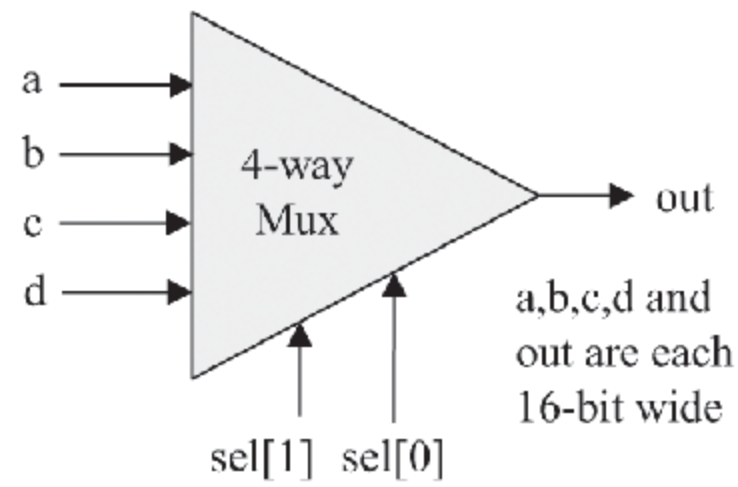
sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d



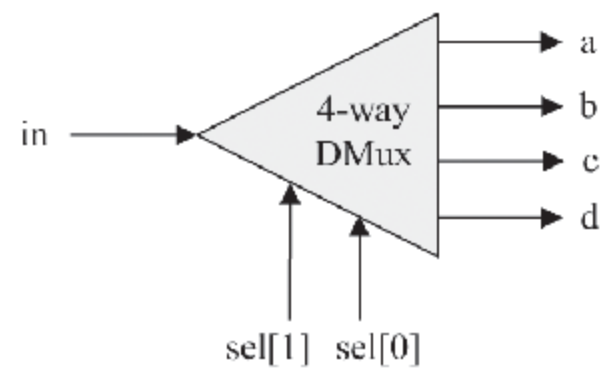
Our target computer platform requires two variants of this chip: a 4-way 16-bit multiplexer and an 8-way 16-bit multiplexer:

```
Chip name: Mux4Way16
Input:     a[16],b[16],c[16],d[16],sel[2]
Output:    out[16]
Function:  if (sel==00,01,10, or 11) then out = a,b,c, or d
Comment:   The assignment is a 16-bit operation.
           For example, "out = a" means "for i = 0..15 out[i] = a[i]".

Chip name: Mux8Way16
Input:     a[16],b[16],c[16],d[16],e[16],f[16],g[16],h[16], sel[3]
Output:    out[16]
Function:  if (sel==000,001,010, ..., or 111)
           then out = a,b,c,d, ..., or h
Comment:   The assignment is a 16-bit operation.
           For example, "out = a" means "for i = 0..15 out[i] = a[i]".
```

Multi-way/Multi-bit demultiplexer: An *m*-way *n*-bit demultiplexer routes its single *n*-bit input to one of its *m* *n*-bit outputs. The other outputs are set to 0. The selection is specified by a set of *k* selection bits, where $k = log_2 m$. Here is the API of a 4-way demultiplexer:

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



Our target computer platform requires two variants of this chip: a 4-way 1-bit demultiplexer and an 8-way 1-bit demultiplexer:

```
Chip name: DMux4Way
Input:     in, sel[2]
Output:    a, b, c, d
Function:  if (sel==00)      then {a,b,c,d} = {1,0,0,0},
           else if (sel==01) then {a,b,c,d} = {0,1,0,0},
           else if (sel==10) then {a,b,c,d} = {0,0,1,0},
           else if (sel==11) then {a,b,c,d} = {0,0,0,1}

Chip name: Dmux8Way
Input:     in, sel[3]
Output:    a, b, c, d, e, f, g, h
Function:  if (sel==000)      then {a,b,c,…, h} = {1,0,0,0,0,0,0,0},
           else if (sel==001) then {a,b,c,…, h} = {0,1,0,0,0,0,0,0},
           else if (sel==010) then {a,b,c,…, h} = {0,0,1,0,0,0,0,0},
           …
           else if (sel==111) then {a,b,c,…, h} = {0,0,0,0,0,0,0,1}
```

1.5 Implementation

The previous section described the specifications, or interfaces, of a family of basic logic gates. Having described the *what*, we now turn to discuss the *how*. In particular, we'll focus on two general approaches to implementing logic gates: *behavioral simulation* and *hardware implementation*. Both approaches play important roles in all our hardware construction projects.

1.5.1 Behavioral Simulation

The chip descriptions presented thus far are strictly abstract. It would have been nice if we could experiment with these abstractions hands-on, before setting out to build them in HDL. How can we possibly do so?

Well, if all we want to do is interact with the chips' behavior, we don't have to go through the trouble of building them in HDL. Instead, we can opt for a much simpler implementation, using conventional programming. For example, we can use some object-oriented language to create a set of classes, each implementing a generic chip. We can write class constructors

for creating chip instances and *eval* methods for evaluating their logic, and we can have the classes interact with each other so that high-level chips can be defined in terms of lower-level ones. We could then add a nice graphical user interface that enables putting different values in the chip inputs, evaluating their logic, and observing the chip outputs. This software-based technique, called *behavioral simulation*, makes a lot of sense. It enables experimenting with chip interfaces before starting the laborious process of building them in HDL.

The Nand to Tetris *hardware simulator* provides exactly such a service. In addition to simulating the behavior of HDL programs, which is its main purpose, the simulator features built-in software implementations of all the chips built in the Nand to Tetris hardware projects. The built-in version of each chip is implemented as an executable software module, invoked by a skeletal HDL program that provides the chip interface. For example, here is the HDL program that implements the built-in version of the Xor chip:

```
/* Xor (exclusive or) gate:
   If a!=b out=1 else out=0. */
CHIP Xor {
    IN  a, b;
    OUT out;
    BUILTIN Xor;
}
```

Compare this to the HDL program listed in figure 1.7. First, note that regular chips and built-in chips have precisely the same interface. Thus, they provide exactly the same functionality. In the built-in implementation though, the PARTS section is replaced with the single statement BUILTIN Xor. This statement informs the simulator that the chip is implemented by Xor.class. This class file, like all the Java class files that implement built-in chips, is located in the folder nand2tetris/tools/builtIn.

We note in passing that realizing logic gates using high-level programming is not difficult, and that's another virtue of behavioral simulation: it's inexpensive and quick. At some point, of course, hardware engineers must do the real thing, which is implementing the chips not as

software artifacts but rather as HDL programs that can be committed to silicon. That's what we'll do next.

1.5.2 Hardware Implementation

This section gives guidelines on how to implement the fifteen logic gates described in this chapter. As a rule in this book, our implementation guidelines are intentionally brief. We give just enough insights to get started, leaving you the pleasure of discovering the rest of the gate implementations yourself.

Nand: Since we decided to base our hardware on elementary Nand gates, we treat Nand as a primitive gate whose functionality is given externally. The supplied hardware simulator features a built-in implementation of Nand, and thus there is no need to implement it.

Not: Can be implemented using a single Nand gate. *Tip:* Inspect the Nand truth table, and ask yourself how the Nand inputs can be arranged so that a single input signal, 0, will cause the Nand gate to output 1, and a single input signal, 1, will cause it to output 0.

And: Can be implemented from the two previously discussed gates.

Or / Xor: The Boolean function Or can be defined using the Boolean functions And and Not. The Boolean function Xor can be defined using And, Not, and Or.

Multiplexer / Demultiplexer: Can be implemented using previously built gates.

Multi-bit Not / And / Or gates: Assuming that you've already built the basic versions of these gates, the implementation of their *n*-ary versions is a matter of arranging arrays of *n* basic gates and having each gate operate separately on its single-bit inputs. The resulting HDL code will be somewhat boring and repetitive (using copy-paste), but it will carry its weight when these multi-bit gates are used in the construction of more complex chips, later in the book.

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

Multi-way gates: Implementation tip: Think forks.

1.5.3 Built-In Chips

As we pointed out when we discussed behavioral simulation, our hardware simulator provides software-based, built-in implementations of most of the chips described in the book. In Nand to Tetris, the most celebrated built-in chip is of course Nand: whenever you use a Nand chip-part in an HDL program, the hardware simulator invokes the built-in tools/builtIn/Nand.hdl implementation. This convention is a special case of a more general chip invocation strategy: whenever the hardware simulator encounters a chip-part, say, *Xxx*, in an HDL program, it looks up the file *Xxx*.hdl in the current folder; if the file is found, the simulator evaluates its underlying HDL code. If the file is not found, the simulator looks it up in the tools/builtIn folder. If the file is found there, the simulator executes the chip's built-in implementation; otherwise, the simulator issues an error message and terminates the simulation.

This convention comes in handy. For example, suppose you began implementing a Mux.hdl program, but, for some reason, you did not complete it. This could be an annoying setback, since, in theory, you cannot continue building chips that use Mux as a chip-part. Fortunately, and actually by design, this is where built-in chips come to the rescue. All you have to do is rename your partial implementation Mux1.hdl, for example. Each time the hardware simulator is called to simulate the functionality of a Mux chip-part, it will fail to find a Mux.hdl file in the current folder. This will cause behavioral simulation to kick in, forcing the simulator to use the built-in Mux version instead. Exactly what we want! At a later stage you may want to go back to Mux1.hdl and resume working on its implementation. At this point you can restore its original file name, Mux.hdl, and continue from where you left off.

1.6 Project

This section describes the tools and resources needed for completing project 1 and gives recommended implementation steps and tips.

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: We assume that you've already downloaded the Nand to Tetris zip file, containing the book's software suite, and that you've extracted it into a folder named nand2tetris on your computer. If that is the case, then the nand2tetris/tools folder on your computer contains the hardware simulator discussed in this chapter. This program, along with a plain text editor, are the only tools needed for completing project 1 as well as all the other hardware projects described in the book.

The fifteen chips mentioned in this chapter, except for Nand, should be implemented in the HDL language described in appendix 2. For each chip *Xxx*, we provide a skeletal *Xxx*.hdl program (sometimes called a *stub file*) with a missing implementation part. In addition, for each chip we provide an *Xxx*.tst script that tells the hardware simulator how to test it, along with an *Xxx*.cmp compare file that lists the correct output that the supplied test is expected to generate. All these files are available in your nand2tetris/projects/01 folder. Your job is to complete and test all the *Xxx*.hdl files in this folder. These files can be written and edited using any plain text editor.

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 the actual outputs generated by the simulator disagree with the desired outputs, the simulator will stop the simulation and produce an error message.

Steps: We recommend proceeding in the following order:

0. The *hardware simulator* needed for this project is available in nand2tetris/tools.
1. Consult appendix 2 (HDL), as needed.
2. Consult the Hardware Simulator Tutorial (available at www.nand2tetris.org), as needed.
3. Build and simulate all the chips listed in nand2tetris/projects/01.

General Implementation Tips

(We use the terms *gate* and *chip* interchangeably.)

- Each gate can be implemented in more than one way. The simpler the implementation, the better. As a general rule, strive to use as few chip-parts as possible.
- Although each chip can be implemented directly from Nand gates only, we recommend always using composite gates that were already implemented. See the previous tip.
- There is no need to build "helper chips" of your own design. Your HDL programs should use only the chips mentioned in this chapter.
- Implement the chips in the order in which they appear in the chapter. If, for some reason, you don't complete the HDL implementation of some chip, you can still use it as a chip-part in other HDL programs. Simply rename the chip file, or remove it from the folder, causing the simulator to use its built-in version instead.

A web-based version of project 1 is available at www.nand2tetris.org.

1.7 Perspective

This chapter specified a set of basic logic gates that are widely used in computer architectures. In chapters 2 and 3 we will use these gates for building our processing and storage chips, respectively. These chips, in turn, will be later used for constructing the central processing unit and the memory devices of our computer.

