: `if load(t) then Mem[address(t)](t+1) = in(t)`

In words: the chip always outputs the value stored at the memory location specified by `address`.

If `load==1`, the `in` value is loaded into the register specified by `address`. This value becomes available through the `out` output from the next time step onward.

The memory access rules are as follows:

Only the top 16K+8K+1 words of the address space are used.

`0x0000-0x5FFF`: accessing an address in this range results in accessing the RAM.

`0x4000-0x5FFF`: accessing an address in this range results in accessing the `Screen`.

`0x6000`: accessing this address results in accessing the `Keyboard`.

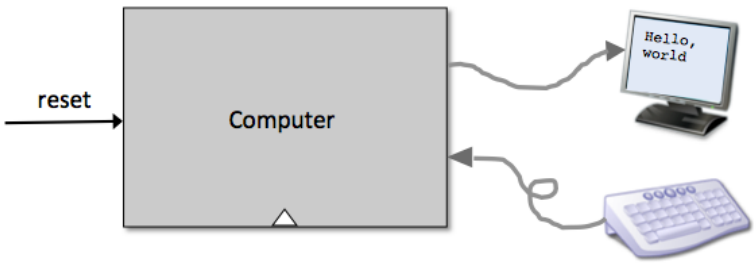
`> 0x6000`: accessing an address in this range is invalid. */

```
CHIP Memory
IN   in[16], load, address[15];
OUT  out[16];
```

Diagram 5.6: The Hack Data Memory interface

### 5.2.7 Computer

The topmost chip in the Hack hardware hierarchy is a `Computer` chip consisting of a CPU, an instruction memory, and a data memory. The computer can interact with a screen and a keyboard. The complete specification is given in diagram 5.7.




```
/** The HACK computer, consisting of CPU, ROM and Memory parts
(these internal chip-parts are not shown in the diagram).
When reset==0, the program stored in the computer's ROM executes.
When reset==1, the execution of the program restarts.
Thus, to start a program's execution, the reset input must be pushed "up"
(signaling 1) and "down" (signaling 0).
From this point onward, the user is at the mercy of the software. In
particular, depending on the program's code, the screen may show some
output, and the user may be able to interact with the computer via the
keyboard. */

CHIP Computer

IN   reset;
```

Diagram 5.7: Interface of the topmost chip in the Hack hardware platform, named `Computer`.

## 5.3 Implementation

This section outlines how a hardware platform can be built to realize the Hack computer specification described 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 hardware simulator supplied with the book. As usual, technical details are given in the final Project section of this chapter.

### 5.3.1 The Central Processing Unit

When we set out to implement the Hack `CPU`, our objective is to come up with 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, the proposed `CPU` implementation includes an `ALU` chip capable of computing arithmetic/logical functions, a set of registers, a program counter, and some additional gates designed to help decode, execute, and fetch instructions. Since all these building blocks were already built in previous chapters, the key question that we face now is how to arrange and connect them in a way that effects the desired `CPU` operation. One possible configuration is illustrated in diagram 5.8.

The architecture shown in diagram 5.8 is used to perform three classical CPU tasks: decoding the current instruction, executing the current instruction, and deciding which instruction to fetch and execute next. We now turn to describe these three tasks.

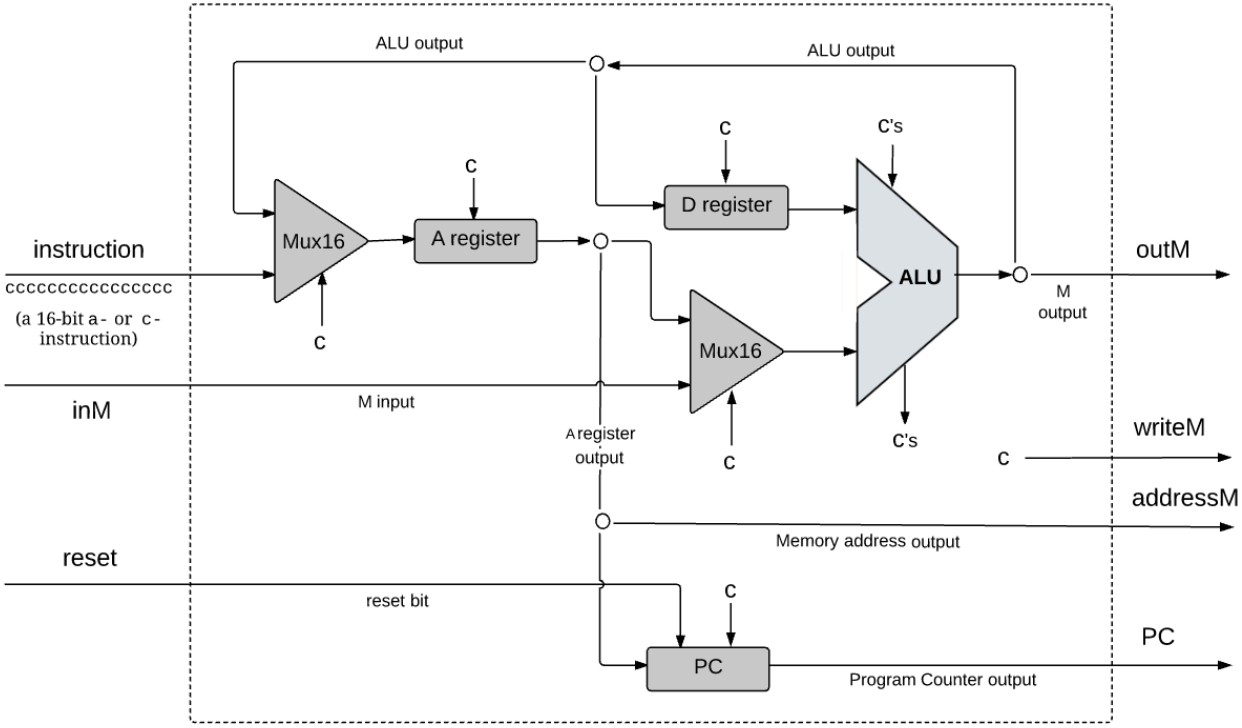



Diagram 5.8: Proposed `CPU` implementation, showing an "incoming" 16-bit instruction denoted `cccccccccccccccc`. The instruction bits are labeled `c`, since in the case of a *C*-instruction, the `CPU` logic treats them as control bits that are extracted from the instruction and routed to different chip-parts of the `CPU`. In particular, 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 6 control bits that instruct the `ALU` what to compute, and the "`c`'s" output stands for its `zr` and `ng` outputs). Taken together, the distributed behaviors that these control bits effect throughout the `CPU` architecture end up executing the instruction.

