

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

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

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

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

1.1 Background

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

1.1.1 Boolean Algebra

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

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

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

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

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

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

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

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

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

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

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

Figure 1.2 All the Boolean functions of two variables.

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

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

1.1.2 Gate Logic

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

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

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

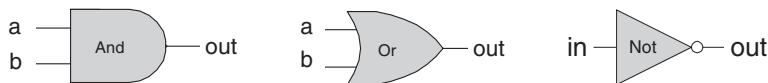


Figure 1.3 Standard symbolic notation of some elementary logic gates.

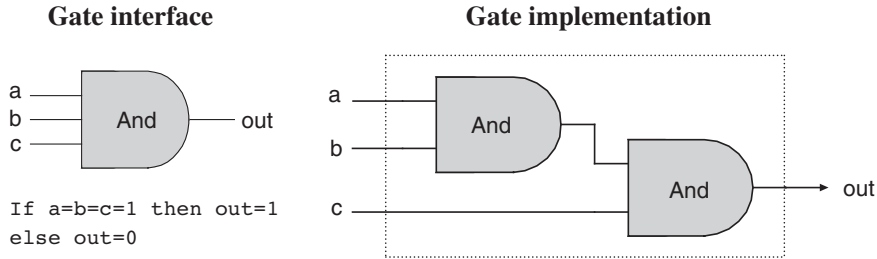


Figure 1.4 Composite implementation of a three-way And gate. The rectangle on the right defines the conceptual boundaries of the gate interface.

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

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

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

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

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

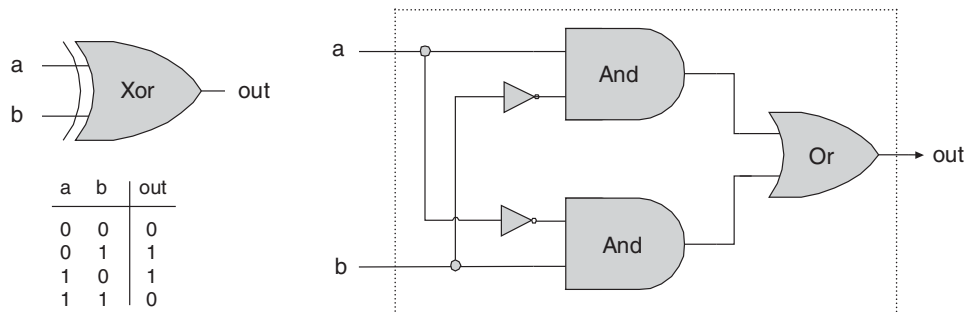


Figure 1.5 Xor gate, along with a possible implementation.

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

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

1.1.3 Actual Hardware Construction

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

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

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

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

1.1.4 Hardware Description Language (HDL)

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

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

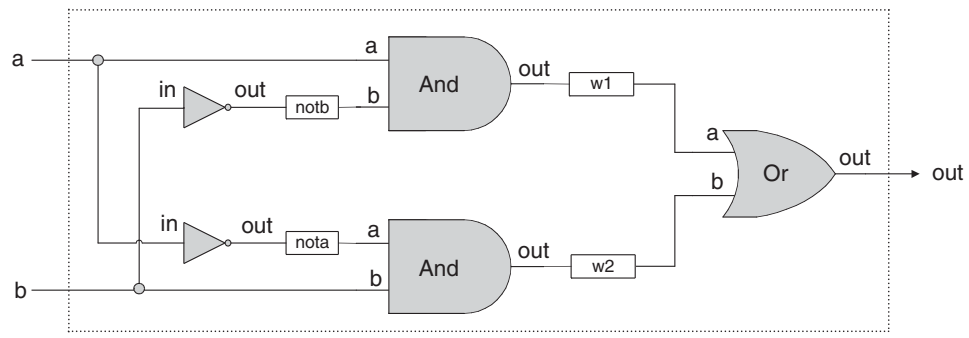
eters can be simulated and quantified by the hardware simulator, helping the designer optimize the design until the simulated chip delivers desired cost/performance levels.

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

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

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

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



<i>HDL program</i> (Xor.hdl)	<i>Test script</i> (Xor.tst)	<i>Output file</i> (Xor.out)															
<pre>/* Xor (exclusive or) gate: If a<>b out=1 else out=0. */ CHIP Xor { IN a, b; OUT out; PARTS: Not(in=a, out=nota); Not(in=b, out=notb); And(a=a, b=notb, out=w1); And(a=nota, b=b, out=w2); Or(a=w1, b=w2, out=out); }</pre>	<pre>load Xor.hdl, output-list a, b, out; set a 0, set b 0, eval, output; set a 0, set b 1, eval, output; set a 1, set b 0, eval, output; set a 1, set b 1, eval, output;</pre>	<table><tr><td>a</td><td>b</td><td>out</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	a	b	out	0	0	0	0	1	1	1	0	1	1	1	0
a	b	out															
0	0	0															
0	1	1															
1	0	1															
1	1	0															

Figure 1.6 HDL implementation of a Xor gate.