Although we have chosen to use Nand as our basic building block, other logic gates can be used as possible points of departure. For example, you can build a complete computer platform using Nor gates only or, alternatively, a combination of And, Or, and Not gates. These constructive approaches to logic design are theoretically equivalent, just like the same geometry can be founded on alternative sets of agreed-upon axioms. In principle, if electrical engineers or physicists can come up with efficient and low-cost implementations of logic gates using any technology that they see fit, we will happily use them as primitive building blocks. The reality, though, is that most computers are built from either Nand or Nor gates.

Throughout the chapter, we paid no attention to efficiency and cost considerations, such as energy consumption or the number of wire crossovers implied by our HDL programs. Such considerations are critically important in practice, and a great deal of computer science and technology expertise focuses on optimizing them. Another issue we did not address is physical aspects, for example, how primitive logic gates can be built from transistors embedded in silicon or from other switching technologies. There are of course several such implementation options, each having its own characteristics (speed, energy consumption, production cost, and so on). Any nontrivial coverage of these issues requires venturing into areas outside computer science, like electrical engineering and solid-state physics.

The next chapter describes how bits can be used to represent binary numbers and how logic gates can be used to realize arithmetic operations. These capabilities will be based on the elementary logic gates built in this chapter.

2 Boolean Arithmetic

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

In this chapter we build a family of chips designed to represent numbers and perform arithmetic operations. Our starting point is the set of logic gates built in chapter 1, and our ending point is a fully functional *Arithmetic Logic Unit*. The ALU will later become the computational centerpiece of the *Central Processing Unit* (CPU)—the chip that executes all the instructions handled by the computer. Hence, building the ALU is an important milestone in our Nand to Tetris journey.

As usual, we approach this task gradually, starting with a background section that describes how binary codes and Boolean arithmetic can be used, respectively, to represent and add signed integers. The Specification section presents 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 surprisingly simple logic design. The Implementation and Project sections provide tips and guidelines on how to build the adder chips and the ALU using HDL and the supplied hardware simulator.

2.1 Arithmetic Operations

General-purpose computer systems are required to perform at least the following arithmetic operations on signed integers:

- addition
- sign conversion
- subtraction
- comparison
- multiplication
- division

We'll start by developing gate logic that carries out addition and sign conversion. Later, we will show how the other arithmetic operations can be implemented from these two building blocks.

In mathematics as well as in computer science, *addition* is a simple operation that runs deep. Remarkably, all the functions performed by digital computers—not only arithmetic operations—can be reduced to adding binary numbers. Therefore, constructive understanding of binary addition holds the key to understanding many fundamental operations performed by the computer's hardware.

2.2 Binary Numbers

When we are told that a certain code, say, 6083, represents a number using the *decimal system*, then, by convention, we take this number to be:

$$(6083)_{10} = 6 \cdot 10^3 + 0 \cdot 10^2 + 8 \cdot 10^1 + 3 \cdot 10^0 = 6083$$

Each digit in the decimal code contributes a value that depends on the *base* 10 and on the digit's position in the code. Suppose now that we are told that the code 10011 represents a number using base 2, or *binary* representation. To compute the value of this number, we follow exactly the same procedure, using base 2 instead of base 10:

$$(10011)_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

Inside computers, *everything* is represented using binary codes. For example, when we press the keyboard keys labeled 1, 9, and Enter in response to "Give an example of a prime number," what ends up stored in the computer's memory is the binary code 10011. When we ask the computer to display this value on the screen, the following process ensues. First, the computer's operating system calculates the decimal value that 10011 represents, which happens to be 19. After converting this integer value to the two characters 1 and 9, the OS looks up the current font and gets the two bitmap images used for rendering these characters on the screen. The OS then causes the screen driver to turn on and off the relevant pixels, and, don't hold your breath—the whole thing lasts a tiny fraction of a second—we finally see the image 19 appear on the screen.

In chapter 12 we'll develop an operating system that carries out such rendering operations, among many other low-level services. For now, suffice it to observe that the decimal representation of numbers is a human indulgence explained by the obscure fact that, at some point in ancient history, humans decided to represent quantities using their ten fingers, and the habit stuck. From a mathematical perspective, the number ten is utterly uninteresting, and, as far as computers go, is a complete nuisance. Computers handle *everything* in binary and care naught about decimal. Yet since humans insist on dealing with numbers using decimal codes, computers have to work hard behind the scenes to carry out binary-to-decimal and decimal-to-binary conversions whenever humans want to see, or supply, numeric information. At all other times, computers stick to binary.

Fixed word size: Integer numbers are of course unbounded: for any given number x there are integers that are less than x and integers that are greater than x. Yet computers are finite machines that use a fixed word size for representing numbers. *Word size* is a common hardware term used for specifying the number of bits that computers use for representing a basic chunk of information—in this case, integer values. Typically, 8-, 16-, 32-, or 64-bit registers are used for representing integers.[1] The fixed word size implies that there is a limit on the number of values that these registers can represent.

For example, suppose we use 8-bit registers for representing integers. This representation can code $2^8 = 256$ different things. If we wish to represent only nonnegative integers, we can assign 00000000 for representing 0, 00000001 for representing 1, 00000010 for representing 2, 00000011 for representing 3, all the way up to assigning 11111111 for representing 255. In general, using n bits we can represent all the nonnegative integers ranging from 0 to $2^n - 1$.

What about representing negative numbers using binary codes? Later in the chapter we'll present a technique that meets this challenge in a most elegant and satisfying way.

And what about representing numbers that are greater than, or less than, the maximal and minimal values permitted by the fixed register size? Every high-level language provides abstractions for handling numbers that are as large or as small as we can practically want. These abstractions are typically implemented by lashing together as many n-bit registers as necessary for representing the numbers. Since executing arithmetic and logical operations on multi-word numbers is a slow affair, it is recommended to use this practice only when the application requires processing extremely large or extremely small numbers.

2.3 Binary Addition

A pair of binary numbers can be added bitwise from right to left, using the same decimal addition algorithm learned in elementary school. First, we add the two rightmost bits, also called the *least significant bits* (LSB) of the two binary numbers. Next, we add the resulting carry bit to the sum of the next pair of bits. We continue this lockstep process until the two left *most significant bits* (MSB) are added. Here is an example of this algorithm in action, assuming that we use a fixed word size of 4 bits:

```
  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
```

If the most significant bitwise addition generates a carry of 1, we have what is known as *overflow*. What to do with overflow is a matter of decision, and ours is to ignore it. Basically, we are content to guarantee that the result of adding any two n-bit numbers will be correct up to n bits. We note in passing that ignoring things is perfectly acceptable as long as one is clear and forthcoming about it.

2.4 Signed Binary Numbers

An n-bit binary system can code 2^n different things. If we have to represent signed (positive and negative) numbers in binary code, a natural solution is to split the available code space into two subsets: one for representing nonnegative numbers, and the other for representing negative numbers. Ideally, the coding scheme should be chosen such that the introduction of signed numbers would complicate the hardware implementation of arithmetic operations as little as possible.

Over the years, this challenge has led to the development of several coding schemes for representing signed numbers in binary code. The solution used today in almost all computers is called the *two's complement* method, also known as *radix complement*. In a binary system that uses a word size of n bits, the two's complement binary code that represents negative x is taken to be the code that represents $2^n - x$. For example, in a 4-bit binary system, -7 is represented using the binary code associated with $2^4 - 7 = 9$, which happens to be 1001. Recalling that +7 is represented by 0111, we see that $1001 + 0111 = 0000$ (ignoring the overflow bit). Figure 2.1 lists all the signed numbers represented by a 4-bit system using the two's complement method.

0000:	0	
0001:	1	
0010:	2	
0011:	3	
0100:	4	
0101:	5	
0110:	6	
0111:	7	
1000:	–8	(16 – 8)
1001:	–7	(16 – 7)
1010:	–6	(16 – 6)
1011:	–5	(16 – 5)
1100:	–4	(16 – 4)
1101:	–3	(16 – 3)
1110:	–2	(16 – 2)
1111:	–1	(16 – 1)

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

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

- The system codes 2^n signed numbers, ranging from $-(2^{n-1})$ to $2^{n-1} - 1$.
- The code of any nonnegative number begins with a 0.
- The code of any negative number begins with a 1.
- To obtain the binary code of $-x$ from the binary code of x, leave all the least significant 0-bits and the first least significant 1-bit of x intact, and flip all the remaining bits (convert 0's to 1's and vice versa). Alternatively, flip all the bits of x and add 1 to the result.

A particularly attractive feature of the two's complement representation is that *subtraction* is handled as a special case of addition. To illustrate, consider $5-7$. Noting that this is equivalent to $5+(-7)$, and following figure 2.1, we proceed to compute `0101 + 1001`. The result is `1110`, which indeed is the binary code of -2. Here is another example: To compute $(-2)+(-3)$, we add `1110 + 1101`, obtaining the sum `11011`. Ignoring the overflow bit, we get `1011`, which is the binary code of -5.

We see that the two's complement method enables adding and subtracting signed numbers using nothing more than the hardware required for adding nonnegative numbers. As we will see later in the book, every arithmetic operation, from multiplication to division to square root, can be implemented reductively using binary addition. So, on the one hand, we observe that a huge range of computer capabilities rides on top of binary addition, and on the other hand, we observe that the two's complement method obviates the need for special hardware for adding and subtracting signed numbers. Taking these two observations together, we are compelled to conclude that the two's complement method is one of the most remarkable and unsung heroes of applied computer science.

2.5 Specification

We now turn to specifying a hierarchy of chips, starting with simple adders and culminating with an Arithmetic Logic Unit (ALU). As usual in this book, we focus first on the abstract (*what* the chips are designed to), delaying implementation details (*how* they do it) to the next section. We cannot resist reiterating, with pleasure, that thanks to the two's complement method we don't have to say anything special about handling signed numbers. All the arithmetic chips that we'll present work equally well on nonnegative, negative, and mixed-sign numbers.

2.5.1 Adders

We'll focus on the following hierarchy of *adders*:

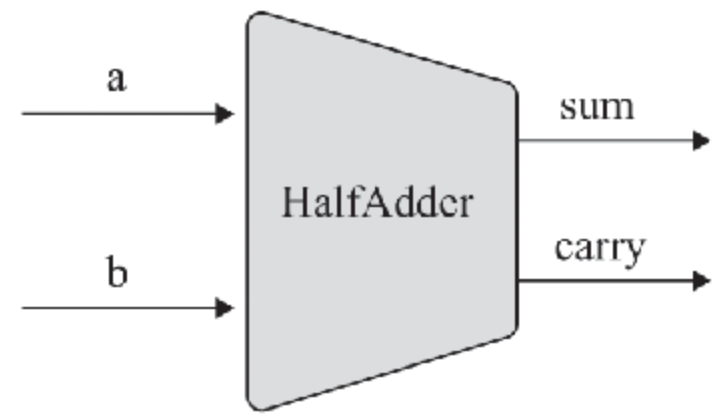
- *Half-adder*: designed to add two bits

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

We'll also specify a special-purpose adder, called an *incrementer*, designed to add 1 to a given number. (The names *half-adder* and *full-adder* derive from the implementation detail that a full-adder chip can be realized from two half-adders, as we'll see later in the chapter.)