**Instruction decoding**

The 16-bit value of the `CPU`'s `instruction` input represents either an *A*-instruction or a *C*-instruction. In order to figure out the semantics of this instruction, we can parse, or unpack it, into the following fields: "`ixxaccccccdddjjj`". The `i`-bit (also known as *opcode*) codes the instruction *type*, which is either `0` for an *A*-instruction or `1` for a *C*-instruction. In case of an *A*-instruction, the entire instruction represent the 16-bit value of the constant that should be loaded into the `A` register. In case of a *C*-instruction, the `a-` and `c`-bits code the *comp* part of the instruction, while the `d-` and `j`-bits code the *dest* and *jump* parts of the instruction, respectively (the `x-`bits are not used, and can be ignored).

**Instruction Execution**

The decoded fields of the instruction (`i-`, `a-`, `c-`, `d-`, and `j`-bits) are routed simultaneously to various parts of the `CPU` architecture, where they cause different chip-parts to do what they are supposed to do in order to execute either the *A*- or the *C*-instruction, as mandated by the Hack machine language specification. In the case of a *C*-instruction, the single `a`-bit determines whether the `ALU` will operate on the `A` register input or on the `M` input, and the six `c`-bits determine which function the `ALU` will compute. The three `d`-bits are used to determine which registers should "accept" the `ALU` resulting output, and the three `j`-bits are used to for branching control, as we now turn to describe.

**Instruction Fetching**

As a side effect of executing the current instruction, the `CPU` must determine, and emit, the address of the instruction that should be fetched and executed next. The key element in this sub-task is the *Program Counter*—a `CPU` chip-part whose role is to always store the address of the next instruction. Later in the chapter we'll describe how we connect the `pc` output of the `CPU` into the `address` input of the instruction memory; this connection causes the instruction

memory to always emit the instruction that ought to be fetched and executed next. This output is connected to the `instruction` input of the `CPU`, closing the fetch-execute cycle.

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 reset the Program Counter to 0. That's why in diagram 5.8 the `reset` input of the `CPU` is fed directly into the `reset` input of the `PC` chip. If we'll 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, the default operation of the Program Counter is `PC++`.

But what if the instruction dictates to effect a "jump *n*" operation, where *n* is the address of an instruction located anywhere in the program? According to the Hack language specification, a "jump *n*" operation is realized using a sequence of two instructions. First, we issue the *A*-instruction `@n`, which sets the `A` register to *n*; next, we issue a *C*-instruction that includes a jump directive. According to the language specification, execution always branches to the instruction that the `A` register points at. Thus, when implementing the `CPU`, one of our challenges is to come up with a logic gate architecture that realizes the following behavior: if *jump* then `PC` = `A` else `PC++`. The value of the Boolean expression *jump* depends on the instruction's `j`-bits and on the `ALU` output.

How to implement this logic? The answer is hinted by diagram 5.8. Note that the output of the `A` register feeds into the input of the `PC` register. Recall that the latter chip has a load-bit that enables it to accept a new input value. Thus, if we'll assert this load-bit, we'll cause the architecture to effect the operation `PC=A` rather than the default operation `PC++`. We should do this only if we have to realize a jump. Now, the question of weather or not a jump should be realized is answered by two signals: (i) the `j`-bits of the current instruction, specifying the jump

condition, and (ii) the ALU output-bits zr and ng, which can be used to determine if the specified condition is satisfied, or not.

We'll stop here, lest we rob readers the pleasure of discovering the missing details and completing the CPU implementation on their own.

### 5.3.2 Memory

The Hack Memory chip is essentially an amalgamation of three lower-level chips: RAM16K, Screen, and Keyboard. Yet this modularity is strictly implicit: users of the Memory chip, like Hack programmers or programmers who write compilers that generate Hack code, see a *single address space*, spanning from address 0 to address 24576 (0×0000 to 0×6000 in hexa).

The implementation of the Memory chip (as shown in diagram 5.6) should realize this continuum effect. For example, if the address input of the Memory chip happens to be 16384, the chip logic should end up accessing address 0 in the Screen chip, and so on. This can be done using similar techniques to those used in chapter 3 to integrate small RAM units into larger ones.

### 5.3.3 Computer

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

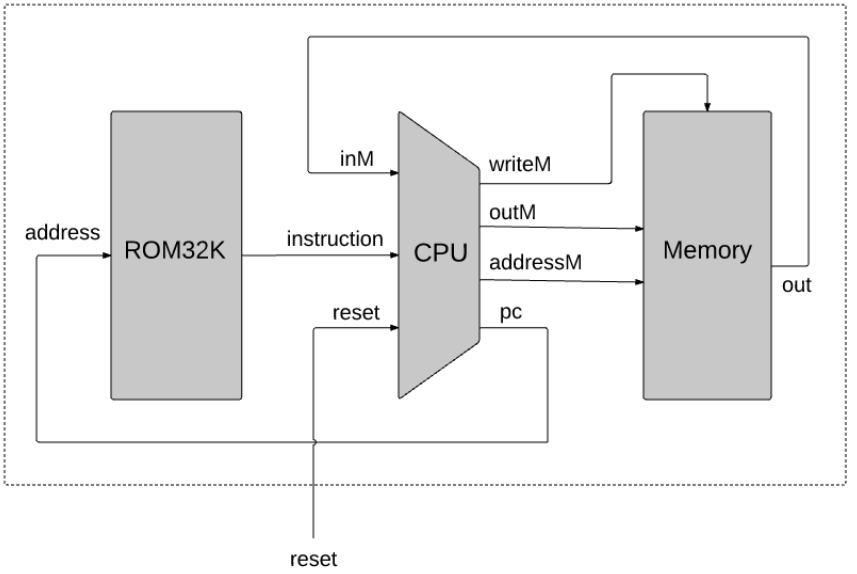



Diagram 5.9: Proposed implementation of the platform's topmost chip, `Computer`.

## 5.4 Project

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

**Resources**: The only resources that you need for completing this project are the hardware simulator supplied with the book and the test materials described here. The computer platform should be written in HDL and tested using the hardware simulator.

**Contract**: Build a hardware platform capable of executing programs written in the Hack machine language specified in chapter 4. Demonstrate the platform's operations by having your `Computer` chip run the three programs described below.

**Test Programs**: A natural way to test the overall `Computer` chip implementation is to have it execute some 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 its `ROM` chip, and then runs the clock enough cycles to execute the program. We supply the following test programs (as well as all the relevant 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.

Before testing your `Computer` chip on any one of the above programs, read the test script associated with the program and be sure to understand the instructions given to the simulator. If needed, consult Appendix B.

