

Boolean Arithmetic

Counting is the religion of this generation, its hope and salvation.

—Gertrude Stein (1874-1946)

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

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

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

2.1 Background

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

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

(1)

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

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

(2)

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

The result of calculation (1) happens to be 19. Thus, when we press the keyboard keys labeled ‘1’, ‘9’ and ENTER while running, say, a spreadsheet program, what ends up in some register in the computer’s memory is the binary code 10011. More precisely, if the computer happens to be a 32-bit machine, what gets stored in the register is the bit pattern 000000000000000000000000000010011.

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

$ \begin{array}{r} 0\ 0\ 0\ 1 \\ 1\ 0\ 0\ 1 \\ + 0\ 1\ 0\ 1 \\ \hline 0\ 1\ 1\ 1\ 0 \end{array} $	(carry) x y $x + y$	$ \begin{array}{r} 1\ 1\ 1\ 1 \\ 1\ 0\ 1\ 1 \\ + 0\ 1\ 1\ 1 \\ \hline 1\ 0\ 0\ 1\ 0 \end{array} $
no overflow		overflow

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

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

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

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

For example, in a 5-bit binary system, the 2's complement representation of -2 or “minus $(00010)_{two}$ ” is $2^5 - (00010)_{two} = (32)_{ten} - (2)_{ten} = (30)_{ten} = (11110)_{two}$. To check the calculation, the reader can verify that $(00010)_{two} + (11110)_{two} = (00000)_{two}$. Note that in the latter computation, the sum is actually $(100000)_{two}$, but since we are dealing with a 5-bit binary system, the left-most sixth bit is simply ignored.

As a rule, when the 2's complement method is applied to n -bit numbers, $x + (-x)$ always sums up to 2^n (i.e., 1 followed by n 0's)—a property that gives the method its name. Figure 2.1 illustrates a 4-bit binary system with the 2's complement method.

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

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

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

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

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

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

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

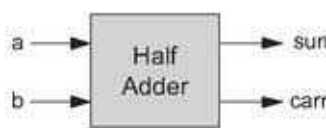
2.2 Specification

2.2.1 Adders

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

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

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



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

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

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

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

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

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

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

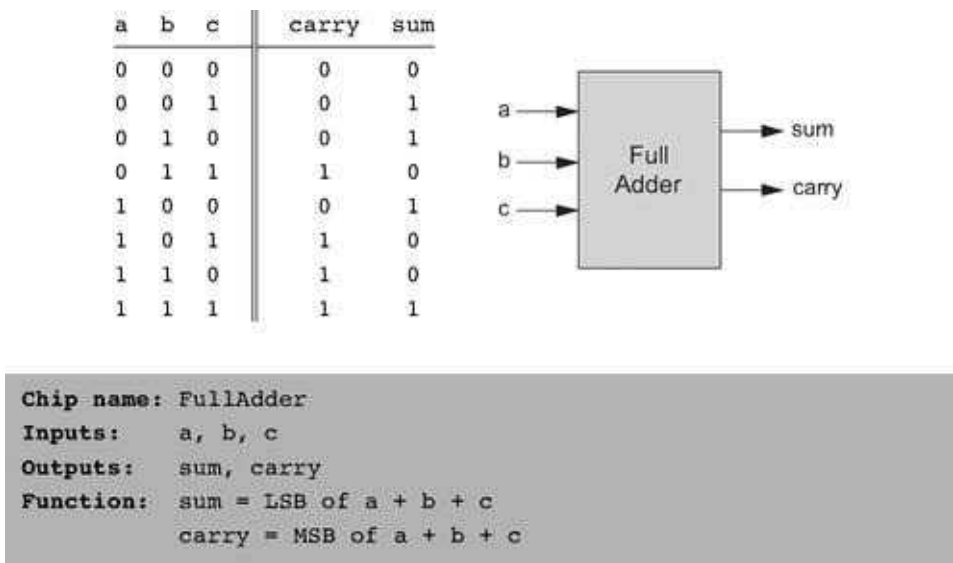


Figure 2.3 Full-adder, designed to add 3 bits.