Half-adder: The first step on our road to adding binary numbers is adding two bits. Let us treat the result of this operation as a 2-bit number, and call its right and left bits sum and carry, respectively. Figure 2.2 presents a chip that carries out this addition operation.

inputs		*outputs*	
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

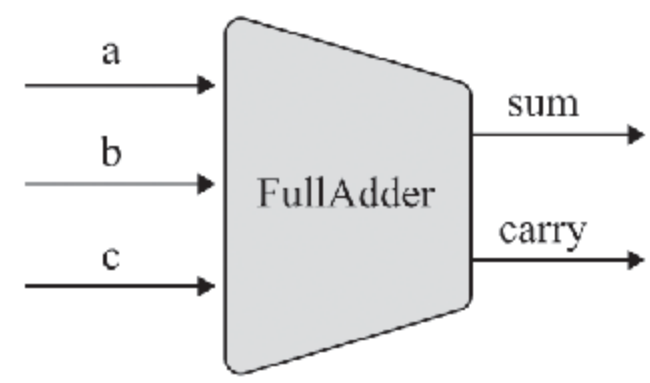



```
Chip name: HalfAdder
Input:     a, b
Output:    sum, carry
Function:  sum   = LSB of a + b
           carry = MSB of a + b
```

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

Full-adder: Figure 2.3 presents a *full-adder* chip, designed to add three bits. Like the half-adder, the full-adder chip outputs two bits that, taken together, represent the addition of the three input bits.

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

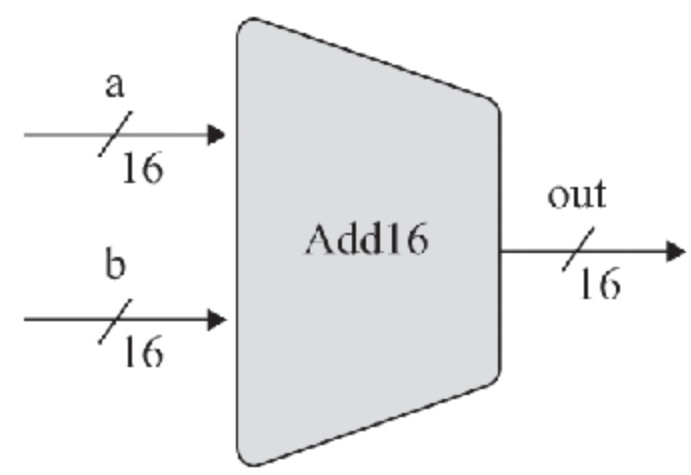



```
Chip name: FullAdder
Input:     a, b, c
Output:    sum, carry
Function:  sum   = LSB of a + b + c
           carry = MSB of a + b + c
```

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

Adder: Computers represent integer numbers using a fixed word size like 8, 16, 32, or 64 bits. The chip whose job is to add two such *n*-bit numbers is called *adder*. Figure 2.4 presents a 16-bit adder.

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




```
Chip name: Add16
Input:     a[16], b[16]
Output:    out[16]
Function:  Adds two 16-bit numbers.
           The overflow bit is ignored.
```

Figure 2.4 16-bit adder, designed to add two 16-bit numbers, with an example of addition action (on the left).

We note in passing that the logic design for adding 16-bit numbers can be easily extended to implement any *n*-bit adder chip, irrespective of *n*.

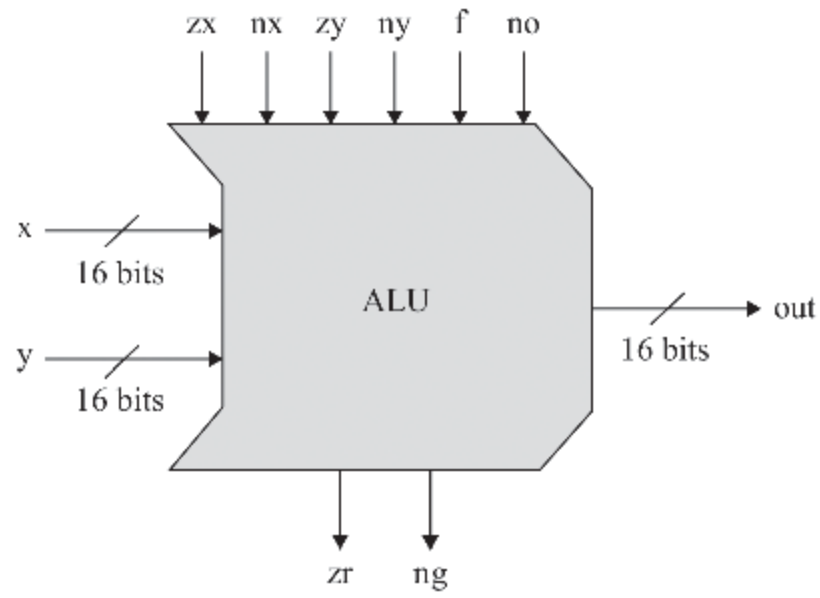
Incrementer: When we later design our computer architecture, we will need a chip that adds 1 to a given number (*Spoiler*: This will enable fetching the next instruction from memory, after executing the current one). Although the $x+1$ operation can be realized by our general-propose Adder chip, a dedicated *incrementer* chip can do it more efficiently. Here is the chip interface:

```
Chip name: Inc16
Input:     in[16]
Output:    out[16]
Function:  out = in + 1
Comment:   The overflow bit is ignored.
```

2.5.2 The Arithmetic Logic Unit

All the adder chips presented so far are generic: any computer that performs arithmetic operations uses such chips, one way or another. Building on these chips, we now turn to describe an *Arithmetic Logic Unit*, a chip that will later become the computational centerpiece of our CPU. Unlike the generic gates and chips discussed thus far, the ALU design is unique to the computer built in Nand to Tetris, named Hack. That said, the design principles underlying the Hack ALU are general and instructive. 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.

As its name implies, an Arithmetic Logic Unit is a chip designed to compute a set of arithmetic and logic operations. Exactly *which* operations an ALU should feature is a design decision derived from cost-effectiveness considerations. In the case of the Hack platform, we decided that (i) the ALU will perform only integer arithmetic (and not, for example, floating point arithmetic) and (ii) the ALU will compute the set of eighteen arithmetic-logical functions shown in figure 2.5a.




```
out(x, y, zx,nx,zy,ny,f,no) =
    0, 1, -1,
    x, y, !x, !y, -x, -y,
    x+1, y+1, x-1, y-1,
    x+y, x-y, y-x,
    x&y, x|y
```

Figure 2.5a The Hack ALU, designed to compute the eighteen arithmetic-logical functions shown on the right (the symbols !, &, and | represent, respectively, the 16-bit operations Not, And, and Or). For now, ignore the zr and ng output bits.

As seen in figure 2.5a, the Hack ALU operates on two 16-bit two's complement integers, denoted x and y, and on six 1-bit inputs, called *control bits*. These control bits "tell" the ALU which function to compute. The exact specification is given in figure 2.5b.

pre-setting the x input		pre-setting the y input		computing + or &	post-setting the output	resulting ALU output
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	out(x,y) =
zx	**nx**	**zy**	**ny**	**f**	**no**	**out**
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 2.5b Taken together, the values of the six control bits zx, nx, zy, ny, f, and no cause the ALU to compute one of the functions listed in the rightmost column.

To illustrate the ALU logic, suppose we wish to compute the function $x-1$, for $x=27$. To get started, we feed the 16-bit binary code of 27 into the x input. In this particular example we don't care about y's value, since it has no impact on the required calculation. Now, looking up x-1 in figure 2.5b, we set the ALU's control bits to 001110. According to the specification, this setting should cause the ALU to output the binary code representing 26.

Is that so? To find out, let's delve deeper, and reckon how the Hack ALU performs its magic. Focusing on the top row of figure 2.5b, note that each one of the six control bits is associated with a standalone, conditional

micro-action. For example, the zx bit is associated with "if (zx==1) then set x to 0". These six directives are to be performed in order: first, we either set the x and y inputs to 0, or not; next, we either negate the resulting values, or not; next, we compute either + or & on the preprocessed values; and finally, we either negate the resulting value, or not. All these settings, negations, additions, and conjunctions are 16-bit operations.

With this logic in mind, let us revisit the row associated with x-1 and verify that the micro-operations coded by the six control bits will indeed cause the ALU to compute $x-1$. Going left to right, we see that the zx and nx bits are 0, so we neither zero nor negate the x input—we leave it as is. The zy and ny bits are 1, so we first zero the y input and then negate the result, yielding the 16-bit value 1111111111111111. Since this binary code happens to represent -1 in two's complement, we see that the two data inputs of the ALU are now x's value and -1. Since the f bit is 1, the selected operation is *addition*, causing the ALU to compute $x+(-1)$. Finally, since the no bit is 0, the output is not negated. To conclude, we've illustrated that if we feed the ALU with x and y values and set the six control bits to 001110, the ALU will compute $x-1$, as specified.

What about the other seventeen functions listed in figure 2.5b? Does the ALU actually compute them as well? To verify that this is indeed the case, you are invited to focus on other rows in the table, go through the same process of carrying out the micro-actions coded by the six control bits, and figure out for yourself what the ALU will output. Or, you can believe us that the ALU works as advertised.

Note that the ALU actually computes a total of sixty-four functions, since six control bits code that many possibilities. We've decided to focus on, and document, only eighteen of these possibilities, since these will suffice for supporting the instruction set of our target computer system. The curious reader may be intrigued to know that some of the undocumented ALU operations are quite meaningful. However, we've opted not to exploit them in the Hack system.

The Hack ALU interface is given in figure 2.5c. Note that in addition to computing the specified function on its two inputs, the ALU also computes the two output bits zr and ng. These bits, which flag whether the ALU output is zero or negative, respectively, will be used by the future CPU of our computer system.

```
Chip name: ALU
Input:     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,            // if f==1 out=add(x,y) else out=and(x,y)
           no            // Negate the out output
Output:    out[16],      // 16-bit output
           zr,           // if out==0 zr=1 else zr=0
           ng            // if out<0  ng=1 else ng=0
Function:
           if zx x=0      // 16-bit zero constant
           if nx x=!x     // Bit-wise negation
           if zy y=0      // 16-bit zero constant
           if ny y=!y     // Bit-wise negation
           if f out=x+y   // Integer two's complement addition
           else out=x&y   // Bit-wise And
           if no out=!out // Bit-wise negation
           if out==0 zr=1 else zr=0  // 16-bit equality comparison
           if out<0 ng=1  else ng=0  // two's complement comparison
Comment:   The overflow bit is ignored.
```

Figure 2.5c The Hack ALU API.

It may be instructive to describe the thought process that led to the design of our ALU. First, we made a tentative list of the primitive operations that we wanted our computer to perform (right column of figure 2.5b). 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 operation that can be easily implemented with basic logic gates. The resulting ALU is simple and elegant. And in the hardware business, simplicity and elegance carry the day.

2.6 Implementation

Our implementation guidelines are intentionally minimal. We already gave many implementation tips along the way, and now it is your turn to discover the missing parts in the chip architectures.

Throughout this section, when we say "build/implement a logic design that …," we expect you to (i) figure out the logic design (e.g., by sketching a gate diagram), (ii) write the HDL code that realizes the design, and (iii) test and debug your design using the supplied test scripts and hardware simulator. More details are given in the next section, which describes project 2.

Half-adder: An inspection of the truth table in figure 2.2 reveals that the outputs sum(a,b) and carry(a,b) happen to be identical to those of two simple Boolean functions discussed and implemented in project 1. Therefore, the half-adder implementation is straightforward.

Full-adder: A full-adder chip can be implemented from two half-adders and one additional gate (and that's why these adders are called *half* and *full)*. Other implementations are possible, including direct approaches that don't use half-adders.

Adder: The addition of two *n*-bit numbers can be done bitwise, from right to left. In step 0, the least significant pair of bits is added, and the resulting carry bit is fed into the addition of the next significant pair of bits. The process continues until the pair of the most significant bits is added. Note that each step involves the addition of three bits, one of which is propagated from the "previous" addition.

Readers may wonder how we can add pairs of bits "in parallel" before the carry bit has been computed by the previous pair of bits. The answer is that these computations are sufficiently fast as to complete and stabilize within one clock cycle. We'll discuss clock cycles and synchronization in the next chapter; for now, you can ignore the time element completely, and write HDL code that computes the addition operation by acting on all the bit-pairs simultaneously.

