1. Boolean Logic[1]

Such simple things,
And we make of them something so complex it defeats us,
Almost.

(John Ashbery, American poet, 1927-)

Every digital device – be it a personal computer, a cellular telephone, or a network router – is based on the same set of elementary logic gates. In this chapter we start out with one primitive logic gate – Nand – and build all the other elementary logic gates from it. The result is a rather standard set of gates that will be used in the construction of more advanced chips in subsequent chapters.

All the hardware chapters in the book, beginning with this one, have the same structure. Each chapter is focused 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 abstraction, i.e. the various services that they should deliver, one way or another. Having presented the *what*, a subsequent *Implementation* section propose guidelines and hints about *how* the chips can be implemented in practice. A *Perspective* section rounds up the chapter with comments on important topics that were left out from the discussion, with pointers to further reading and self-study. Each chapter concludes with a technical *Build It* section. This section gives step-by-step instructions for actually building the chips on your home computer, using the hardware simulator supplied by the book's web site.

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 logic and Boolean functions. We continue with a description of how Boolean gates can be inter-connected in order to achieve the complex functionality necessary for building typical hardware circuits. We conclude the background section with a description of how hardware design is actually undertaken in practice, using software simulation.

[1] From *The Digital Core*, by Nisan & Schocken, forthcoming in 2003, www.idc.ac.il/csd

Boolean Functions and Boolean Algebra

Boolean algebra deals with Boolean (or binary) values that are typically labeled true/false, 1/0, yes/no, on/off, etc. We will use 1 and 0. A Boolean function is a function that operates on binary inputs and returns binary outputs. Since all computer hardware today 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 specify Boolean functions is the first step toward constructing computer architectures.

Truth Table representation of a Boolean Function: The simplest way to specify a Boolean function is to provide a full enumeration of 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. An example of a truth table for some arbitrary 3-input Boolean function is given in table 1-1.

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

TABLE 1-1: Truth table of the Boolean function $f(x,y,z)=(x+y)(y+x\bar{y}z)$

The first three columns of Table 1-1 enumerate all the possible binary values of the function's variables. For each one of the 2^n possible tuples $v_1 \ldots v_n$ (here n=3), the last column gives the value of $f(v_1 \ldots v_n)$.

Boolean Expressions: Every Boolean function can also be expressed 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".

One may verify that the function whose truth-table was given in table 1-1 is equivalently given by the Boolean expression $f(x,y,z) = (x+y)(y+x\bar{y}z)$. For example, let us evaluate this expression on the inputs $x = 0$ and $y = z = 1$ (4th row in the table). Since y is 1, it follows that $x + y = 1$ and $y + x\bar{y}z = 1$, and thus $(x+y)(y+x\bar{y}z) = 1$. The complete verification of the equivalence between the expression and the truth table is achieved by evaluating the expression on each of the 8 possible input combinations, verifying that it yields the same value listed in the output column of the respective table row.

Canonical Representation: As it turns out, every Boolean function can be expressed using at least one Boolean expression. In particular, there is a canonical way to generate a Boolean expression that is equivalent to a given truth-table. We start by constructing a *term* for each input combination (row) where the function has value 1. This *term* is a Boolean function that gives value 1 exactly for this input combination. This is achieved by And-ing together *literals* (variables or their negations) that fix the values of all the inputs in the respective row. For example, the term associated with the input combination $x = 0$ and $y = z = 1$ (4th row in the table) is $\bar{x}\,y\,z$. If we now Or-together the terms for all the rows where the function has value 1, we get a Boolean expression that is equivalent to the given truth-table. For example, the function whose truth table is given in table 1-1, $f(x,y,z) = (x+y)(y+x\bar{y}z)$, can be expressed canonically by Or-ing the five terms that correspond to the rows where the function has value 1: $f(x,y,z) = \bar{x}\,y\,\bar{z} + \bar{x}\,y\,z + x\,\bar{y}\,z + x\,y\,\bar{z} + x\,y\,z$. It follows that every Boolean function can be expressed using three Boolean operators only: And, Or, and Not.

Two-input Boolean Functions: An inspection of the structure of a truth table like Table 1-1 reveals that the number of Boolean functions that can be defined over *n* binary variables is 2^{2^n}. In particular, two binary variables span $2^{2^2} = 16$ different Boolean functions. These functions are listed in Table 1-2.

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

TABLE 1-2: All the Boolean functions of two variables along with their common names, notations, and truth table definitions.