**Steps**: Implement the hardware platform in the following order:

*Memory*: Composed from three chips: `RAM16K`, `Screen`, and `Keyboard`. The `Screen` and the `Keyboard` are available as built-in chips and there is no need to build them. Although the `RAM16K` chip was built in project 3, we recommend using its built-in version, as it provides a debugging-friendly GUI.

*CPU:* The central processing unit can be built according to the proposed implementation given in figure 5.8, using the `ALU` and `Register` chips built in chapters 2 and 3, respectively. We recommend using the built-in versions of these chips, in particular `ARegister` and `DRegister`. These chips have exactly the same functionality of the `Register` chip specified in chapter 3, plus GUI side effects.

In the course of implementing the `CPU`, you may be tempted to specify and build some internal 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 diagram 5.8, plus some elementary logic gates built in project 1.

*Instruction Memory*: Use the built-in `ROM32K` chip.

*Computer*: Build the topmost `Computer` chip using the three chip-parts shown in diagram 5.9.

**The 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 screen shot of testing the `Rect.hack` program on a `Computer` chip implementation.

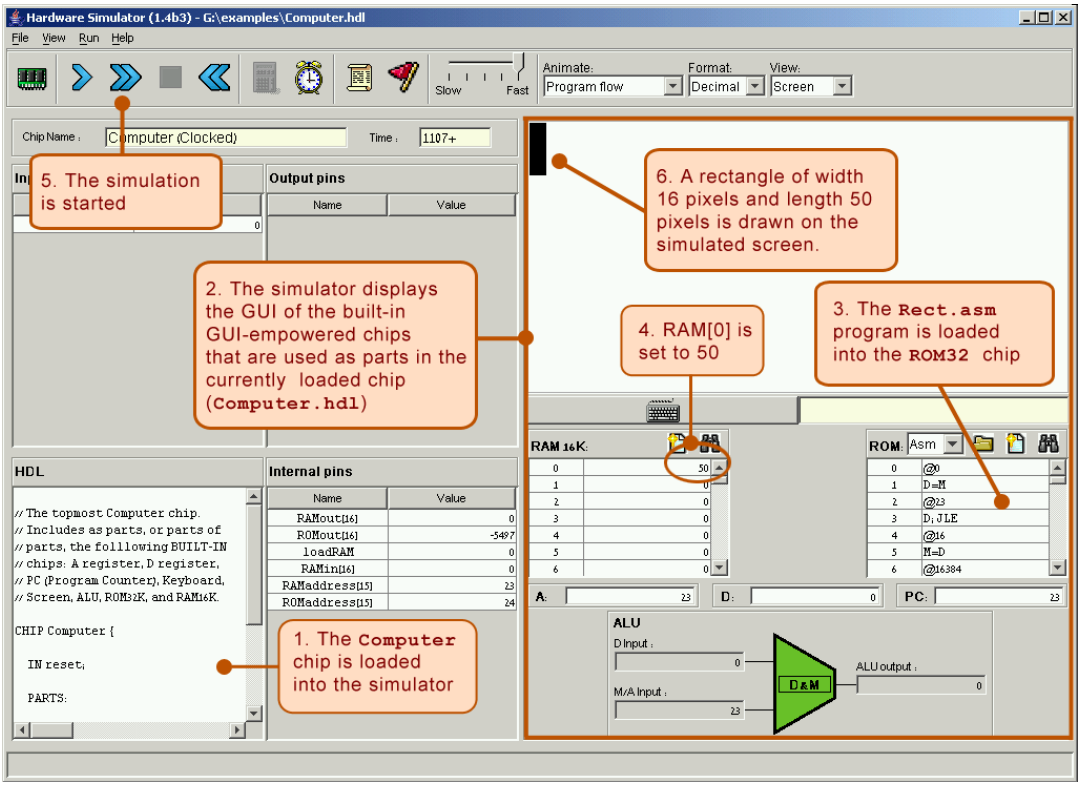



Diagram 5.10: Testing the `Computer` chip on the supplied hardware simulator. The `Rect` program draws a rectangle of `RAM[0]` rows of 16 pixels each, all black, at the top-left of the screen. Note that the program is error-free. Therefore, if it does not operate as expected, it means that the hardware platform on which it is running (`Computer.hdl` and/or, possibly, some of it's chip-parts) is buggy.

## 5.5 Perspective

Following the general spirit of the book, the architecture of the Hack computer is rather minimal. Typical computer platforms have more registers, more data types, more powerful ALUs, and richer instruction sets. However, these differences are mainly quantitative. From a qualitative standpoint, Hack is quite similar to most digital computers, as they all follow the same conceptual paradigm: the von Neumann architecture.

In terms of function, computer systems can be classified into two categories: *general-purpose computers*, designed to easily switch from executing one program to another, and *dedicated computers*, usually embedded in other systems like cell phones, game consoles, digital cameras, weapon systems, factory equipment, and so on. For any particular application, a single program is burned into the dedicated computer's ROM, and is the only one that can be executed (in game consoles, for example, the game software resides in an external cartridge that is simply a replaceable ROM module encased in some fancy package). Aside from this difference, general-purpose and dedicated computers share the same architectural ideas: stored programs, fetch-decode-execute logic, CPU, registers, program counter, and so on.

Unlike Hack, most general-purpose computers use a single address space for storing both data and instructions. In such architectures, the instruction address as well as the optional data address specified by the instruction must be fed into the same destination: the single address input of the shared address space. Clearly, this cannot be done at the same time. The standard solution is to base the computer implementation 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 optional data address inferred from it is fed to the memory's address input, allowing the instruction to manipulate the selected memory location. In contrast, the Hack architecture is unique in that it partitions the address

space into two separate parts, allowing a single-cycle fetch-execute logic. The price of this simpler hardware design is that programs cannot be changed dynamically.

In terms of I/O, the Hack keyboard and screen are rather spartan. General-purpose computers are typically connected to multiple I/O devices like printers, disks, network connections, and so on. Also, typical screens are obviously much more powerful than the Hack screen, featuring more pixels, many brightness levels in each pixel, and colors. Still, the basic principle that each pixel is controlled by a memory-resident binary value is maintained: instead of a single bit controlling the pixel's black or white color, several bits are devoted to control the level of brightness of each of the three primary colors that, together, produce the pixel's ultimate color. Likewise, the memory mapping of the Hack screen is simplistic. Instead of mapping pixels directly into bits of memory, most modern computers allow the CPU to send high-level graphic instructions to a *graphics card* that controls the screen. This way, the CPU is relieved from the tedium of drawing figures like circles and polygons directly—the graphics card takes care of this task using its own embedded chip-set.