Incrementer: An *n*-bit incrementer can be implemented easily in a number of different ways.

ALU: Our ALU was carefully planned to effect all the desired ALU operations *logically*, using the simple Boolean operations implied by the six

control bits. Therefore, the *physical* implementation of the ALU can be reduced to implementing these simple operations, following the pseudocode specifications listed at the top of figure 2.5b. Your first step will likely be creating a logic design for zeroing and negating a 16-bit value. This logic can be used for handling the x and y inputs as well as the out output. Chips for bitwise And-ing and addition have already been built in projects 1 and 2, respectively. Thus, what remains is building logic that selects between these two operations, according to the f control bit (this selection logic was also implemented in project 1). Once this main ALU functionality works properly, you can proceed to implement the required functionality of the single-bit zr and ng outputs.

2.7 Project

Objective: Implement all the chips presented in this chapter. The only building blocks that you need are some of the gates described in chapter 1 and the chips that you will gradually build in this project.

Built-in chips: As was just said, the chips that you will build in this project use, as chip-parts, some of the chips described in chapter 1. Even if you've built these lower-level chips successfully in HDL, we recommend using their built-in versions instead. As best-practice advice pertaining to all the hardware projects in Nand to Tetris, always prefer using built-in chip-parts instead of their HDL implementations. The built-in chips are guaranteed to operate to specification and are designed to speed up the operation of the hardware simulator.

There is a simple way to follow this best-practice advice: Don't add to the project folder nand2tetris/projects/02 any .hdl file from project 1. Whenever the hardware simulator will encounter in your HDL code a reference to a chip-part from project 1, for example, And16, it will check whether there is an And16.hdl file in the current folder. Failing to find it, the hardware simulator will resort by default to using the built-in version of this chip, which is exactly what we want.

The remaining guidelines for this project are identical to those of project 1. In particular, remember that good HDL programs use as few chip-parts as

possible, and there is no need to invent and implement any "helper chips"; your HDL programs should use only chips that were specified in chapters 1 and 2.

A web-based version of project 2 is available at www.nand2tetris.org.

2.8 Perspective

The construction of the multi-bit adder presented in this chapter was standard, although no attention was paid to efficiency. Indeed, our suggested adder implementation is inefficient, due to the delays incurred while the carry bits propagate throughout the *n*-bit addends. This computation can be accelerated by logic circuits that effect so-called *carry lookahead* heuristics. Since addition is the most prevalent operation in computer architectures, any such low-level improvement can result in dramatic performance gains throughout the system. Yet in this book we focus mostly on functionality, leaving chip optimization to more specialized hardware books and courses.[2]

The overall functionality of any hardware/software system is delivered jointly by the CPU and the operating system that runs on top of the hardware platform. Thus, when designing a new computer system, the question of how to allocate the desired functionality between the ALU and the OS is essentially a cost/performance dilemma. As a rule, direct hardware implementations of arithmetic and logical operations are more efficient than software implementations but make the hardware platform more expensive.

The trade-off that we have chosen in Nand to Tetris is to design a basic ALU with minimal functionality, and use system software to implement additional mathematical operations as needed. For example, our ALU features neither multiplication nor division. In part II of the book, when we discuss the operating system (chapter 12), we'll implement elegant and efficient bitwise algorithms for multiplication and division, along with other mathematical operations. These OS routines can then be used by compilers of high-level languages operating on top of the Hack platform. Thus, when a high-level programmer writes an expression like, say, `x * 12 + sqrt(y)`, then, following compilation, some parts of the expression will be evaluated directly by the ALU and some by the OS, yet the high-level programmer will be completely oblivious to this low-level division of work. Indeed, one of the key roles of an operating system is closing gaps between the high-level language abstractions that programmers use and the barebone hardware on which they are realized.

---

1. Which correspond, respectively, to the typical high-level data types *byte*, *short*, *int*, and *long*. For example, when reduced to machine-level instructions, *short* variables can be handled by 16-bit registers. Since 16-bit arithmetic is four times faster than 64-bit arithmetic, programmers are advised to always use the most compact data type that satisfies the application's requirements.
2. A technical reason for not using carry look-ahead techniques in our adder chips is that their hardware implementation requires cyclical pin connections, which are not supported by the Nand to Tetris hardware simulator.

3 Memory

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

Consider the high-level operation `x=y+17`. In chapter 2 we showed how logic gates can be utilized for representing numbers and for computing simple arithmetic expressions like `y+17`. We now turn to discuss how logic gates can be used to *store values over time*—in particular, how a variable like `x` can be set to "contain" a value and persist it until we set it to another value. To do so, we'll develop a new family of *memory chips*.

So far, all the chips that we built in chapters 1 and 2, culminating with the ALU, were time independent. Such chips are sometimes called *combinational*: they respond to different combinations of their inputs without delay, except for the time it takes their inner chip-parts to complete the computation. In this chapter we introduce and build *sequential* chips. Unlike combinational chips, which are oblivious to time, the outputs of sequential chips depend not only on the inputs in the current time but also on inputs and outputs that have been processed previously.

Needless to say, the notions of *current* and *previous* go hand in hand with the notion of *time*: you remember now what was committed to memory before. Thus, before we start talking about memory, we must first figure out how to use logic to model the progression of time. This can be done using a *clock* that generates an ongoing train of binary signals that we call *tick* and *tock*. The time between the beginning of a tick and the end of the subsequent tock is called a *cycle*, and these cycles will be used to regulate the operations of all the memory chips used by the computer.

Following a brief, user-oriented introduction to memory devices, we will present the art of sequential logic, which we will use for building time-dependent chips. We will then set out to build registers, RAM devices, and counters. These memory devices, along with the arithmetic devices built in chapter 2, comprise all the chips needed for building a complete, general-purpose computer system—a challenge that we will take up in chapter 5.

3.1 Memory Devices

Computer programs use variables, arrays, and objects—abstractions that persist data over time. Hardware platforms support this ability by offering memory devices that know how to *maintain state*. Because evolution gave humans a phenomenal electro-chemical memory system, we tend to take for granted the ability to remember things over time. However, this ability is hard to implement in classical logic, which is aware of neither time nor state. Thus, to get started, we must first find a way to model the progression of time and endow logic gates with the ability to maintain state and respond to time changes.

We will approach this challenge by introducing a clock and an elementary, time-dependent logic gate that can flip and flop between two stable states: representing 0 and representing 1. This gate, called *data flip-flop* (DFF), is the fundamental building block from which all memory devices will be built. In spite of its central role, though, the DFF is a low-profile, inconspicuous gate: unlike registers, RAM devices, and counters, which play prominent roles in computer architectures, DFFs are used implicitly, as low-level chip-parts embedded deep within other memory devices.

The fundamental role of the DFF is seen clearly in figure 3.1, where it serves as the foundation of the memory hierarchy that we are about to build. We will show how DFFs can be used to create 1-bit registers and how *n* such registers can be lashed together to create an *n*-bit register. Next, we'll construct a RAM device containing an arbitrary number of such registers. Among other things, we'll develop a means for *addressing*, that is, accessing by address, any randomly chosen register from the RAM directly and instantaneously.

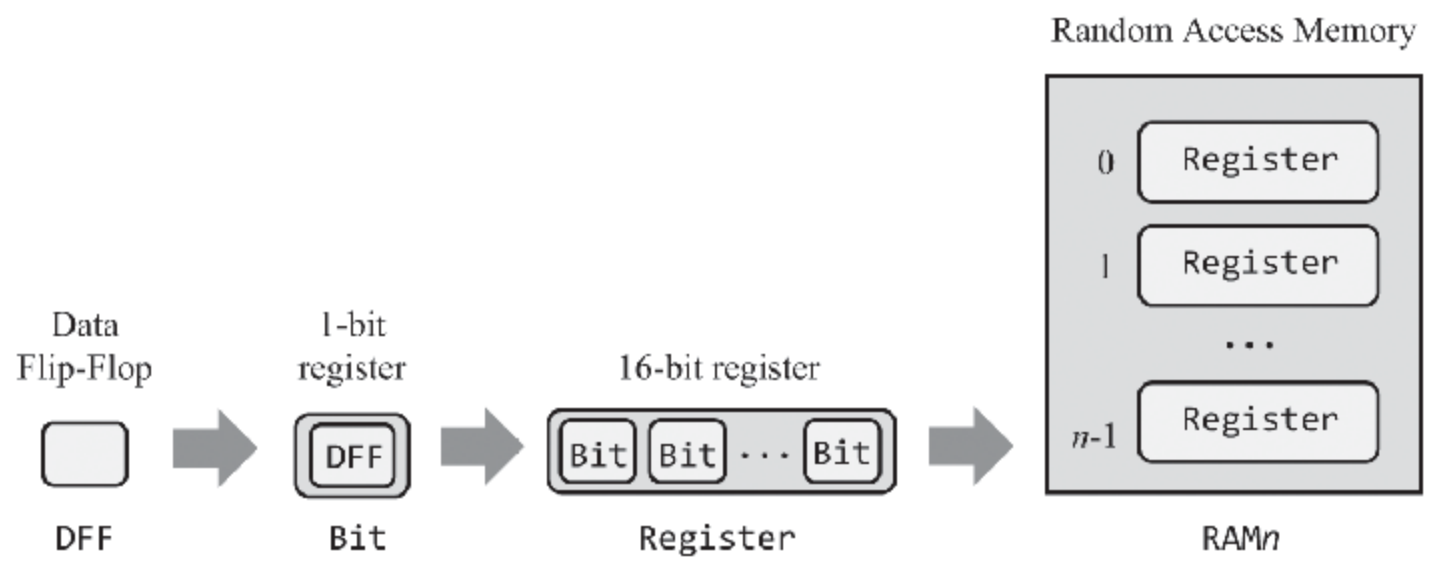


Figure 3.1 The memory hierarchy built in this chapter.

Before setting out to build these chips, though, we'll present a methodology and tools that enable modeling the progression of time and maintaining state over time.

3.2 Sequential Logic

All the chips discussed in chapters 1 and 2 were based on classical logic, which is time independent. In order to develop memory devices, we need to extend our gate logic with the ability to respond not only to input changes but also to the ticking of a clock: we remember the meaning of the word *dog* in time t since we remembered it in time $t-1$, all the way back to the point of time when we first committed it to memory. In order to develop this temporal ability to *maintain state*, we must extend our computer architecture with a time dimension and build tools that handle time using Boolean functions.

3.2.1 Time Matters

So far in our Nand to Tetris journey, we have assumed that chips respond to their inputs instantaneously: you input 7, 2, and "subtract" into the ALU, and … *poof!* the ALU output becomes 5. In reality, outputs are always delayed, due to at least two reasons. First, the inputs of the chips don't appear out of thin air; rather, the signals that represent them travel from the outputs of other chips, and this travel takes time. Second, the computations that chips perform also take time; the more chip-parts the chip has—the

more elaborate its logic—the more time it will take for the chip's outputs to emerge fully formed from the chip's circuitry.

Thus *time* is an issue that must be dealt with. As seen at the top of figure 3.2, time is typically viewed as a metaphorical arrow that progresses relentlessly forward. This progression is taken to be continuous: between every two time-points there is another time-point, and changes in the world can be infinitesimally small. This notion of time, which is popular among philosophers and physicists, is too deep and mysterious for computer scientists. Thus, instead of viewing time as a continuous progression, we prefer to break it into fixed-length intervals, called *cycles*. This representation is discrete, resulting in cycle 1, cycle 2, cycle 3, and so on. Unlike the continuous arrow of time, which has an infinite granularity, the cycles are atomic and indivisible: changes in the world occur only during cycle transitions; within cycles, the world stands still.