The 16 functions in Table 1-2 were constructed systematically, by enumerating all the possible 4-wise combinations of binary values in the four right columns. Each function has a conventional name that describes 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 (e.g. x Or y = (x Nand x) Nand (y Nand y). Since, as we have seen, 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 important practical implications:

once we have in our disposal a physical device that implements the Nand operation, we can implement in hardware any Boolean function.

Gate Logic

A *gate* is a physical implementation of a Boolean function. Typically, gates are built from tiny switching devices, called *transistors*, wired in a certain topology designed to effect the gate functionality. Although most digital computers 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 50 years researchers have built many hardware implementations of Boolean functions, including magnetic, optical, biological, hydraulic, and even tinker toy–based, 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 *gate* 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 logic gates, trusting that someone else (the physicists and electrical engineers – god bless their souls) will figure out how to actually realize them in hardware. Hence, a *primitive gate* (see Fig. 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 inter-connecting them, leading to the construction of *composite* gates.

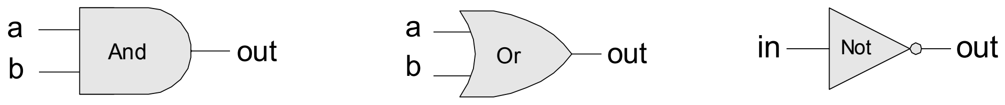


FIGURE 1-3: Standard symbolic notation of some elementary logic gates.

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 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, And(a,b,c)= And(And(a,b),c). Next, we can use this result to construct the composite gate depicted in the right side of Fig. 1-4.

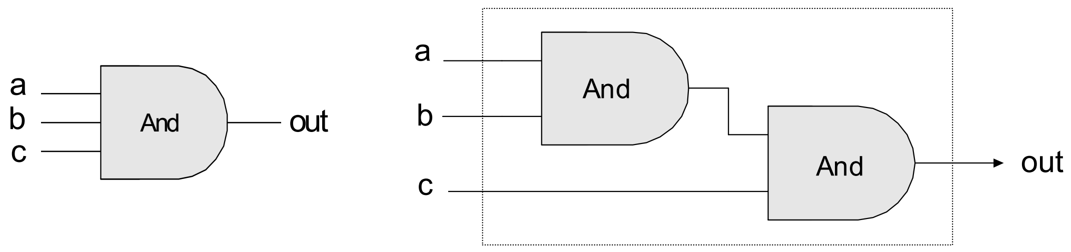
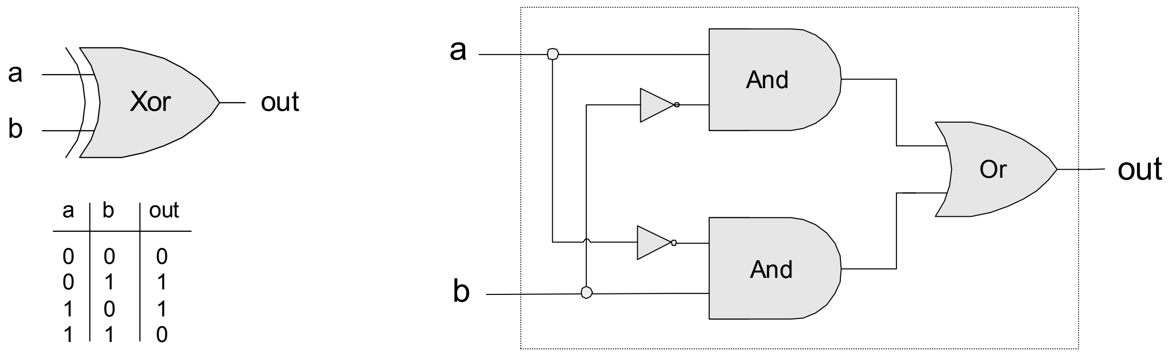


FIGURE 1-4: Composite implementation of a 3-way And gate, based on the observation And(a,b,c)=And(And(a,b),c). The rectangle on the right defines the conceptual boundaries of the gate interface.

The construction described in Fig. 1-4 is a simple example of "gate logic," also called "logic design". Simply put, logic design is the art of inter-connecting elementary 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 Fig. 1-4) looks just like that of primitive gates. At the same time, their internal structure can be rather complex.