Finally, it should be stressed that most of the effort and creativity in designing computer hardware is invested in achieving better performance. Thus, hardware architecture courses and textbooks typically evolve around such issues as implementing memory hierarchies (cache), better access to I/O devices, pipelining, parallelism, instruction prefetching, and other optimization techniques that were sidestepped in this chapter.

Historically, attempts to enhance the processor's performance have led to two main schools of hardware design. Advocates of the *Complex Instruction Set Computing* (CISC) approach argue for achieving better performance by providing rich and elaborate instruction sets. Conversely, the *Reduced Instruction Set Computing* (RISC) camp uses simpler instruction sets in order to promote as fast a hardware implementation as possible. 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, from *Romeo and Juliet*

The first half of the book (chapters 1–5) described and built a computer's *hardware platform*. The second half of the book (chapters 6–12) focuses on the computer's *software hierarchy*, culminating in the development of a compiler and a basic operating system for a simple, object-based programming language. The first and most basic module in this software hierarchy is the *assembler*. In particular, chapter 4 presented machine languages in both their *assembly* and *binary* representations. This chapter describes how assemblers can systematically translate programs written in the former into programs written in the latter. As the chapter unfolds, we explain how to develop a *Hack assembler*—a program that generates binary code that can run as is on the hardware platform built in chapter 5.

Since the relationship between symbolic assembly commands and their corresponding binary codes is straightforward, writing an assembler (using some high-level 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 user-defined symbols and resolve them to physical memory addresses. This task is normally done using a *symbol table*—a classical data structure that comes to play in many software translation projects.

As usual, the Hack assembler is not an end in itself. Rather, it provides a simple and concise demonstration of the key software engineering principles used in the construction of any assembler. Further, writing the assembler is the first in the series of seven software development projects that accompany the rest of the book. Unlike the hardware projects, which were implemented in HDL, the software projects that construct the translator programs (*assembler*, *virtual machine*, and *compiler*) may be implemented in any programming language. In each project, we provide a language-neutral API and a detailed step-by-step test plan, along with all the necessary test

programs and test scripts. Each one of these projects, beginning with the assembler, is a stand-alone module that can be developed and tested in isolation from all the other projects.

## 6.1 Background

Machine languages are typically specified in two flavors: *symbolic* and *binary*. The binary codes—for example, `1100001010000001100000000000000111`—represent actual machine instructions, as understood by the underlying hardware. For example, the instruction's leftmost 8 bits can represent an operation code, say `LOAD`, the next 8 bits a register, say `R3`, and the remaining 16 bits an address, say `7`. Depending on the hardware's logic design and the agreed-upon machine language, the overall 32-bit pattern can thus cause the hardware to effect the operation "load the contents of Memory[7] into register R3." Modern computer platforms support dozens if not hundreds of such elementary operations. Thus, machine languages can be rather complex, involving many operation codes, different memory addressing modes, and various instruction formats.

One way to cope with this complexity is to document machine instructions using an agreed-upon syntax, say `LOAD R3,7` rather than `110000101000000110 000000000000111`. And since the translation from symbolic notation to binary code is straightforward, it makes sense to allow low-level programs to be written in symbolic notation and to 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 command into its underlying fields, translates each field into its equivalent binary code, and assembles the generated codes into a binary instruction that can be actually executed by the hardware.

**Symbols** Binary instructions are represented in binary code. By definition, they refer to memory addresses using actual numbers. For example, consider a program that uses a variable to represent the *weight* of various things, and suppose that this variable has been mapped on location 7 in the computer's memory. At the binary code level, instructions that manipulate the *weight* variable must refer to it using the explicit address 7. Yet once we step up to the assembly level, we can allow writing commands like `LOAD R3,weight` instead of `LOAD R3,7`. In both cases, the command will effect the same operation: "set R3 to the contents of Memory[7]." In a similar fashion, rather than using commands like `goto 250`, assembly languages allow commands like `goto loop`, assuming that somewhere in the program the symbol `loop` is

made to refer to address 250. In general then, symbols are introduced into assembly programs from two sources:

- *Variables:* The programmer can use symbolic variable names, and the translator will "automatically" assign them to memory addresses. Note that the actual values of these addresses are insignificant, so long as each symbol is resolved to the same address throughout the program's translation.
- *Labels:* The programmer can mark various locations in the program with symbols. For example, one can declare the label `loop` to refer to the beginning of a certain code segment. Other commands in the program can then `goto loop`, either conditionally or unconditionally.

The introduction of symbols into assembly languages suggests that assemblers must be more sophisticated than dumb text processing programs. Granted, translating agreed-upon symbols into agreed-upon binary codes is not a complicated task. At the same time, the mapping of user-defined variable names and symbolic labels on actual memory addresses is not trivial. In fact, this symbol resolution task is the first nontrivial translation challenge in our ascent up the software hierarchy from the hardware level. The following example illustrates the challenge and the common way to address it.

**Symbol Resolution** Consider figure 6.1, showing a program written in some self-explanatory low-level language. The program contains four user-defined symbols: two variable names (`i` and `sum`) and two labels (`loop` and `end`). How can we systematically convert this program into a symbol-less code?

We start by making two arbitrary game rules: The translated code will be stored in the computer's memory starting at address 0, and variables will be allocated to memory locations starting at address 1024 (these rules depend on the specific target hardware platform). Next, we build a *symbol table*, as follows. For each new symbol *xxx* encountered in the source code, we add a line $(xxx, n)$ to the symbol table, where $n$ is the memory address associated with the symbol according to the game rules. After completing the construction of the symbol table, we use it to translate the program into its symbol-less version.

Note that according to the assumed game rules, variables `i` and `sum` are allocated to addresses 1024 and 1025, respectively. Of course any other two addresses will be just as good, so long as *all* references to `i` and `sum` in the program resolve to the same physical addresses, as indeed is the case. The remaining code is self-explanatory, except perhaps for instruction 6. This instruction terminates the program's execution by putting the computer in an infinite loop.