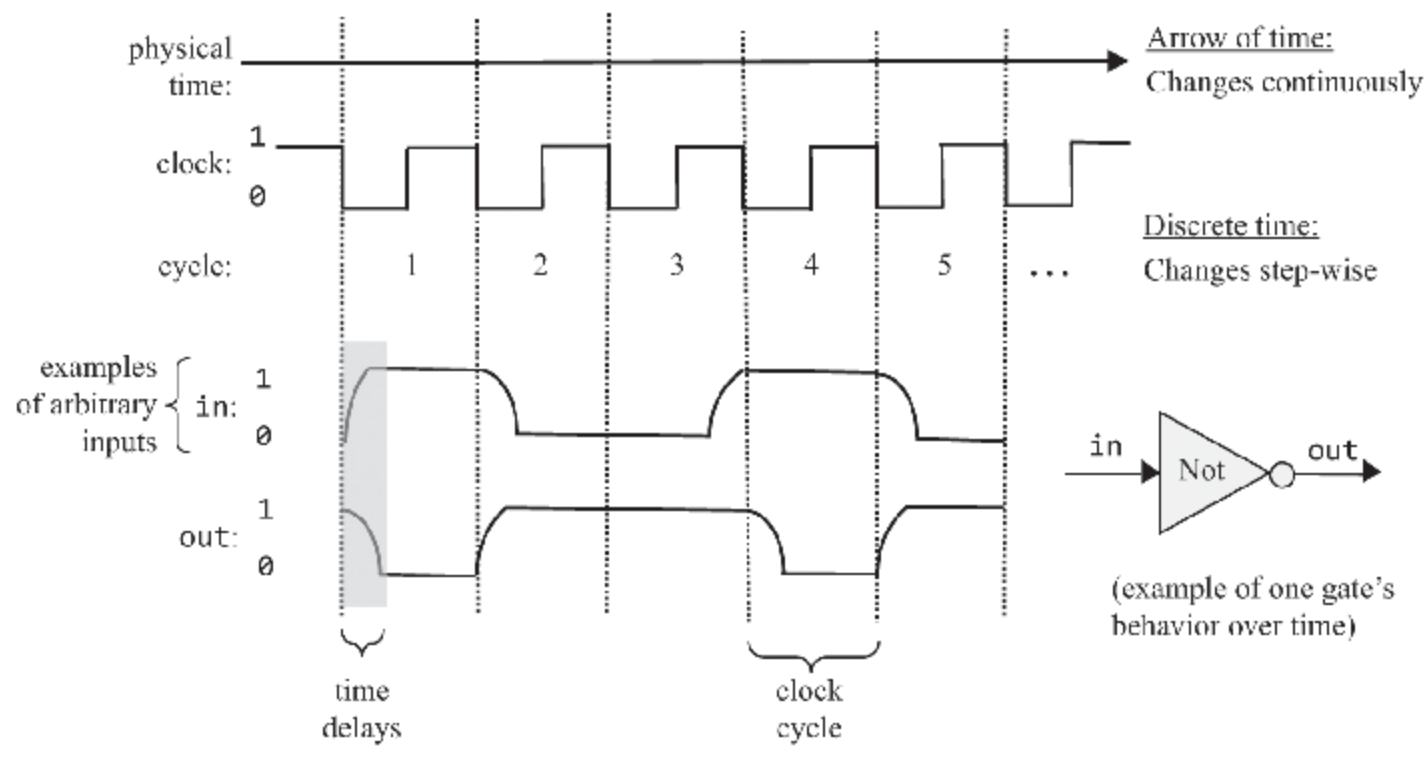


Figure 3.2 Discrete time representation: State changes (input and output values) are observed only during cycle transitions. Within cycles, changes are ignored.

Of course the world never stands still. However, by treating time discretely, we make a conscious decision to ignore continuous change. We are content to know the state of the world in cycle n, and then in cycle $n+1$, but *during* each cycle the state is assumed to be—well, we don't care. When it comes to building computer architectures, this discrete view of time serves two important design objectives. First, it can be used for neutralizing the randomness associated with communications and

computation time delays. Second, it can be used for synchronizing the operations of different chips across the system, as we'll see later.

To illustrate, let's focus on the bottom part of figure 3.2, which tracks how a Not gate (used as an example) responds to arbitrarily chosen inputs. When we feed the gate with 1, it takes a while before the gate's output stabilizes on 0. However, since the cycle duration is—*by design*—longer than the time delay, when we reach the cycle's end, the gate output has already stabilized on 0. Since we probe the state of the world only at cycle ends, we don't get to see the interim time delays; rather, it appears as if we fed the gate with 0, and *poof!* the gate responded with 1. If we make the same observations at the end of each cycle, we can generalize that when a Not gate is fed with some binary input *x*, it instantaneously outputs Not (*x*).

Thoughtful readers have probably noticed that for this scheme to work, the *cycle's length* must be longer than the maximal time delays that can occur in the system. Indeed, cycle length is one of the most important design parameters of any hardware platform: When planning a computer, the hardware engineer chooses a cycle length that meets two design objectives. On the one hand, the cycle should be sufficiently long to contain, and neutralize, any possible time delay; on the other hand, the shorter the cycle, the faster the computer: if things happen only during cycle transitions, then obviously things happen faster when the cycles are shorter. To sum up, the cycle length is chosen to be slightly longer than the maximal time delay in any chip in the system. Following the tremendous progress in switching technologies, we are now able to create cycles as tiny as a billionth of a second, achieving remarkable computer speed.

Typically, the cycles are realized by an oscillator that alternates continuously between two phases labeled 0−1, *low-high*, or *ticktock* (as seen in figure 3.2). The elapsed time between the beginning of a tick and the end of the subsequent tock is called a *cycle*, and each 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, the same master clock signal is simultaneously broadcast to every memory chip in the system. In every such chip, the clock input is funneled to the lower-level DFF gates, where it serves to ensure that the chip will commit to a new state, and output it, only at the end of the clock cycle.

3.2.2 Flip-Flops

Memory chips are designed to "remember", or store, information over time. The low-level devices that facilitate this storage abstraction are named *flip-flop* gates, of which there are several variants. In Nand to Tetris we use a variant named *data flip-flop*, or DFF, whose interface includes a single-bit data input and a single-bit data output (see top of figure 3.3). In addition, the DFF has a clock input that feeds from the master clock's signal. Taken together, the data input and the clock input enable the DFF to implement the simple time-based behavior $\text{out}(t) = \text{in}(t-1)$, where in and out are the gate's input and output values, and t is the current time unit (from now on, we'll use the terms "time unit" and "cycle" interchangeably). Let us not worry how this behavior is actually implemented. For now, we simply observe that at the end of each time unit, the DFF outputs the input value from the previous time unit.

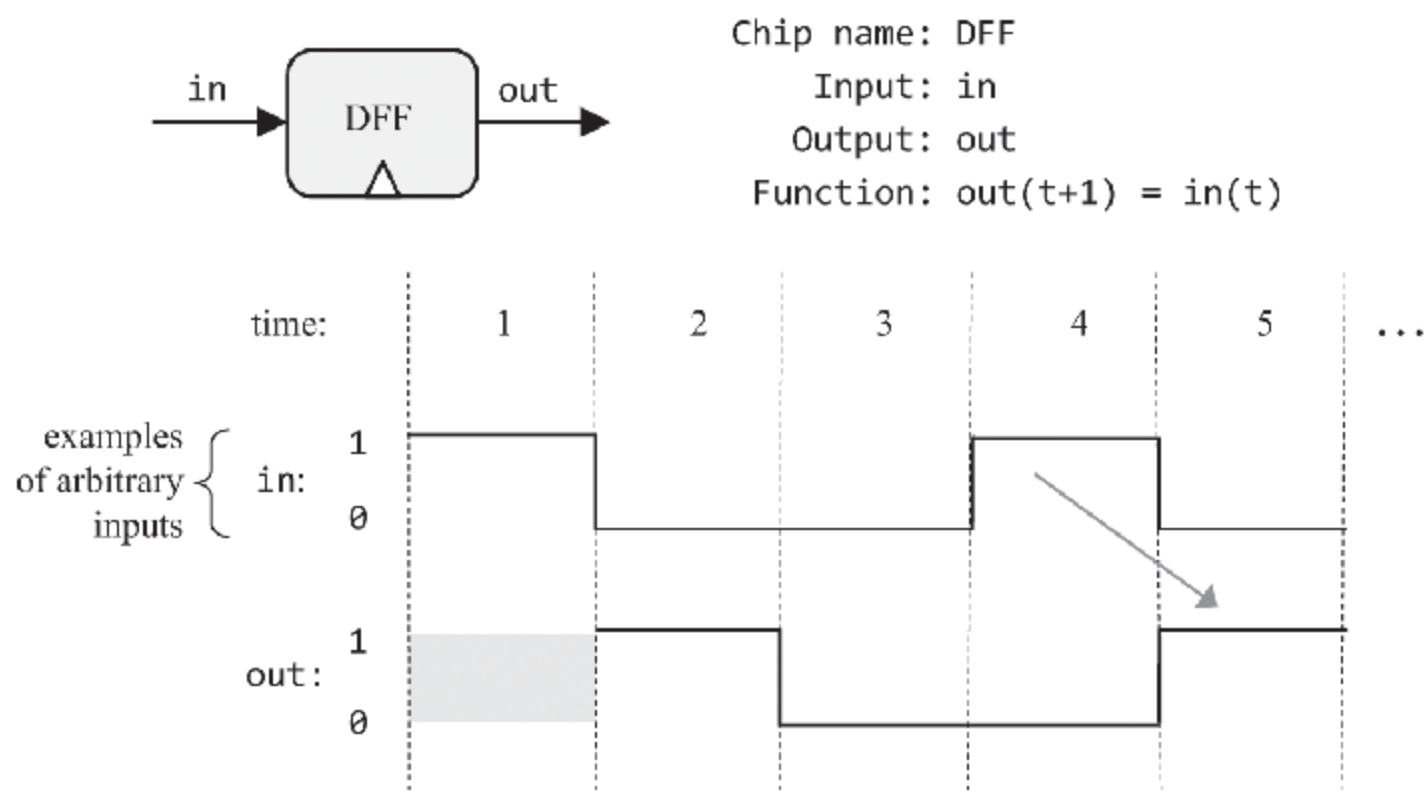


Figure 3.3 The data flip-flop (top) and behavioral example (bottom). In the first cycle the previous input is unknown, so the DFF's output is undefined. In every subsequent time unit, the DFF outputs the input from the previous time unit. Following gate diagramming conventions, the clock input is marked by a small triangle, drawn at the bottom of the gate icon.

Like Nand gates, DFF gates lie deep in the hardware hierarchy. As shown in figure 3.1, all the memory chips in the computer—registers, RAM units, and counters—are based, at bottom, on DFF gates. All these DFFs are connected to the same master clock, forming a huge distributed "chorus

line." At the end of each clock cycle, the outputs of *all* the DFFs in the computer commit to their inputs from the previous cycle. 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 system's numerous DFF gates many times per second (depending on the computer's clock frequency).

Hardware implementations realize the time dependency using a dedicated clock bus that feeds the master clock signal simultaneously to all the DFF gates in the system. Hardware simulators emulate the same effect in software. In particular, the Nand to Tetris hardware simulator features a clock icon, enabling the user to advance the clock interactively, as well as `tick` and `tock` commands that can be used programmatically, in test scripts.

3.2.3 Combinational and Sequential Logic

All the chips that were developed in chapters 1 and 2, starting with the elementary logic gates and culminating with the ALU, were designed to respond only to changes that occur during the current clock cycle. Such chips are called *time-independent* chips, or *combinational* chips. The latter name alludes to the fact that these chips respond only to different combinations of their input values, while paying no attention to the progression of time.