We conclude this section with one more logic design example: a gate logic implementation of the Xor function, depicted in Fig. 1-5. The diagram is based on the functional specification Xor$(a,b) = a \cdot \overline{b} + \overline{a} \cdot b$, or, in prefix notation, Xor(a,b)=Or(And(a,Not(b)),And(Not(a),b)). Note however that this Boolean expression is not necessarily the best implementation solution of the Xor function In general, more than one gate architecture can implement the same Boolean function, and some architectures will be better than others in terms of cost, speed, and simplicity. Thus, from a functional standpoint, the fundamental requirement is that *the gate will implement the function's truth table, in one way or another*. From an efficiency standpoint, the general rule is to try to do more with less, i.e. use as few gates as possible.



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 1-5: External (left) and internal (right) views of a Xor gate. The external view specifies the *gate interface*, showing the names of its input and output variables and describing its operation using a truth table or some other descriptive means.

Composite logic gates may be viewed from two different perspectives: external and internal. The right-hand side of Fig. 1-5 gives the Xor-gate's internal architecture, or *implementation*, whereas the left side shows only the gate *interface*, i.e. the input and output pins that the gate exposes to the outside world. The gate implementation diagram is relevant only to the gate designer, whereas the gate interface is the right level of detail for *other* designers who wish to use the gate as an off-the-shelf component, without paying attention to its internal structure.

In sum, the art of logic design can be described as follows: given an external specification of a gate, find a way to create an internal implementation for it, using other gates that were already implemented. This, in a nutshell, is what the rest of the chapter is concerned with.

Actual Hardware Construction

Having described the logic of designing complex gates by composing more primitive ones, we now turn to describe how composite gates can be actually built. Let us start with an intentionally naïve example.

Suppose we open a chip production shop in our home garage. Our first contract is to build a hundred Xor gates. Using the order's down payment, 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 the Xor gate diagram from Fig. 1-5 to our garage wall, and proceed to implement it using our hardware. First, we take two And gates, two Not gates, and one Or gate, and mount them on a board in more or less the same layout specified in the diagram. Next, we connect the chips to each other by running copper wires among them, and 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) will be asked to construct some other chips in the future, we'll be able to use these Xor gates as elementary building blocks, just like 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 will fail to deliver the desired outputs, we will have to tinker 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!

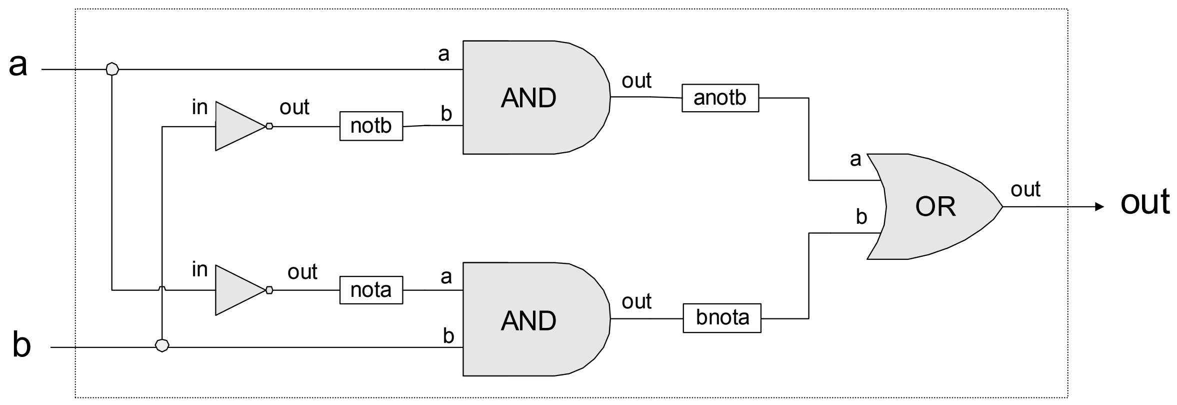
Hardware Description Language

Indeed there is. Today, hardware designers no longer build anything with their bare hands. Instead, they plan and optimize the chip architecture on a computer workstation, using a structured modeling formalism called *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. That is to say, a special software tool, called *hardware simulator*, takes the HDL program as input, and builds an image of the implied 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 person 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 (and this is perhaps the right place to point out that a Xor chip can be built with 3 elementary gates rather than 5, as we have done before). All these parameters can be simulated and quantified by the hardware simulator, helping the designer optimize the design until the simulated chip delivers desired 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, i.e. 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.