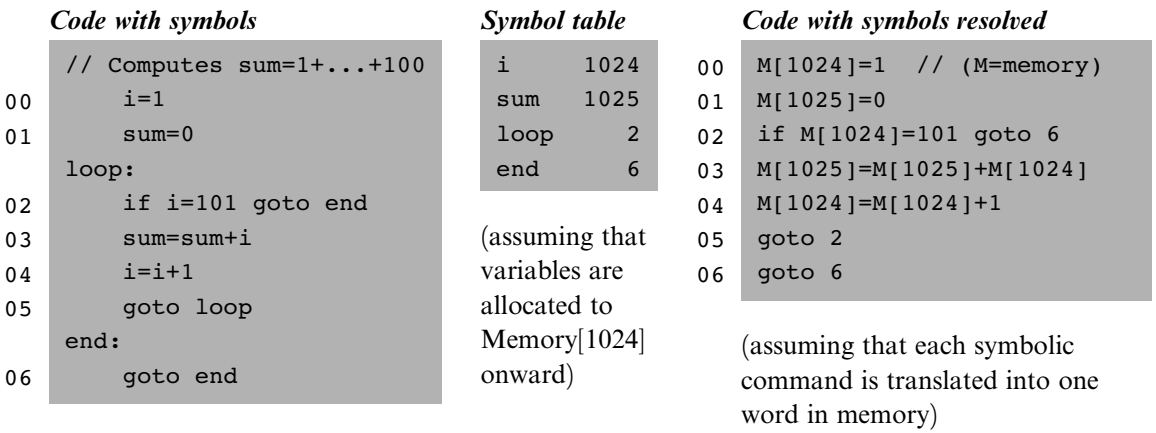



**Figure 6.1** Symbol resolution using a symbol table. The line numbers are not part of the program—they simply count all the lines in the program that represent real instructions, namely, neither comments nor label declarations. Note that once we have the symbol table in place, the symbol resolution task is straightforward.

Three comments are in order here. First, note that the variable allocation assumption implies that the largest program that we can run is 1,024 instructions long. Since realistic programs (like the operating system) are obviously much larger, the base address for storing variables will normally be much farther. Second, the assumption that each source command is mapped on one word may be naïve. Typically, some assembly commands (e.g., `if i=101 goto end`) may translate into several machine instructions and thus will end up occupying several memory locations. The translator can deal with this variance by keeping track of how many words each source command generates, then updating its ''instruction memory counter'' accordingly.

Finally, the assumption that each variable is represented by a single memory location is also naïve. Programming languages feature variables of different types, and these occupy different memory spaces on the target computer. For example, the C language data types `short` and `double` represent 16-bit and 64-bit numbers, respectively. When a C program is run on a 16-bit machine, these variables will occupy a single memory address and a block of four consecutive addresses, respectively. Thus, when allocating memory space for variables, the translator must take into account both their data types and the word width of the target hardware.

**The Assembler** Before an assembly program can be executed on a computer, it must be translated into the computer's binary machine language. The translation task is

done by a program called the assembler. The assembler takes as input a stream of assembly commands and generates as output a stream of equivalent binary instructions. The resulting code can be loaded as is into the computer's memory and executed by the hardware.

We see that the assembler is essentially a text-processing program, designed to provide translation services. The programmer who is commissioned to write the assembler must be given the full documentation of the assembly syntax, on the one hand, and the respective binary codes, on the other. Following this contract—typically called *machine language specification*—it is not difficult to write a program that, for each symbolic command, carries out the following tasks (not necessarily in that order):

- Parse the symbolic command into its underlying fields.
- For each field, generate the corresponding bits in the machine language.
- Replace all symbolic references (if any) with numeric addresses of memory locations.
- Assemble the binary codes into a complete machine instruction.

Three of the above tasks (parsing, code generation, and final assembly) are rather easy to implement. The fourth task—symbols handling—is more challenging, and considered one of the main functions of the assembler. This function was described in the previous section. The next two sections specify the Hack assembly language and propose an assembler implementation for it, respectively.

## 6.2 Hack Assembly-to-Binary Translation Specification

The Hack assembly language and its equivalent binary representation were specified in chapter 4. A compact and formal version of this language specification is repeated here, for ease of reference. This specification can be viewed as the contract that Hack assemblers must implement, one way or another.

### 6.2.1 Syntax Conventions and File Formats

**File Names** By convention, programs in binary machine code and in assembly code are stored in text files with "`hack`" and "`asm`" extensions, respectively. Thus, a `Prog.asm` file is translated by the assembler into a `Prog.hack` file.

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

**Assembly Language (`.asm`) Files** An assembly language file is composed of text lines, each representing either an *instruction* or a *symbol declaration*:

- *Instruction*: an *A*-instruction or a *C*-instruction, described in section 6.2.2.
- `(Symbol)`: This pseudo-command binds the `Symbol` to the memory location into which the next command in the program will be stored. It is called "pseudo-command" since it generates no machine code.

(The remaining conventions in this section pertain to assembly programs only.)

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

**Comments** Text beginning with two slashes (`//`) and ending at the end of the line is considered a comment and is ignored.

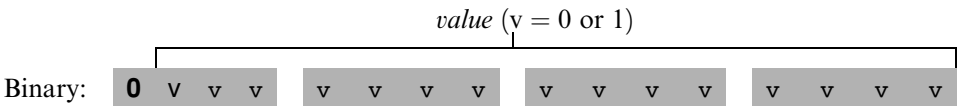
**White Space** Space characters are ignored. Empty lines are ignored.

**Case Conventions** All the assembly mnemonics must be written in uppercase. The rest (user-defined labels and variable names) is case sensitive. The convention is to use uppercase for labels and lowercase for variable names.

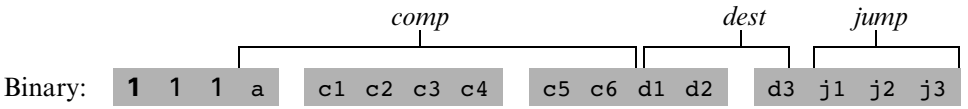
### 6.2.2 Instructions

The Hack machine language consists of two instruction types called *addressing instruction* (*A*-instruction) and *compute instruction* (*C*-instruction). The instruction format is as follows.

*A*-instruction: *@value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.




*C*-instruction: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is omitted;
// If *jump* is empty, the ";" is omitted.




The translation of each of the three fields *comp, dest, jump* to their binary forms is specified in the following three tables.

| ***comp*** (when a=0) | c1 | c2 | c3 | c4 | c5 | c6 | ***comp*** (when a=1) |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| -1 | 1 | 1 | 1 | 0 | 1 | 0 | |
| D | 0 | 0 | 1 | 1 | 0 | 0 | |
| A | 1 | 1 | 0 | 0 | 0 | 0 | M |
| !D | 0 | 0 | 1 | 1 | 0 | 1 | |
| !A | 1 | 1 | 0 | 0 | 0 | 1 | !M |
| -D | 0 | 0 | 1 | 1 | 1 | 1 | |
| -A | 1 | 1 | 0 | 0 | 1 | 1 | -M |
| D+1 | 0 | 1 | 1 | 1 | 1 | 1 | |
| A+1 | 1 | 1 | 0 | 1 | 1 | 1 | M+1 |
| D-1 | 0 | 0 | 1 | 1 | 1 | 0 | |
| A-1 | 1 | 1 | 0 | 0 | 1 | 0 | M-1 |
| D+A | 0 | 0 | 0 | 0 | 1 | 0 | D+M |
| D-A | 0 | 1 | 0 | 0 | 1 | 1 | D-M |
| A-D | 0 | 0 | 0 | 1 | 1 | 1 | M-D |
| D&A | 0 | 0 | 0 | 0 | 0 | 0 | D&M |
| D\|A | 0 | 1 | 0 | 1 | 0 | 1 | D\|M |