In contrast, chips that are designed to respond to changes that occurred during previous time units (and possibly during the current time unit as well) are called *sequential*, or *clocked*. The most fundamental sequential gate is the DFF, and any chip that includes it, either directly or indirectly, is also said to be sequential. Figure 3.4 depicts a typical sequential logic configuration. The main element in this configuration is a set of one or more chips that include DFF chip-parts, either directly or indirectly. As shown in the figure, these sequential chips may also interact with combinational chips. The feedback loop enables the sequential chip to respond to inputs and outputs from the previous time unit. In combinational chips, where time is neither modeled nor recognized, the introduction of feedback loops is problematic: the output of the chip would depend on its input, which itself would depend on the output, and thus the output would depend on itself. Note, however, that there is no difficulty in feeding

outputs back into inputs, as long as the feedback loop goes through a DFF gate: the DFF introduces an inherent time delay so that the output at time t does not depend on itself but rather on the output at time $t-1$.

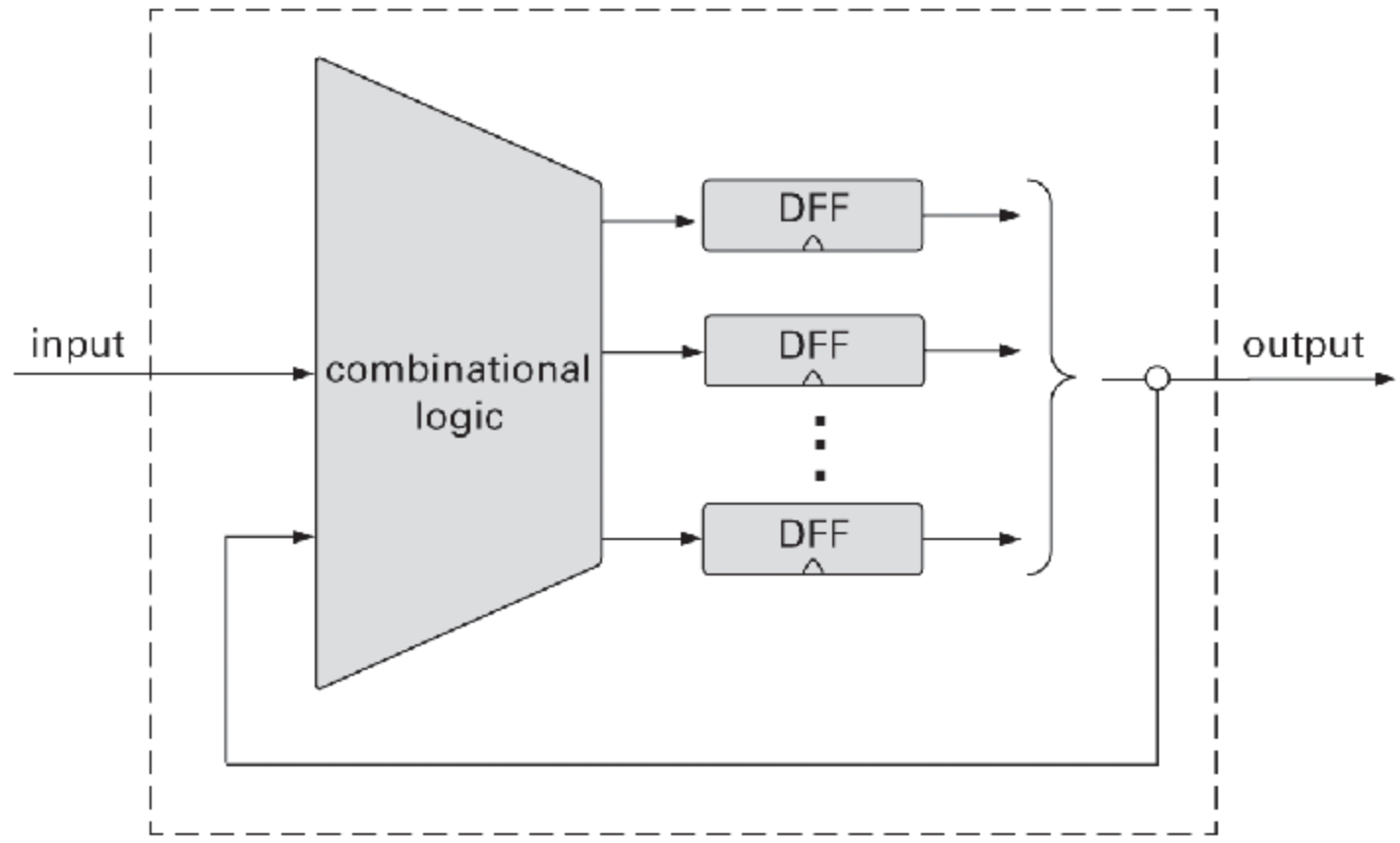


Figure 3.4 Sequential logic design typically involves DFF gates that feed from, and connect to, combinational chips. This gives sequential chips the ability to respond to current as well as to previous inputs and outputs.

The time dependency of sequential chips has an important side effect that serves to synchronize the overall computer architecture. To illustrate, suppose we instruct the ALU to compute $x+y$, where x is the output of a nearby register, and y is the output of a remote RAM register. Because of physical constraints like distance, resistance, and interference, 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 and happily adds up whichever data values happen to lodge at 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, if we use a discrete representation of time, *we simply don't care*. All we have to do is ensure, when we build the computer's clock, that the duration of the clock cycle will be slightly longer than the time it takes a bit to travel the longest distance from one chip to another, plus the time it takes to complete the

most time-consuming within-chip calculation. This way, we are guaranteed that by the end of the clock cycle, the ALU's output will be valid. This, in a nutshell, is the trick that turns a set of standalone hardware components into a well-synchronized system. We will have more to say about this master orchestration when we build the computer architecture in chapter 5.

3.3 Specification

We now turn to specify the memory chips that are typically used in computer architectures:

- data flip-flops (DFFs)
- registers (based on DFFs)
- RAM devices (based on registers)
- counters (based on registers)

As usual, we describe these chips *abstractly*. In particular, we focus on each chip's interface: inputs, outputs, and function. How the chips deliver this functionality will be discussed in the Implementation section.

3.3.1 Data Flip-Flop

The most elementary sequential device that we will use—the basic component from which all other memory chips will be constructed—is the *data flip-flop*. A DFF gate has a single-bit data input, a single-bit data output, a clock input, and a simple time-dependent behavior: $\mathtt{out}(t) = \mathtt{in}(t-1)$.

Usage: If we put a one-bit value in the DFF's input, the DFF's state will be set to this value, and the DFF's output will emit it in the next time unit (see figure 3.3). This humble operation will prove most useful in the implementation of registers, which is described next.

3.3.2 Registers

We present a single-bit register, named Bit, and a 16-bit register, named Register. The Bit chip is designed to store a single bit of information—0 or 1

—over time. The chip interface consists of an in input that carries a data bit, a load input that enables the register for writes, and an out output that emits the current state of the register. The Bit API and input/output behavior are described in figure 3.5.

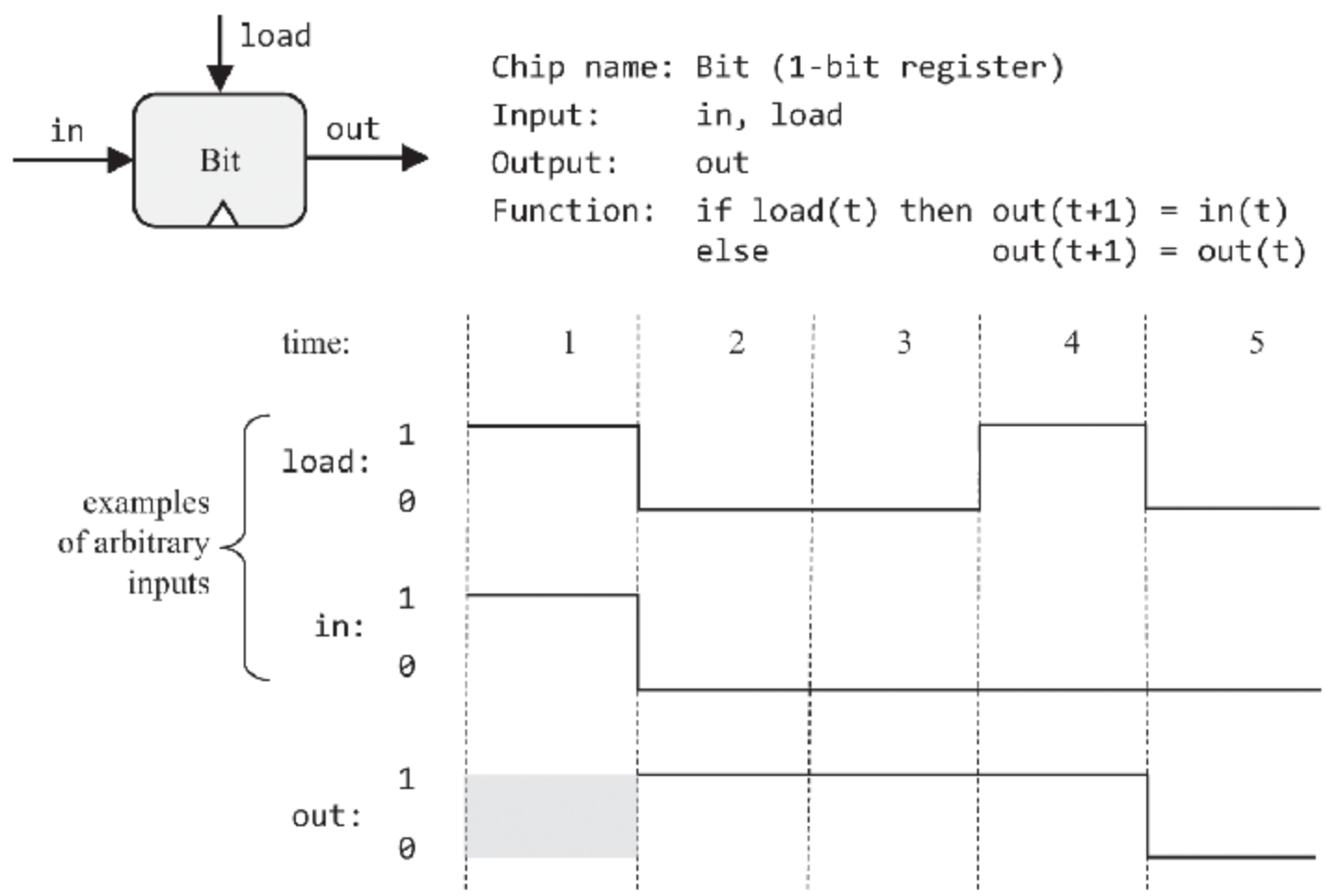


Figure 3.5 1-bit register. Stores and emits a 1-bit value until instructed to load a new value.

Figure 3.5 illustrates how the single-bit register behaves over time, responding to arbitrary examples of in and load inputs. Note that irrespective of the input value, as long as the load bit is not asserted, the register is latched, maintaining its current state.

The 16-bit Register chip behaves exactly the same as the Bit chip, except that it is designed to handle 16-bit values. Figure 3.6 gives the details.