Example: Building a Xor Gate: By definition, Xor returns true when its two inputs have opposing values, i.e. Xor(a,b)=Or(And(a,Not(b)),And(Not(a),b)). This logic can be expressed either graphically, as a gate diagram, or textually, as an HDL program (Fig. 1-10). The latter program is written in the HDL variant used throughout this book, which is completely defined in appendix A.



HDL program

```
// Xor gate
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not(in=a,out=Nota);
   Not(in=b,out=Notb);
   And(a=a,b=Notb,out=aNotb);
   And(a=Nota,b=b,out=bNota);
   Or(a=aNotb,b=bNota,out=out);
}
```

Test script

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

Output file

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

FIGURE 1-10: HDL implementation of a Xor chip, including a Xor.hdl program, Xor.tst test script, and Xor.out output file.

Explanation: An HDL definition of a chip consists of a *header* section and a *parts* section. The *header* section describes the chip *interface*, or *signature*, which specifies 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 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 description of the *interfaces* of the underlying parts. For example, Fig. 1-10 implies 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. Without knowing this API-based convention, it would be impossible to plug these chip parts into the present code.

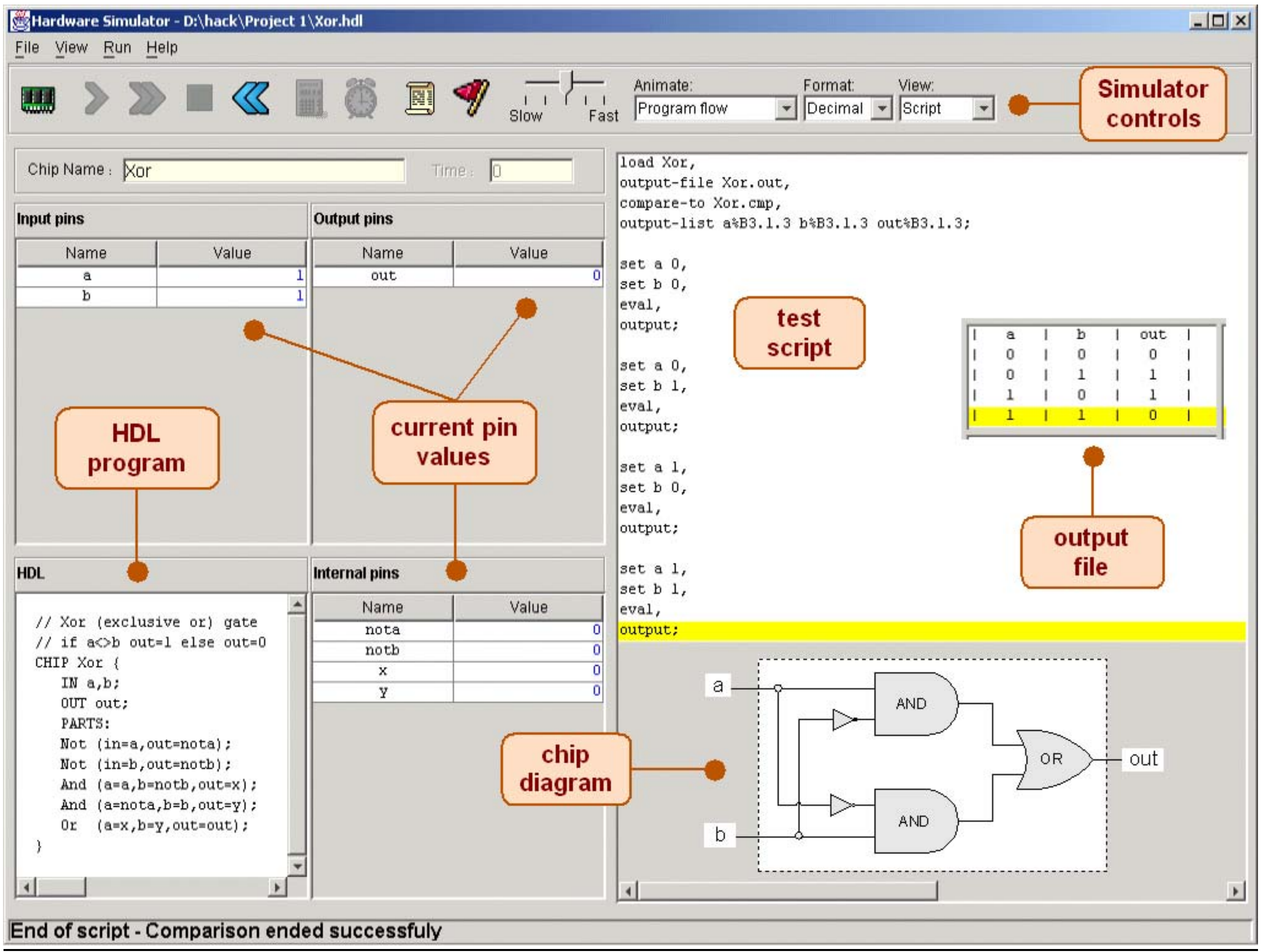
Inter-part connections are described by creating and connecting *internal pins*, as needed. For example, consider the bottom of the gate diagram, where the output of a Not gate is piped into the input of a subsequent And gate. The HDL code describes this connection by the pair of statements `Not(...,out=Nota)` and `And(a=Nota,...)`. The first statement creates an internal pin (wire) named Nota, and then feeds out into it. The second statement feeds the value of the Nota pin into the a input of an And gate. Note that pins may have an unlimited fan-out. For example, the input a is simultaneously fed into both an And gate and a Not gate. In gate diagrams, multiple connections are described using forks. In HDL, the existence of forks is implied by the code.