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

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

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

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

1.1.5 Hardware Simulation

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

1.2 Specification

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

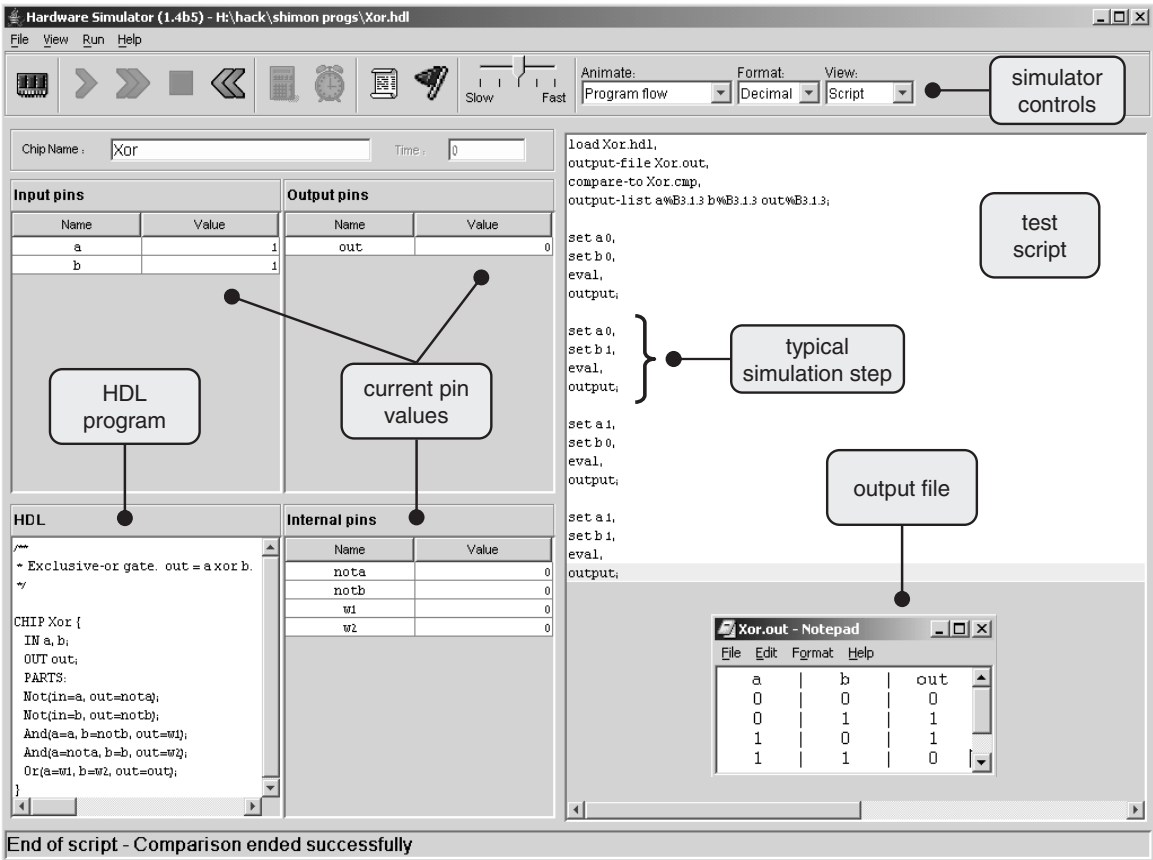


Figure 1.7 A screen shot of simulating an Xor chip on the hardware simulator. The simulator state is shown just after the test script has completed running. The pin values correspond to the last simulation step ($a = b = 1$). Note that the *output file* generated by the simulation is consistent with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

1.2.1 The Nand Gate

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

a	b	Nand(a, b)
0	0	1
0	1	1
1	0	1
1	1	0

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

```

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

```

1.2.2 Basic Logic Gates

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

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

```

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

```

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

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

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

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

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

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

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

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

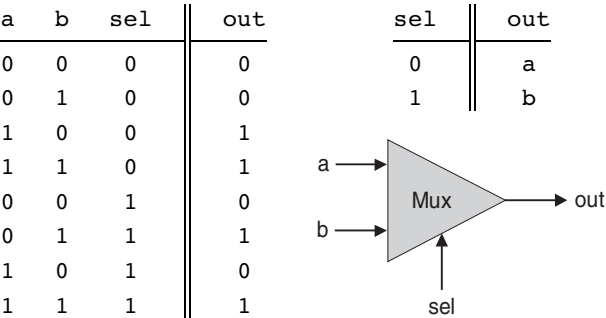


Figure 1.8 Multiplexor. The table at the top right is an abbreviated version of the truth table on the left.