| *dest* | d1 | d2 | d3 |
|---|---|---|---|
| null | 0 | 0 | 0 |
| M | 0 | 0 | 1 |
| D | 0 | 1 | 0 |
| MD | 0 | 1 | 1 |
| A | 1 | 0 | 0 |
| AM | 1 | 0 | 1 |
| AD | 1 | 1 | 0 |
| AMD | 1 | 1 | 1 |

| *jump* | j1 | j2 | j3 |
|---|---|---|---|
| null | 0 | 0 | 0 |
| JGT | 0 | 0 | 1 |
| JEQ | 0 | 1 | 0 |
| JGE | 0 | 1 | 1 |
| JLT | 1 | 0 | 0 |
| JNE | 1 | 0 | 1 |
| JLE | 1 | 1 | 0 |
| JMP | 1 | 1 | 1 |

### 6.2.3 Symbols

Hack assembly commands can refer to memory locations (addresses) using either constants or symbols. Symbols in assembly programs arise from three sources.

**Predefined Symbols** Any Hack assembly program is allowed to use the following predefined symbols.

| *Label* | *RAM address* | *(hexa)* |
|---|---|---|
| SP | 0 | 0x0000 |
| LCL | 1 | 0x0001 |
| ARG | 2 | 0x0002 |
| THIS | 3 | 0x0003 |
| THAT | 4 | 0x0004 |
| R0–R15 | 0-15 | 0x0000-f |
| SCREEN | 16384 | 0x4000 |
| KBD | 24576 | 0x6000 |

Note that each one of the top five RAM locations can be referred to using two predefined symbols. For example, either `R2` or `ARG` can be used to refer to RAM[2].

**Label Symbols** The pseudo-command `(Xxx)` defines the symbol `Xxx` to refer to the instruction memory location holding the next command in the program. A label can be defined only once and can be used anywhere in the assembly program, even before the line in which it is defined.

**Variable Symbols** Any symbol `Xxx` appearing in an assembly program that is not predefined and is not defined elsewhere using the `(Xxx)` command is treated as a variable. Variables are mapped to consecutive memory locations as they are first encountered, starting at RAM address 16 (0x0010).

### 6.2.4 Example

Chapter 4 presented a program that sums up the integers 1 to 100. Figure 6.2 repeats this example, showing both its assembly and binary versions.

**Assembly code (`Prog.asm`)**

```
// Adds 1 + ... + 100
      @i
      M=1    // i=1
      @sum
      M=0    // sum=0
(LOOP)
      @i
      D=M    // D=i
      @100
      D=D-A  // D=i-100
      @END
      D;JGT  // if (i-100)>0 goto END
      @i
      D=M    // D=i
      @sum
      M=D+M  // sum=sum+i
      @i
      M=M+1  // i=i+1
      @LOOP
      0;JMP  // goto LOOP
(END)
      @END
      0;JMP  // infinite loop
```

Assembler

**Binary code (`Prog.hack`)**

```
(this line should be erased)
0000 0000 0001 0000
1110 1111 1100 1000
0000 0000 0001 0001
1110 1010 1000 1000
(this line should be erased)
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0110 0100
1110 0100 1101 0000
0000 0000 0001 0010
1110 0011 0000 0001
0000 0000 0001 0000
1111 1100 0001 0000
0000 0000 0001 0001
1111 0000 1000 1000
0000 0000 0001 0000
1111 1101 1100 1000
0000 0000 0000 0100
1110 1010 1000 0111
(this line should be erased)
0000 0000 0001 0010
1110 1010 1000 0111
```

**Figure 6.2** Assembly and binary representations of the same program.

## 6.3 Implementation

The Hack assembler reads as input a text file named `Prog.asm`, containing a Hack assembly program, and produces as output a text file named `Prog.hack`, containing the translated Hack machine code. The name of the input file is supplied to the assembler as a command line argument:

```
prompt> Assembler Prog.asm
```

The translation of each individual assembly command to its equivalent binary instruction is direct and one-to-one. Each command is translated separately. In particular, each mnemonic component (field) of the assembly command is translated into its corresponding bit code according to the tables in section 6.2.2, and each symbol in the command is resolved to its numeric address as specified in section 6.2.3.

We propose an assembler implementation based on four modules: a *Parser* module that parses the input, a *Code* module that provides the binary codes of all the assembly mnemonics, a *SymbolTable* module that handles symbols, and a main program that drives the entire translation process.

**A Note about API Notation** The assembler development is the first in a series of five software construction projects that build our hierarchy of translators (*assembler*, *virtual machine*, and *compiler*). Since readers can develop these projects in the programming language of their choice, we base our proposed implementation guidelines on language independent APIs. A typical project API describes several *modules*, each containing one or more *routines*. In object-oriented languages like Java, C++, and C#, a module usually corresponds to a *class*, and a routine usually corresponds to a *method*. In procedural languages, routines correspond to *functions*, *subroutines*, or *procedures*, and modules correspond to collections of routines that handle related data. In some languages (e.g., Modula-2) a module may be expressed explicitly, in others implicitly (e.g., a *file* in the C language), and in others (e.g., Pascal) it will have no corresponding language construct, and will just be a conceptual grouping of routines.

### 6.3.1 The *Parser* Module

The main function of the parser is to break each assembly command into its underlying components (fields and symbols). The API is as follows.