Testing: Rigorous quality assurance requires that chips will be tested in a specific, replicable, and well-documented fashion. With that in mind, hardware simulators are usually able to run test scripts, written in some scripting language. In particular, the test script in Fig. 1-10 is written in the scripting language understood by the hardware simulator supplied with this book. Both the simulator and the scripting language are described fully in appendix A.

Let us give a brief description of the test script from Fig. 1-10. The first two lines of the test script instruct the simulator to load the Xor.hdl program and get ready to print the current values of selected variables. Next, the script lists a series of testing scenarios, designed to simulate the various contingencies under which the Xor gate will have to operate in "real life" situations. In each scenario, the script instructs the simulator to bind the gate inputs to certain data values, compute the gate 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 chip. The resulting output file (right side of Fig. 1-10) can then be viewed as an empirical proof that the chip is well-defined. The luxury of such certitude is not feasible in more complex chips, as we will see later.

Hardware Simulation

Although HDL is a hardware construction language, the process of writing and debugging an HDL program 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 rather then using a compiler and a virtual machine to translate and test it, we use a *hardware simulator* instead. 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 in 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, which is sufficiently powerful to illustrate all the key elements of the hardware design process. This simulator is all you need in order to build, test, and integrate all the chips presented in the book, leading to the construction of a powerful general-purpose computer. Fig. 1-11 illustrates a typical chip simulation session.



Animate: Program flow
Format: Decimal
View: Script

Chip Name : Xor
Time : 0

Input pins

Name	Value
a	1
b	1

Output pins

Name	Value
out	0

HDL

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

Internal pins

Name	Value
nota	0
notb	0
x	0
y	0

```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

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;
```

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

End of script - Comparison ended successfuly

FIGURE 1-11: Chip Simulation. A screen shot of simulating a Xor chip on the hardware simulator, with an *output file* and *gate diagram* superimposed on it. The simulator state is shown just after the test script stopped running. Note that the *output file* is consistent with the Xor truth table, indicating that the currently loaded HDL program provides a correct implementation of Xor. The *compare file*, not shown in the figure, 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 left of the screen.

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 next chapters 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. Importantly, we provide only the gates *specifications*, or *interfaces*, delaying *implementation* issues to a subsequent section. Readers who wish to construct the specified gates in HDL (*Hardware Description Language*) are welcome to do so, after reading Appendix A. The gates can be built and simulated on your home computer, using the hardware simulator that can be downloaded from the book web site.

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

Throughput this section, 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.
```

Basic Logic Gates

Some of the logic gates presented below 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.
```

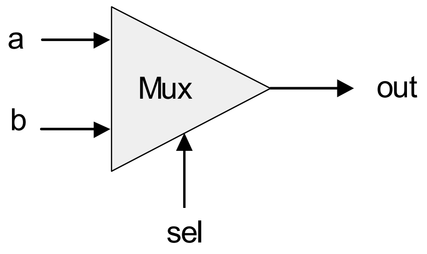
Xor: The Xor function, also known as "exclusive or," returns 1 when its two inputs have opposing values, and 0 otherwise.