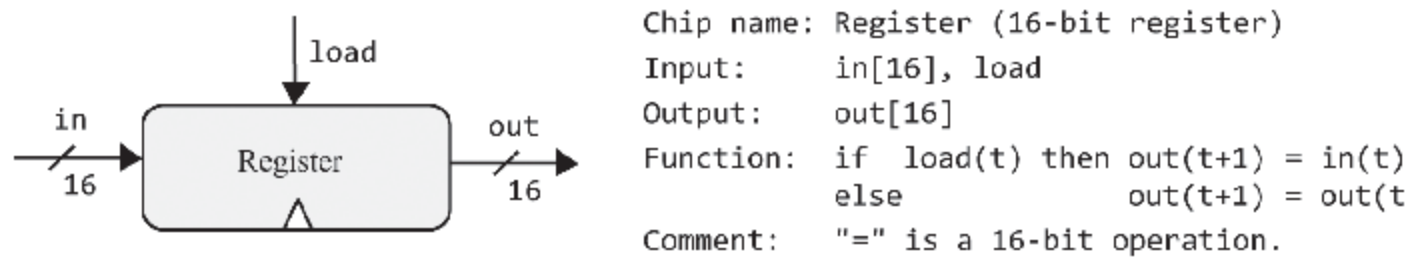
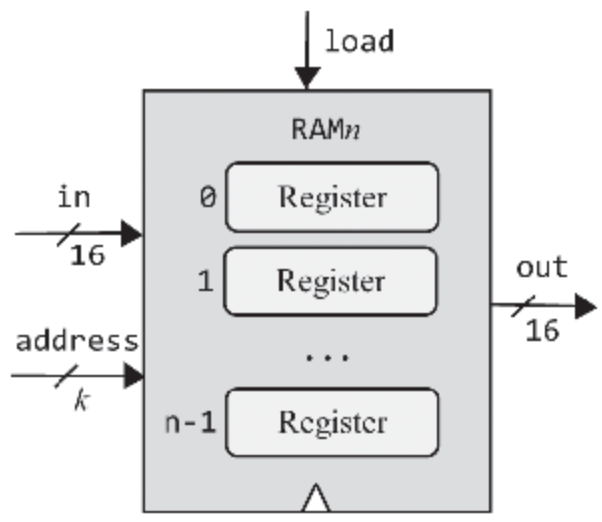


Figure 3.6 16-bit Register. Stores and emits a 16-bit value until instructed to load a new value.

Usage: The Bit register and the 16-bit Register are used identically. To read the state of the register, probe the value of out. To set the register's state to *v*, put *v* in the in input, and assert (put 1 into) the load bit. This will set the register's state to *v*, and, from the next time unit onward, the register will commit to the new value, and its out output will start emitting it. We see that the Register chip fulfills the classical function of a memory device: it remembers and emits the last value that was written into it, until we set it to another value.

3.3.3 Random Access Memory

A direct-access memory unit, also called *Random Access Memory*, or RAM, is an aggregate of n Register chips. By specifying a particular address (a number between 0 to $n-1$), each register in the RAM can be selected and made available for read/write operations. Importantly, the access time to any randomly selected memory register is instantaneous and independent of the register's address and the size of the RAM. That's what makes RAM devices so remarkably useful: even if they contain billions of registers, we can still access and manipulate each selected register directly, in the same instantaneous access time. The RAM API is given in figure 3.7.



Chip name: RAM*n*
Input: in[16], load, address[*k*] (*k* = $\log_2 n$)
Output: out[16]
Function:
Out emits the value stored at the memory location (register) specified by address. If load==1, then the memory location specified by address is set to the value of in. The loaded value will be emitted by out from the next time step onward.

Figure 3.7 A RAM chip, consisting of n 16-bit Register chips that can be selected and manipulated separately. The register addresses (0 to $n-1$) are not part of the chip hardware. Rather, they are realized by a gate logic implementation that will be discussed in the next section.

Usage: To read the contents of register number *m*, set the address input to *m*. This action will select register number *m*, and the RAM's output will emit its value. To write a new value *v* into register number *m*, set the address

input to *m*, set the in input to *v*, and assert the load bit (set it to 1). This action will select register number *m*, enable it for writing, and set its value to *v*. In the next time unit, the RAM's output will start emitting *v*.

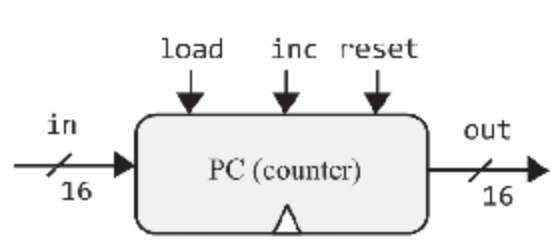
The net result is that the RAM device behaves exactly as required: a bank of addressable registers, each of which can be accessed, and operated upon, independently. In the case of a read operation (load==0), the RAM's output immediately emits the value of the selected register. In the case of a write operation (load==1), the selected memory register is set to the input value, and the RAM's output will start emitting it from the next time unit onward.

Importantly, the RAM implementation must ensure that the access time to *any* register in the RAM will be nearly instantaneous. If this were not the case, we would not be able to fetch instructions and manipulate variables in a reasonable time, making computers impractically slow. The magic of instantaneous access time will be unfolded shortly, in the Implementation section.

3.3.4 Counter

The Counter is a chip that knows how to increment its value by 1 each time unit. When we build our computer architecture in chapter 5, we will call this chip Program Counter, or PC, so that's the name that we will use here also.

The interface of our PC chip is identical to that of a register, except that it also has control bits labeled inc and reset. When inc==1, the counter increments its state in every clock cycle, effecting the operation PC++. If we want to reset the counter to 0, we assert the reset bit; if we want to set the counter to the value *v*, we put *v* in the in input and assert the load bit, as we normally do with registers. The details are given in figure 3.8.




```
Chip name: PC
Input:     in[16], load, inc, reset
Output:    out[16]
Function:  16-bit counter
if reset(t)      out(t+1) = 0
else if load(t)  out(t+1) = in(t)
else if inc(t)   out(t+1) = out(t) + 1 (integer addition)
else             out(t+1) = out(t)
```

Figure 3.8 Program Counter (PC): To use it properly, at most one of the load, inc, or reset bits should be asserted.

Usage: To read the current contents of the PC, probe the out pin. To reset the PC, assert the reset bit and set the other control bits to 0. To have the PC increment by 1 in each time unit until further notice, assert the inc bit and set the other control bits to 0. To set the PC to the value *v*, set the in input to *v*, assert the load bit, and set the other control bits to 0.

3.4 Implementation

The previous section presented a family of memory chip abstractions, focusing on their interface and functionality. This section focuses on how these chips can be realized, using simpler chips that were already built. As usual, our implementation guidelines are intentionally suggestive; we want to give you enough hints to complete the implementation yourself, using HDL and the supplied hardware simulator.

3.4.1 Data Flip-Flop

A DFF gate is designed to be able to "flip-flop" between two stable states, representing 0 and representing 1. This functionality can be implemented in several different ways, including ones that use Nand gates only. The Nand-based DFF implementations are elegant, yet intricate and impossible to model in our hardware simulator since they require feedback loops among combinational gates. Wishing to abstract away this complexity, we will treat the DFF as a primitive building block. In particular, the Nand to Tetris hardware simulator provides a built-in DFF implementation that can be readily used by other chips, as we now turn to describe.

3.4.2 Registers

Register chips are memory devices: they are expected to implement the basic behavior $\texttt{out}(t+1) = \texttt{out}(t)$, remembering and emitting their state over time. This looks similar to the DFF behavior, which is $\texttt{out}(t+1) = \texttt{in}(t)$. If we could only feed the DFF output back into its input, this could be a good starting point for implementing the one-bit Bit register. This solution is shown on the left of figure 3.9.

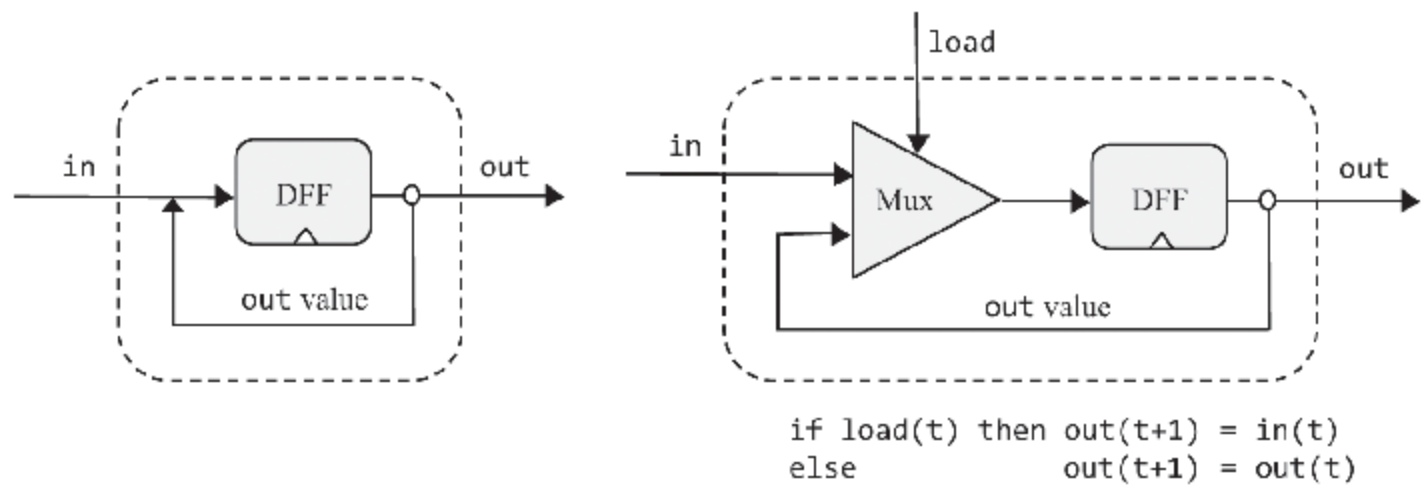


Figure 3.9 The Bit (1-bit register) implementation: invalid (left) and correct (right) solutions.

It turns out that the implementation shown on the left of figure 3.9 is invalid, for several related reasons. First, the implementation does not expose a load bit, as required by the register's interface. Second, there is no way to tell the DFF chip-part when to draw its input from the in wire and when from the incoming out value. Indeed, HDL programming rules forbid feeding a pin from more than one source.

The good thing about this invalid design is that it leads us to the correct implementation, shown on the right of figure 3.9. As the chip diagram shows, a natural way to resolve the input ambiguity is introducing a multiplexer into the design. The *load bit* of the overall register chip can then be funneled to the *select bit* of the inner multiplexer: If we set this bit to 1, the multiplexer will feed the in value into the DFF; if we set the load bit to 0, the multiplexer will feed the DFF's previous output. This will yield the behavior "if load, set the register to a new value, else set it to the previously stored value"—exactly how we want a register to behave.

Note that the feedback loop just described does not entail cyclical *data race* problems: the loop goes through a DFF gate, which introduces a time delay. In fact, the Bit design shown in figure 3.9 is a special case of the general sequential logic design shown in figure 3.4.

Once we've completed the implementation of the single-bit Bit register, we can move on to constructing a *w*-bit register. This can be achieved by forming an array of *w* Bit chips (see figure 3.1). The basic design parameter of such a register is *w*—the number of bits that it is supposed to hold—for example, 16, 32, or 64. Since the Hack computer will be based on a 16-bit hardware platform, our Register chip will be based on sixteen Bit chip-parts.

The Bit register is the only chip in the Hack architecture that uses a DFF gate directly; all the higher-level memory devices in the computer use DFF chips indirectly, by virtue of using Register chips made of Bit chips. Note that the inclusion of a DFF gate in the design of any chip—directly or indirectly—turns the latter chip, as well as all the higher-level chips that use it as a chip-part, into time-dependent chips.

3.4.3 RAM

The Hack hardware platform requires a RAM device of 16K (16384) 16-bit registers, so that's what we have to implement. We propose the following gradual implementation roadmap:

chip	n	k	built from:
RAM8	8	3	8 Register chips
RAM64	64	6	8 RAM8 chips
RAM512	512	9	8 RAM64 chips
RAM4K	4096	12	8 RAM512 chips
RAM16K	16384	14	4 RAM4K chips

All these memory chips have precisely the same RAMn API given in figure 3.7. Each RAM chip has n registers, and the width of its address input is $k = log_2 n$ bits. We now describe how these chips can be implemented, starting with RAM8.