**Parser:** Encapsulates access to the input code. Reads an assembly language command, parses it, and provides convenient access to the command's components (fields and symbols). In addition, removes all white space and comments.

| Routine | Arguments | Returns | Function |
|---|---|---|---|
| Constructor/ initializer | Input file/ stream | — | Opens the input file/stream and gets ready to parse it. |
| `hasMoreCommands` | — | Boolean | Are there more commands in the input? |
| `advance` | — | — | Reads the next command from the input and makes it the current command. Should be called only if `hasMoreCommands()` is true. Initially there is no current command. |
| `commandType` | — | `A_COMMAND`, `C_COMMAND`, `L_COMMAND` | Returns the type of the current command:<br>■ `A_COMMAND` for `@Xxx` where `Xxx` is either a symbol or a decimal number<br>■ `C_COMMAND` for `dest=comp;jump`<br>■ `L_COMMAND` (actually, pseudo-command) for `(Xxx)` where `Xxx` is a symbol. |
| `symbol` | — | string | Returns the symbol or decimal `Xxx` of the current command `@Xxx` or `(Xxx)`. Should be called only when `commandType()` is `A_COMMAND` or `L_COMMAND`. |
| `dest` | — | string | Returns the `dest` mnemonic in the current *C*-command (8 possibilities). Should be called only when `commandType()` is `C_COMMAND`. |

| Routine | Arguments | Returns | Function |
|---|---|---|---|
| `comp` | — | string | Returns the `comp` mnemonic in the current *C*-command (28 possibilities). Should be called only when `commandType()` is `C_COMMAND`. |
| `jump` | — | string | Returns the `jump` mnemonic in the current *C*-command (8 possibilities). Should be called only when `commandType()` is `C_COMMAND`. |

### 6.3.2 The *Code* Module

**Code:** Translates Hack assembly language mnemonics into binary codes.

| Routine | Arguments | Returns | Function |
|---|---|---|---|
| `dest` | mnemonic (string) | 3 bits | Returns the binary code of the `dest` mnemonic. |
| `comp` | mnemonic (string) | 7 bits | Returns the binary code of the `comp` mnemonic. |
| `jump` | mnemonic (string) | 3 bits | Returns the binary code of the `jump` mnemonic. |

### 6.3.3 Assembler for Programs with No Symbols

We suggest building the assembler in two stages. In the first stage, write an assembler that translates assembly programs without symbols. This can be done using the Parser and Code modules just described. In the second stage, extend the assembler with symbol handling capabilities, as we explain in the next section.

The contract for the first symbol-less stage is that the input `Prog.asm` program contains no symbols. This means that (a) in all address commands of type `@Xxx` the `Xxx` constants are decimal numbers and not symbols, and (b) the input file contains no label commands, namely, no commands of type `(Xxx)`.

The overall symbol-less assembler program can now be implemented as follows. First, the program opens an output file named `Prog.hack`. Next, the program marches through the lines (assembly instructions) in the supplied `Prog.asm` file. For each *C*-instruction, the program concatenates the translated binary codes of the instruction fields into a single 16-bit word. Next, the program writes this word into the `Prog.hack` file. For each *A*-instruction of type `@Xxx`, the program translates the decimal constant returned by the parser into its binary representation and writes the resulting 16-bit word into the `Prog.hack` file.

### 6.3.4 The *SymbolTable* Module

Since Hack instructions can contain symbols, the symbols must be resolved into actual addresses as part of the translation process. The assembler deals with this task using a *symbol table*, designed to create and maintain the correspondence between symbols and their meaning (in Hack's case, RAM and ROM addresses). A natural data structure for representing such a relationship is the classical *hash table*. In most programming languages, such a data structure is available as part of a standard library, and thus there is no need to develop it from scratch. We propose the following API.

**SymbolTable:** Keeps a correspondence between symbolic labels and numeric addresses.

| Routine | Arguments | Returns | Function |
|---|---|---|---|
| `Constructor` | — | — | Creates a new empty symbol table. |
| `addEntry` | `symbol` (string), `address` (int) | — | Adds the pair (`symbol`, `address`) to the table. |
| `contains` | `symbol` (string) | Boolean | Does the symbol table contain the given `symbol`? |
| `GetAddress` | `symbol` (string) | int | Returns the address associated with the `symbol`. |

### 6.3.5 Assembler for Programs with Symbols

Assembly programs are allowed to use symbolic labels (destinations of *goto* commands) before the symbols are defined. This convention makes the life of assembly

programmers easier and that of assembler developers harder. A common solution to this complication is to write a two-pass assembler that reads the code twice, from start to end. In the first pass, the assembler builds the symbol table and generates no code. In the second pass, all the label symbols encountered in the program have already been bound to memory locations and recorded in the symbol table. Thus, the assembler can replace each symbol with its corresponding meaning (numeric address) and generate the final binary code.

Recall that there are three types of symbols in the Hack language: *predefined symbols*, *labels*, and *variables*. The symbol table should contain and handle all these symbols, as follows.

**Initialization** Initialize the symbol table with all the predefined symbols and their pre-allocated RAM addresses, according to section 6.2.3.

**First Pass** Go through the entire assembly program, line by line, and build the symbol table without generating any code. As you march through the program lines, keep a running number recording the ROM address into which the current command will be eventually loaded. This number starts at 0 and is incremented by 1 whenever a *C*-instruction or an *A*-instruction is encountered, but does not change when a label pseudocommand or a comment is encountered. Each time a pseudocommand `(Xxx)` is encountered, add a new entry to the symbol table, associating `Xxx` with the ROM address that will eventually store the next command in the program. This pass results in entering all the program's *labels* along with their ROM addresses into the symbol table. The program's variables are handled in the second pass.

**Second Pass** Now go again through the entire program, and parse each line. Each time a symbolic *A*-instruction is encountered, namely, `@Xxx` where `Xxx` is a symbol and not a number, look up `Xxx` in the symbol table. If the symbol is found in the table, replace it with its numeric meaning and complete the command's translation. If the symbol is not found in the table, then it must represent a new variable. To handle it, add the pair (`Xxx`, *n*) to the symbol table, where *n* is the next available RAM address, and complete the command's translation. The allocated RAM addresses are consecutive numbers, starting at address 16 (just after the addresses allocated to the predefined symbols).

This completes the assembler's implementation.

## 6.4 Perspective

Like most assemblers, the Hack assembler is a relatively simple program, dealing mainly with text processing. Naturally, assemblers for richer machine languages are more complex. Also, some assemblers feature more sophisticated symbol handling capabilities not found in Hack. For example, the assembler may allow programmers to explicitly associate symbols with particular data addresses, to perform "constant arithmetic" on symbols (e.g., to use `table+5` to refer to the fifth memory location after the address referred to by `table`), and so on. Additionally, many assemblers are capable of handling *macro commands*. A macro command is simply a sequence of machine instructions that has a name. For example, our assembler can be extended to translate an agreed-upon macro-command, say `D=M[`*xxx*`]`, into the two instructions *@xxx* followed immediately by `D=M` (*xxx* being an address). Clearly, such macro commands can considerably simplify the programming of commonly occurring operations, at a low translation cost.

We note in closing that stand-alone assemblers are rarely used in practice. First, assembly programs are rarely written by humans, but rather by compilers. And a compiler—being an automaton—does not have to bother to generate symbolic commands, since it may be more convenient to directly produce binary machine code. On the other hand, many high-level language compilers allow programmers to embed segments of assembly language code within high-level programs. This capability, which is rather common in C language compilers, gives the programmer direct control of the underlying hardware, for optimization.