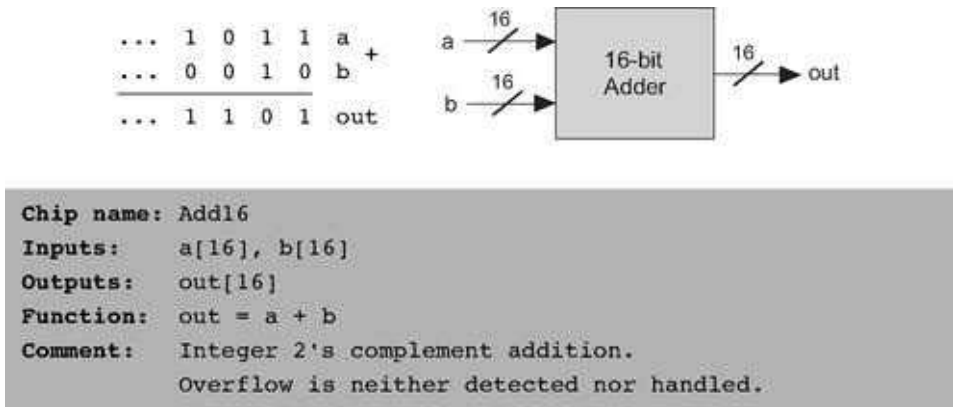
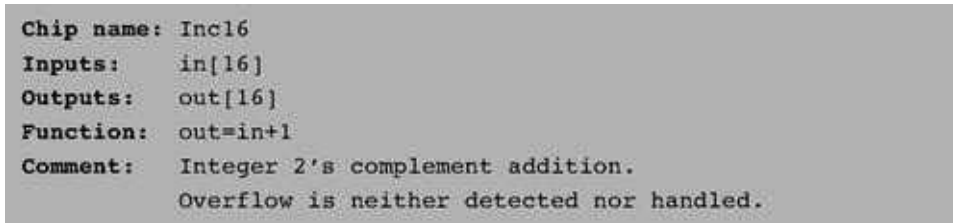


Figure 2.4 16-bit adder. Addition of two n -bit binary numbers for any n is “more of the same.”



2.2.2 The Arithmetic Logic Unit (ALU)

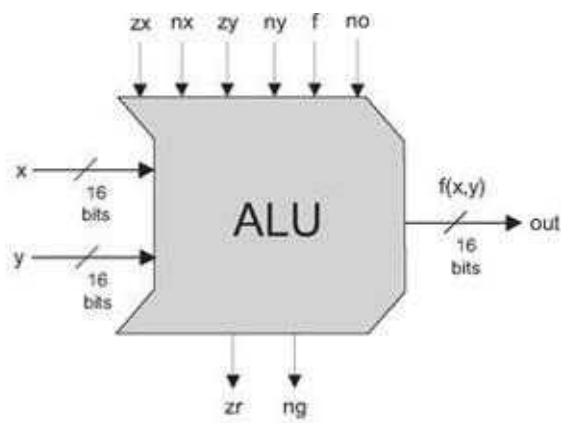
The specifications of the adder chips presented so far were generic, meaning that they hold for any computer. In contrast, this section describes an ALU that will later become the centerpiece of a specific computer platform called Hack. At the same time, the principles underlying the design of our ALU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

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

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

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

For example, let us consider the twelfth row of figure 2.6, where the ALU is instructed to compute the function $x-1$. The zx and nx bits are 0, so the x input is neither zeroed nor negated. The zy and ny bits are 1, so the y input is first zeroed, and then negated bit-wise. Bit-wise negation of zero, $(000 \dots 00)_{\text{two}}$, gives $(111 \dots 11)_{\text{two}}$, the 2's complement code of -1. Thus the ALU inputs end up being x and -1. Since the f -bit is 1, the selected operation is arithmetic addition, causing the ALU to calculate $x + (-1)$. Finally, since the no bit is 0, the output is not negated but rather left as is. To conclude, the ALU ends up computing $x-1$, which was our goal.



```

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

```

Figure 2.5 The Arithmetic Logic Unit.

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

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

Does the ALU logic described in figure 2.6 compute every one of the other seventeen functions listed in the figure's right column? To verify that this is indeed the case, the reader can pick up some other rows in the table and prove their respective ALU operation. We note that some of these computations, beginning with the function $f(x, y) = 1$, are not trivial. We also note that there are some other useful functions computed by the ALU but not listed in the figure.

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

2.3 Implementation

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

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

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

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

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

2.4 Perspective

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

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

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

2.5 Project

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

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

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

Sequential Logic

It's a poor sort of memory that only works backward.