A RAM8 chip features 8 registers, as shown in figure 3.7, for $n = 8$. Each register can be selected by setting the RAM8's 3-bit address input to a value between 0 and 7. The act of *reading* the value of a selected register can be described as follows: Given some address (a value between 0 and 7), how can we "select" register number address and pipe its output to the RAM8's output? *Hint:* We can do it using one of the combinational chips built in project 1. That's why reading the value of a selected RAM register is achieved nearly instantaneously, independent of the clock and of the number of registers in the RAM. In a similar way, the act of *writing* a value into a selected register can be described as follows: Given an address value, a load value (1), and a 16-bit in value, how can we set the value of register number address to in? *Hint:* The 16-bit in data can be fed simultaneously to

the in inputs of all eight Register chips. Using another combinational chip developed in project 1, along with the address and load inputs, you can ensure that only one of the registers will accept the incoming in value, while all the other seven registers will ignore it.

We note in passing that the RAM registers are not marked with addresses in any physical sense. Rather, the logic described above is capable of, and sufficient for, selecting individual registers according to their address, and this is done by virtue of using combinational chips. Now here is a crucially important observation: since combinational logic is time independent, the access time to any individual register will be nearly instantaneous.

Once we've implemented the RAM8 chip, we can move on to implementing a RAM64 chip. The implementation can be based on eight RAM8 chip-parts. To select a particular register from the RAM64 memory, we use a 6-bit address, say *xxxyyy*. The *xxx* bits can be used to select one of the RAM8 chips, and the *yyy* bits can be used to select one of the registers within the selected RAM8. This hierarchical addressing scheme can be effected by gate logic. The same implementation idea can guide the implementation of the remaining RAM512, RAM4K, and RAM16K chips.

To recap, we take an aggregate of an unlimited number of registers, and impose on it a combinational superstructure that permits direct access to any individual register. We hope that the beauty of this solution does not escape the reader's attention.

3.4.4 Counter

A counter is a memory device that can increment its value in every time unit. In addition, the counter can be set to 0 or some other value. The basic storage and counting functionalities of the counter can be implemented, respectively, by a Register chip and by the incrementer chip built in project 2. The logic that selects between the counter's inc, load, and reset modes can be implemented using some of the multiplexers built in project 1.

3.5 Project

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

Resources: The only tool that you need for this project is the Nand to Tetris hardware simulator. All the chips should be implemented in the HDL language specified in appendix 2. 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 that defines the expected results. 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; thus there is no need to build it. When the simulator encounters a DFF chip-part in an HDL program, it automatically invokes the tools/builtIn/DFF.hdl implementation.

Folders structure of this project: When constructing RAM chips from lower-level RAM chip-parts, we recommend using built-in versions of the latter. Otherwise, the simulator will recursively generate numerous memory-resident software objects, one for each of the many chip-parts that make up a typical RAM unit. This may cause the simulator to run slowly or, worse, run out of the memory of the host computer on which the simulator is running.

To avert this problem, we've partitioned the RAM chips built in this project into two subfolders. The RAM8.hdl and RAM64.hdl programs are stored in projects/03/a, and the other, higher-level RAM chips are stored in projects/03/b. This partitioning is done for one purpose only: when evaluating the RAM chips stored in the b folder, the simulator will be forced to use built-in implementations of the RAM64 chip-parts, because RAM64.hdl cannot be found in the b folder.

Steps: We recommend proceeding in the following order:

1. The hardware simulator needed for this project is available in nand2tetris/tools.
2. Consult appendix 2 and the Hardware Simulator Tutorial, as needed.
3. Build and simulate all the chips specified in the projects/03 folder.

A web-based version of project 3 is available at www.nand2tetris.org.

3.6 Perspective

The cornerstone of all the memory systems described in this chapter is the flip-flop, which we treated abstractly as a primitive, built-in gate. The usual approach is to construct flip-flops from elementary combinational gates (e.g., Nand gates) connected in feedback loops. The standard construction begins by building a non-clocked flip-flop which is bi-stable, that is, can be set to be in one of two states (storing 0, and storing 1). Then a clocked flip-flop is obtained by cascading two such non-clocked 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.

Such flip-flop implementations are both elegant and intricate. In this book we have chosen to abstract away these low-level considerations by treating the flip-flop as a primitive 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.

One reason not to dwell on flip-flop esoterica is that the lowest level of the memory devices used in modern computers is not necessarily constructed from flip-flop gates. Instead, modern memory chips are 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. Likewise, the recursive ascent method that we used to build the RAM chips is elegant but not necessarily efficient. More efficient implementations are possible.

Aside from these physical considerations, all the chip constructions described in this chapter—the registers, the counter, and the RAM chips—are standard, and versions of them can be found in every computer system.

In chapter 5, we will use the register chips built in this chapter, along with the ALU built in chapter 2, to build a Central Processing Unit. The CPU will then be augmented with a RAM device, leading up to a general-purpose computer architecture capable of executing programs written in a machine language. This machine language is discussed in the next chapter.

4 Machine Language

Works of imagination should be written in very plain language; the more purely imaginative they are, the more necessary it is to be plain.

—Samuel Taylor Coleridge (1772–1834)

In chapters 1–3 we built processing and memory chips that can be integrated into the hardware platform of a general-purpose computer. Before we set out to complete this construction, let's pause and ask: What exactly is the *purpose* of this computer? As the architect Louis Sullivan famously observed, "Form follows function." If you wish to understand a system, or build one, start by studying the function that the system is supposed to serve. With that in mind, before we set out to complete the construction of our hardware platform, we'll devote this chapter to studying the *machine language* that this platform is designed to realize. After all, executing programs written in machine language efficiently is the ultimate function of any general-purpose computer.

A machine language is an agreed-upon formalism designed to code machine instructions. Using these instructions, we can instruct the computer's processor to perform arithmetic and logical operations, read and write values from and to the computer's memory, test Boolean conditions, and decide which instruction to fetch and execute next. Unlike high-level languages, whose design goals are cross-platform compatibility and power of expression, machine languages are designed to effect direct execution in, and total control of, a specific hardware platform. Of course, generality, elegance, and power of expression are still desired, but only to the extent that they support the basic requirement of direct and efficient execution in hardware.

Machine language is the most profound interface in the computer enterprise—the fine line where hardware meets software. This is the point where the abstract designs of humans, as manifested in high-level programs, are finally reduced to physical operations performed in silicon. Thus, a machine language can be construed as both a programming artifact and an integral part of the hardware platform. In fact, just as we say that the machine language is designed to control a particular hardware platform, we can say that the hardware platform is designed to execute instructions written in a particular machine language.

The chapter begins with a general introduction to low-level programming in machine language. Next, we give a detailed specification of the *Hack machine language*, covering both its binary and symbolic versions. The project that ends the chapter focuses on writing some machine language programs. This provides a hands-on appreciation of low-level programming and sets the stage for completing the construction of the computer hardware in the next chapter.

Although programmers rarely write programs directly in machine language, the study of low-level programming is a prerequisite to a complete and rigorous understanding of how computers work. Further, an intimate understanding of low-level programming helps the programmer write better and more efficient high-level programs. Finally, it is rather fascinating to observe, hands-on, how the most sophisticated software systems are, at bottom, streams of simple instructions, each specifying a primitive bitwise operation on the underlying hardware.

4.1 Machine Language: Overview

This chapter focuses not on the machine but rather on the *language* used to control the machine. Therefore, we will abstract away the hardware platform, focusing on the minimal subset of hardware elements that are mentioned explicitly in machine language instructions.

4.1.1 Hardware Elements

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. Functionally speaking, a memory is a continuous sequence of cells, also referred to as *locations* or *memory registers*, each having a unique *address*. An individual memory register is accessed by supplying its address.

Processor: The processor, normally called the *Central Processing Unit*, or *CPU*, is a device capable of performing a fixed set of primitive operations. These include arithmetic and logical operations, memory access operations, and control (also called *branching*) operations. The processor draws its inputs from selected registers and memory locations and writes its outputs to selected registers and memory locations. It consists of an ALU, a set of registers, and gate logic that enables it to parse and execute binary instructions.

Registers: The processor and the memory are implemented as two separate, standalone chips, and moving data from one to the other is a relatively slow affair. For this reason, processors are normally equipped with several on-board registers, each capable of holding a single value. Located inside the processor's chip, the registers serve as a high-speed local memory, allowing the processor to manipulate data and instructions without having to venture outside the chip.

The CPU-resident registers fall into two categories: *data registers*, designed to hold data values, and *address registers*, designed to hold values that can be interpreted either as data values or as memory addresses. The computer architecture is configured in such a way that placing a particular value, say *n*, in an address register, causes the memory location whose address is *n* to become *selected* instantaneously.[1] This sets the stage for a subsequent operation on the selected memory location.

4.1.2 Languages

Machine language programs can be written in two alternative, but equivalent, ways: *binary* and *symbolic*. For example, consider the abstract operation "set R1 to the value of R1+R2". As language designers, we can decide to represent the addition operation using the 6-bit code 101011, and registers R1 and R2 using the codes 00001 and 00010, respectively. Assembling these codes left to right, we can decide to use the 16-bit instruction 1010110001000001 as the binary version of "set R1 to the value of R1+R2".

In the early days of computer systems, computers were programmed manually: When proto-programmers wanted to issue the instruction "set R1 to the value of R1+R2", they pushed up and down mechanical switches that stored a binary code like 1010110001000001 in the computer's instruction memory. And if the program was a hundred instructions long, they had to go through this ordeal a hundred times. Of course debugging such programs was a perfect nightmare. This led programmers to invent and use symbolic codes as a convenient way for documenting and debugging programs *on paper*, before entering them into the computer. For example, the symbolic format add R2,R1 could be chosen for representing the semantics "set R1 to the value of R1+R2" and the binary instruction 1010110001000001.

It didn't take long before several people hit on the same idea: Symbols like R, 1, 2, and + can also be represented using agreed-upon binary codes. Why not use symbolic instructions for writing programs, and then use another program—a *translator*—for translating the symbolic instructions into executable binary code? This innovation liberated programmers from the tedium of writing binary code, paving the way for the subsequent onslaught of high-level programming languages. For reasons that will become clear in chapter 6, symbolic machine languages are called *assembly languages*, and the programs that translate them into binary code are called *assemblers*.

Unlike the syntax of high-level languages, which is portable and hardware independent, the syntax of an assembly language is tightly related to the low-level details of the target hardware: the available ALU operations, number and type of registers, memory size, and so on. Since different computers vary greatly in terms of any one of these parameters, there is a Tower of Babel of machine languages, each with its obscure syntax, each designed to control a particular family of CPUs. Irrespective of

this variety, though, all machine languages are theoretically equivalent, and all of them support similar sets of generic tasks, as we now turn to describe.

4.1.3 Instructions

In what follows, we assume that the computer's processor is equipped with a set of registers denoted R0, R1, R2, … The exact number and type of these registers are irrelevant to our present discussion.

Arithmetic and logical operations: Every machine language features instructions for performing basic arithmetic operations like addition and subtraction, as well as basic logical operations like And, Or, Not. For example, consider the following code segments:

```
// Adds up two numbers:
load R1,17      // R1 ← 17
load R2,4       // R2 ← 4
add  R1,R1,R2   // R1 ← R1 + R2
```

```
// Computes a logical operation:
load R1,true    // R1 ← binary representation of true
load R2,false   // R2 ← binary representation of false
and  R1,R1,R2   // R1 ← R1 And R2 (bit-wise And)
```

For such symbolic instructions to execute on a computer, they must first be translated into binary code. The translation is done by a program named *assembler*, which we'll develop in chapter 6. For now, we assume that we have access to such an assembler and that we can use it as needed.

Memory access: Every machine language features means for accessing, and then manipulating, selected memory locations. This is typically done using an *address register*, let's call it A. For example, suppose we wish to set memory location 17 to the value 1. We can decide to do so using the two instructions load A,17 followed by load M,1, where, by convention, M stands for the memory register selected by A (namely, the memory register whose address is the current value of A). With that in mind, suppose we wish to set the fifty memory locations 200, 201, 202, …, 249 to 1. This can be done by