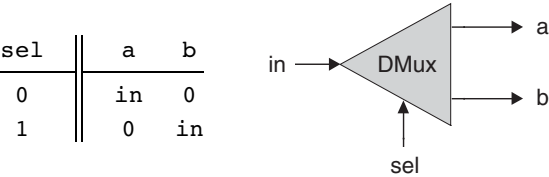


Figure 1.9 Demultiplexor.

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

Chip name: DMux

Inputs: in, sel

Outputs: a, b

Function: If sel=0 then {a=in, b=0} else {a=0, b=in}.

1.2.3 Multi-Bit Versions of Basic Gates

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

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

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

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

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

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

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

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

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

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


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

```

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

```

1.2.4 Multi-Way Versions of Basic Gates

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

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

```

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

```

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

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

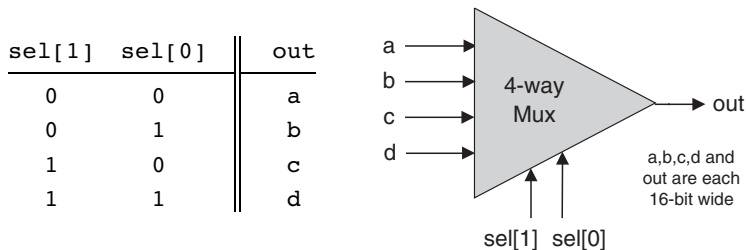


Figure 1.10 4-way multiplexor. The width of the input and output buses may vary.

```

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

```

```

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

```

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

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

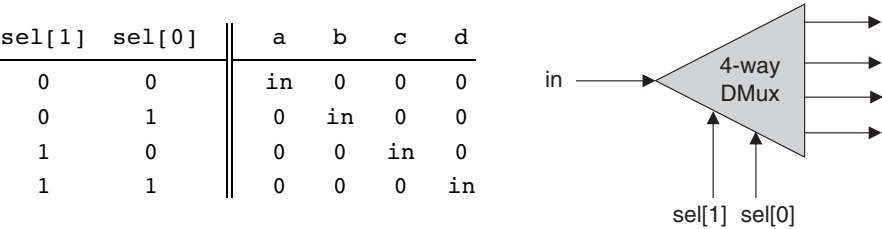


Figure 1.11 4-way demultiplexor.

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

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

1.3 Implementation

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

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

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

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

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

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

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

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

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

Multi-Way Gates: Implementation tip: Think forks.

1.4 Perspective

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

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

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

1.5 Project

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

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

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

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.

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.

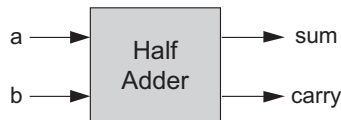
2.2 Specification

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:

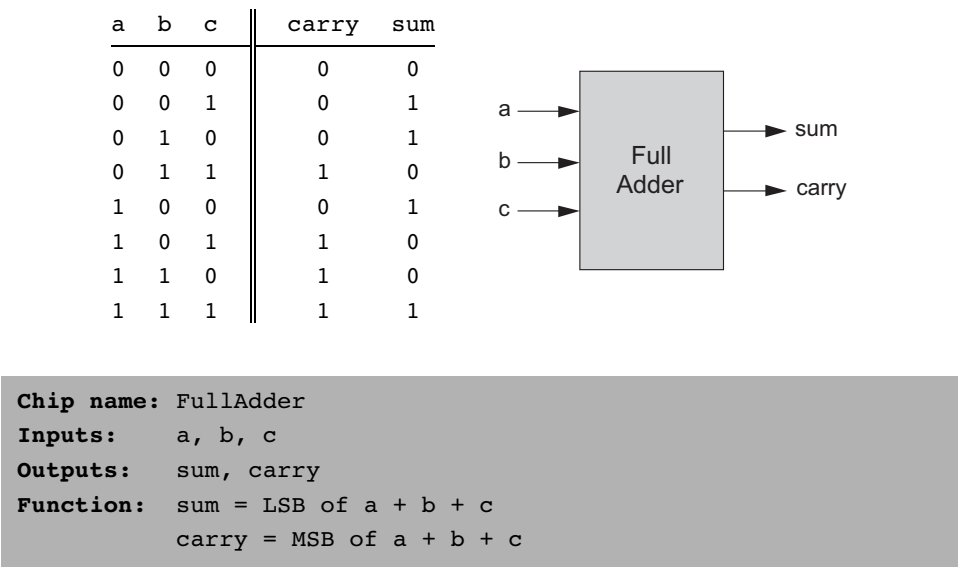


Figure 2.3 Full-adder, designed to add 3 bits.