—Lewis Carroll (1832-1898)

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

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

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

3.1 Background

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

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

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

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

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

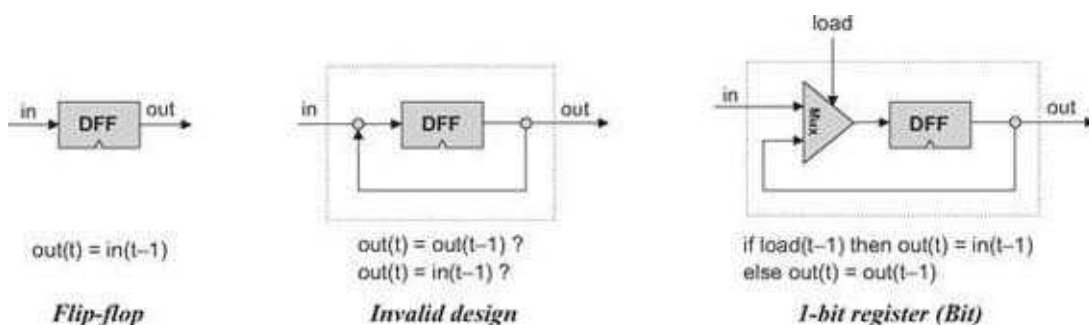


Figure 3.1 From a DFF to a single-bit register. The small triangle represents the clock input. This icon is used to state that the marked chip, as well as the overall chip that encapsulates it, is time-dependent.

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

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

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

Memories Once we have the basic ability to represent words, we can proceed to build memory banks of arbitrary length. As figure 3.3 shows, this can be done by stacking together many registers to form a Random Access Memory RAM unit. The term random access memory derives from the requirement that read/write operations on a RAM should be able to access randomly chosen words, with no restrictions on the order in which they are accessed. That is to say, we require that any word in the memory—irrespective of its physical location—be accessed directly, in equal speed.

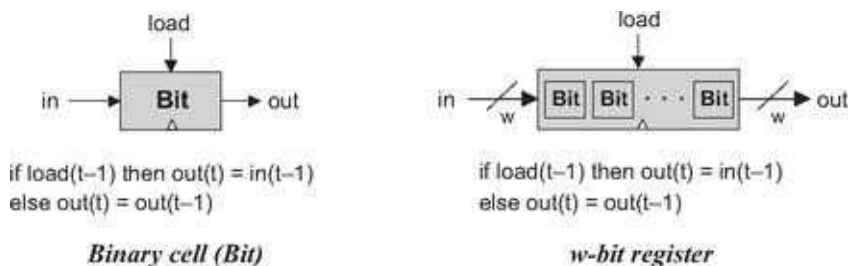


Figure 3.2 From single-bit to multi-bit registers. A multi-bit register of width w can be constructed from an array of w 1-bit chips. The operating functions of both chips is exactly the same, except that the “=” assignments are single-bit and multi-bit, respectively.

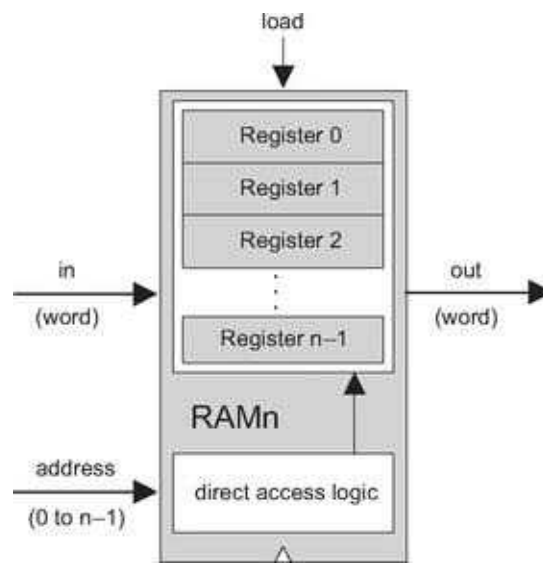


Figure 3.3 RAM chip (conceptual). The width and length of the RAM can vary.