## 6.5 Project

**Objective** Develop an assembler that translates programs written in Hack assembly language into the binary code understood by the Hack hardware platform. The assembler must implement the translation specification described in section 6.2.

**Resources** The only tool needed for completing this project is the programming language in which you will implement your assembler. You may also find the following two tools useful: the assembler and CPU emulator supplied with the book. These tools allow you to experiment with a working assembler before you set out to build one yourself. In addition, the supplied assembler provides a visual

line-by-line translation GUI and allows online code comparisons with the outputs that *your* assembler will generate. For more information about these capabilities, refer to the assembler tutorial (part of the book's software suite).

**Contract** When loaded into your assembler, a `Prog.asm` file containing a valid Hack assembly language program should be translated into the correct Hack binary code and stored in a `Prog.hack` file. The output produced by your assembler must be identical to the output produced by the assembler supplied with the book.

**Building Plan** We suggest building the assembler in two stages. First write a symbol-less assembler, namely, an assembler that can only translate programs that contain no symbols. Then extend your assembler with symbol handling capabilities. The test programs that we supply here come in two such versions (without and with symbols), to help you test your assembler incrementally.

**Test Programs** Each test program except the first one comes in two versions: `ProgL.asm` is symbol-less, and `Prog.asm` is with symbols.

*Add:* Adds the constants 2 and 3 and puts the result in R0.

*Max:* Computes max(R0, R1) and puts the result in R2.

*Rect:* Draws a rectangle at the top left corner of the screen. The rectangle is 16 pixels wide and R0 pixels high.

*Pong:* A single-player Ping-Pong game. A ball bounces constantly off the screen's "walls." The player attempts to hit the ball with a bat by pressing the left and right arrow keys. For every successful hit, the player gains one point and the bat shrinks a little to make the game harder. If the player misses the ball, the game is over. To quit the game, press `ESC`.

The *Pong* program was written in the *Jack* programming language (chapter 9) and translated into the supplied assembly program by the *Jack compiler* (chapters 10–11). Although the original Jack program is only about 300 lines of code, the executable Pong application is about 20,000 lines of binary code, most of which being the Jack operating system (chapter 12). Running this interactive program in the CPU emulator is a slow affair, so don't expect a high-powered Pong game. This slowness is actually a virtue, since it enables your eye to track the graphical behavior of the program. In future projects in the book, this game will run much faster.

**Steps** Write and test your assembler program in the two stages described previously. You may use the assembler supplied with the book to compare the output of your assembler to the correct output. This testing procedure is described next. For more information about the supplied assembler, refer to the assembler tutorial.

**The Supplied Assembler** The practice of using the supplied assembler (which produces correct binary code) to test another assembler (which is not necessarily correct) is illustrated in figure 6.3. Let `Prog.asm` be some program written in Hack assembly. Suppose that we translate this program using the supplied assembler, producing

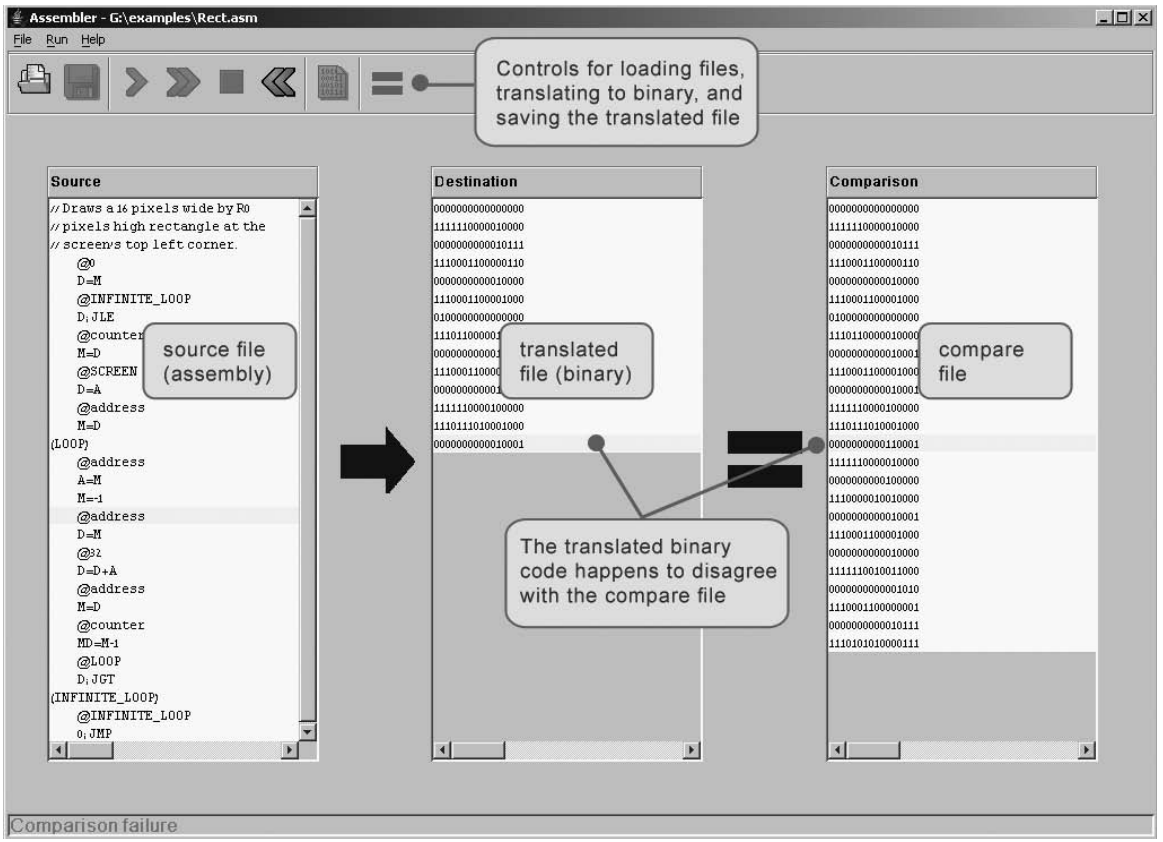



**Figure 6.3** Using the supplied assembler to test the code generated by another assembler.

a binary file called `Prog.hack`. Next, we use another assembler (e.g., the one that you wrote) to translate the same program into another file, say `Prog1.hack`. Now, if the latter assembler is working correctly, it follows that `Prog.hack` = `Prog1.hack`. Thus, one way to test a newly written assembler is to load `Prog.asm` into the supplied assembler program, load `Prog1.hack` as a compare file, then translate and compare the two binary files (see figure 6.3). If the comparison fails, the assembler that produced `Prog1.hack` must be buggy; otherwise, it may be error-free.