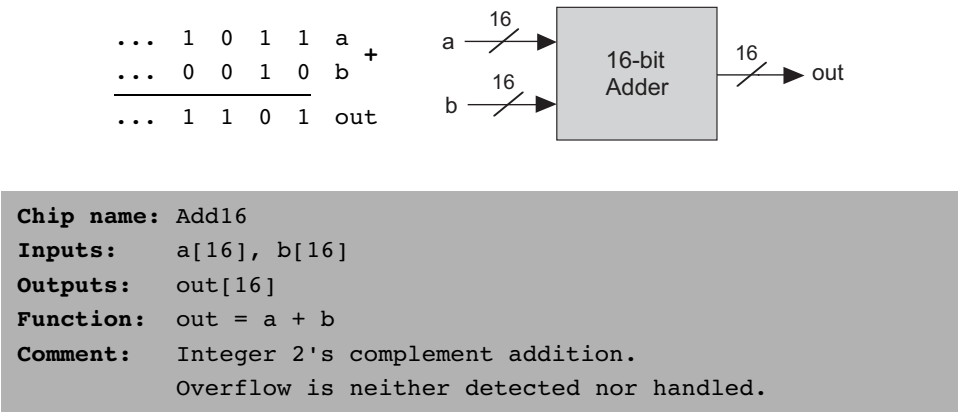


Figure 2.4 16-bit adder. Addition of two n -bit binary numbers for any n is “more of the same.”

```

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

```

2.2.2 The Arithmetic Logic Unit (ALU)

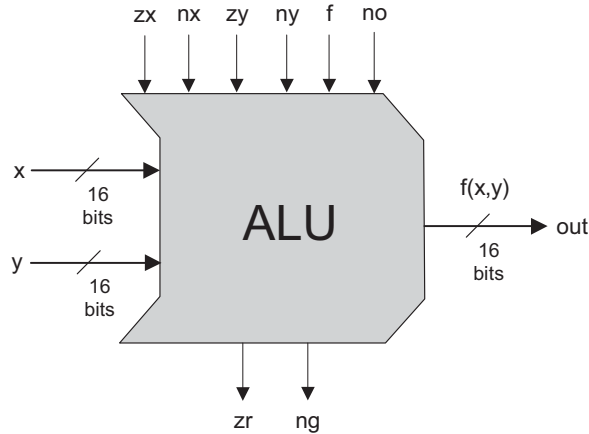
The specifications of the adder chips presented so far were generic, meaning that they hold for any computer. In contrast, this section describes an ALU that will later become the centerpiece of a specific computer platform called *Hack*. At the same time, the principles underlying the design of our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

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

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

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

For example, let us consider the twelfth row of figure 2.6, where the ALU is instructed to compute the function $x-1$. The zx and nx bits are 0, so the x input is neither zeroed nor negated. The zy and ny bits are 1, so the y input is first zeroed, and then negated bit-wise. Bit-wise negation of zero, $(000 \dots 00)_{\text{two}}$, gives $(111 \dots 11)_{\text{two}}$, the 2's complement code of -1 . Thus the ALU inputs end up being x



Chip name: ALU

Inputs: x[16], y[16], // Two 16-bit data inputs
 zx, // Zero the x input
 nx, // Negate the x input
 zy, // Zero the y input
 ny, // Negate the y input
 f, // Function code: 1 for Add, 0 for And
 no // Negate the out output

Outputs: out[16], // 16-bit output
 zr, // True iff out=0
 ng // True iff out<0

Function: if zx then x = 0 // 16-bit zero constant
 if nx then x = !x // Bit-wise negation
 if zy then y = 0 // 16-bit zero constant
 if ny then y = !y // Bit-wise negation
 if f then out = x + y // Integer 2's complement addition
 else out = x & y // Bit-wise And
 if no then out = !out // Bit-wise negation
 if out=0 then zr = 1 else zr = 0 // 16-bit eq. comparison
 if out<0 then ng = 1 else ng = 0 // 16-bit neg. comparison

Comment: Overflow is neither detected nor handled.

Figure 2.5 The Arithmetic Logic Unit.

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

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

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

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

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

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

2.3 Implementation

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

Half-Adder An inspection of figure 2.2 reveals that the functions $sum(a, b)$ and $carry(a, b)$ happen to be identical to the standard $Xor(a, b)$ and $And(a, b)$ 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.”

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

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

Steps We recommend proceeding in the following order:

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