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

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

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

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

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

Time Matters All the chips described so far in this chapter are sequential. Simply stated, a sequential chip is a chip that embeds one or more DFF gates, either directly or indirectly. Functionally speaking, the

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

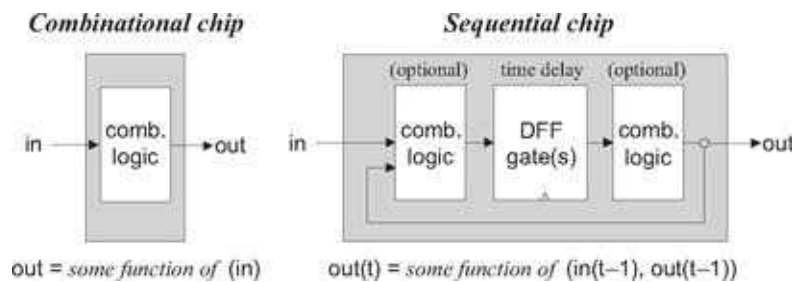


Figure 3.4 Combinational versus sequential logic (in and out stand for one or more input and output variables). Sequential chips always consist of a layer of DFFs sandwiched between optional combinational logic layers.

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

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

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

3.2 Specification

This section specifies a hierarchy of sequential chips:

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

3.2.1 Data-Flip-Flop

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



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

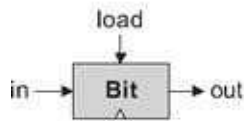
Like Nand gates, DFF gates enter our computer architecture at a very low level. Specifically, all the sequential chips in the computer (registers, memory, and counters) are based on numerous DFF gates. All these DFFs are connected to the same master clock, forming a huge distributed “chorus line.” At the beginning of each clock cycle, the outputs of all the DFFs in the computer commit to their inputs during the previous time unit. At all other times, the DFFs are “latched,” meaning that changes in their inputs have no immediate effect on their outputs. This conduction operation effects any one of the millions of DFF gates that make up the system, about a billion times per second (depending on the computer’s clock frequency).

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

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

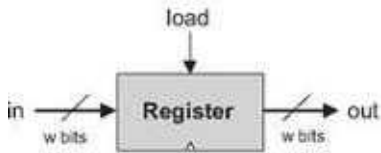
3.2.2 Registers

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



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

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



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

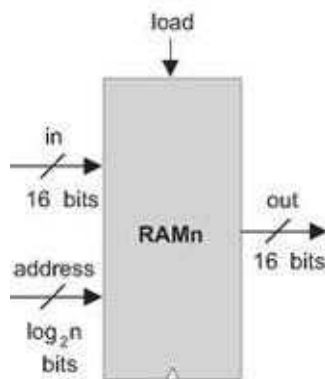
The Bit and Register chips have exactly the same read/write behavior:

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

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

3.2.3 Memory

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



Chip name: RAMn // n and k are listed below

Inputs: in[16], address[k], load

Outputs: out[16]

Function: $out(t) = RAM[address(t)](t)$

If load($t-1$) then

$RAM[address(t-1)](t) = in(t-1)$

Comment: "=" is a 16-bit operation.

The specific RAM chips needed for the Hack platform are:

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

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

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

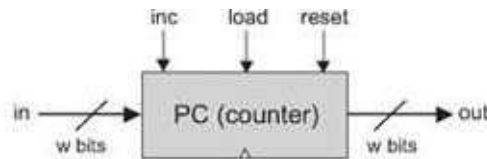
3.2.4 Counter

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

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

3.3 Implementation

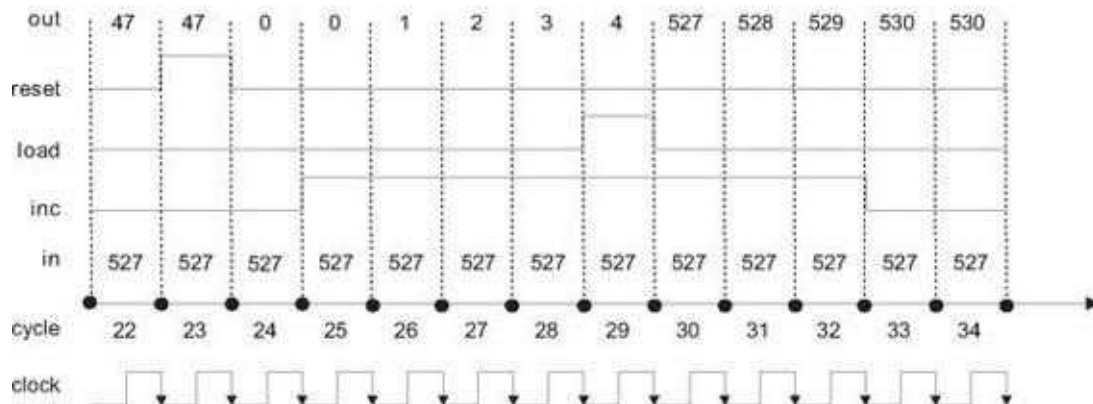
Flip-Flop DFF gates can be implemented from lower-level logic gates like those built in chapter 1. However, in this book we treat DFFs as primitive gates, and thus they can be used in hardware construction projects without worrying about their internal implementation.



```

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

```



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

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

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

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

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

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

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

3.4 Perspective

The cornerstone of all the memory systems described in this chapter is the flip-flop—a gate that we treated here as an atomic, or primitive, building block. The usual approach in hardware textbooks is to construct flip-flops from elementary combinatorial gates (e.g., Nand gates) using appropriate feedback loops. The standard construction begins by building a simple (non-clocked) flip-flop that is bi-stable, namely, that can be set to be in one of two states. Then a clocked flip-flop is obtained by cascading two such simple flip-flops, the first being set when the clock ticks and the second when the clock tocks. This “master-slave” design endows the overall flip-flop with the desired clocked synchronization functionality.

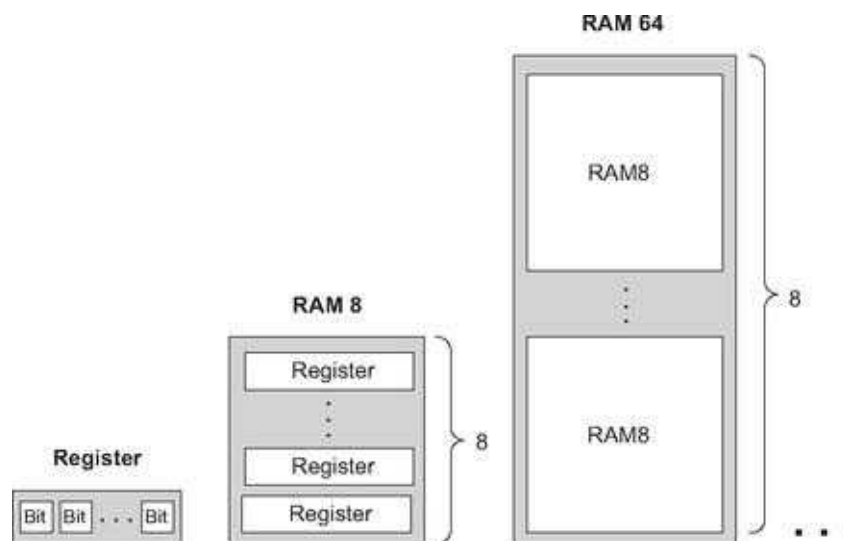


Figure 3.6 Gradual construction of memory banks by recursive ascent. A w -bit register is an array of w binary cells, an 8-register RAM is an array of eight w -bit registers, a 64-register RAM is an array of eight RAM8 chips, and so on. Only three more similar construction steps are necessary to build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

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

In closing, we should mention that memory devices of modern computers are not always constructed from standard flip-flops. Instead, modern memory chips are usually very carefully optimized, exploiting the unique physical properties of the underlying storage technology. Many such alternative technologies are available today to computer designers; as usual, which technology to use is a cost-performance issue.

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

3.5 Project

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

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

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

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

The Directory Structure of This Project When constructing RAM chips from smaller ones, we recommend using built-in versions of the latter. Otherwise, the simulator may run very slowly or even out of (real) memory space, since large RAM chips contain tens of thousands of lower-level chips, and all these chips are kept in memory (as software objects) by the simulator. For this reason, we have placed the RAM512.hdl, RAM4K.hdl, and RAM16K.hdl programs in a separate directory. This way, the recursive descent construction of the RAM4K and RAM16K chips stops with the RAM512 chip, whereas the lower-level chips from which the latter chip is made are bound to be built-in (since the simulator does not find them in this directory).

Steps We recommend proceeding in the following order:

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