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

Multiplexor: A multiplexor is a 3-input gate that uses one of the inputs, called "selection bit", to select and output one of the other two inputs, called "data bits". Although a better name for this device could have been *selector*, the name *multiplexor* is commonly used since a similar device is used in telecommunications systems to combine (multiplex) several input signals over a single output wire.

a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

sel	out
0	a
1	b

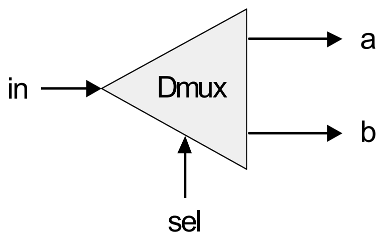


```
Chip name:  Mux
Inputs:     a,b,sel
Outputs:    out
Function:   If sel=0 then out=a else out=b.
```

FIGURE 1-6: Multiplexor. The table at the top right is an abbreviated version of the truth table that appears on the left.

Demultiplexor: A demultiplexor performs the opposite function of a multiplexor: it takes a single input and channels it to one of two possible output wires according to a selector input that specifies which wire to chose.

sel	a	b
0	in	0
1	0	in



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

FIGURE 1-7: Deultiplexor.

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 busses. 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 a simple package, we can encapsulate the array in a single chip whose interface consists of two 32-bit input busses and one 32-bit output bus.

This section describes a typical set of multi-bit logic gates, as needed for the construction of a 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 a syntax similar to that used to handle arrays in programming languages. 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]
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 drawn from its two *n*-bit input busses:

```
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 drawn from its two *n*-bit input busses:

```
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 Fig. 1-6, only 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
Function:   if sel=0 then for i=0..15 out[i]=a[i]
            else for i=0..15 out[i]=b[i].
```

Multi-Way Versions of Basic Gates

Many two-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 1-bit inputs is 1, and 0 otherwise.

```
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 is a device that selects one of *m* *n*-bit input busses and outputs it to a single *n*-bit output bus. The selection is specified by a set of *k* control (input) bits, where $k = \log_2 m$.

s0	s1	out
0	0	a
0	0	b
0	1	c
0	1	d

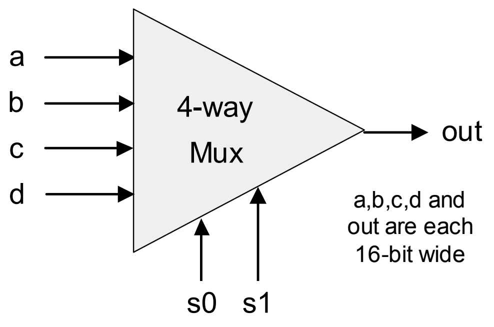


FIGURE 1-8: 4-way multiplexor. The width of the input busses (a,b,c,d) and output bus (out) may vary from one computer platform to another.

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:

```
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=c
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 is a device that channels a single *n*-bit input into one of *m* possible *n*-bit outputs. The selection is specified by a set of *k* control (input) bits, where $k = \log_2 m$.

s0	s1	a	b	c	d
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

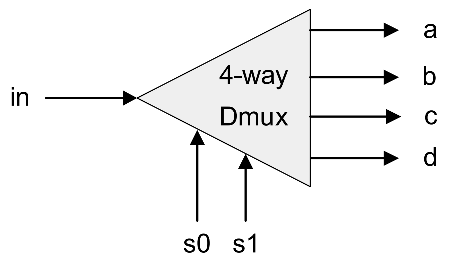


FIGURE 1-9: 4-way demultiplexor.

The 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:

```
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 the elementary building blocks from which everything else can be built. Operationally, primitive gates have an "off-the-shelf" implementation which is supplied externally. Thus, primitive gates can be used in the construction of other gates and chips without worrying about how they are actually implemented. 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 outline 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. Importantly, note 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 rather simple. Tip: think negative.

And: Once again, the gate implementation is rather simple. Tip: think double-negative.

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

Multiplexor / Demultiplexor: Likewise, 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 1-bit inputs. This implementation task is rather boring, but it will carry its weight when these multi-bit gates will be used in the overall computer architecture, 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 but useful task.

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 like all of geometry (and any other mathematical field) can be founded on different sets of axioms as alternative points of departure. Detailed treatments of *digital design* (also called *logic design*) techniques can be found in standard undergraduate textbooks like [Hennessy & Patterson, Appendix B] and [Mano, Chapters 2 and 3].

Throughout the chapter, we paid no attention to efficiency considerations, e.g. the number of elementary gates used in constructing a composite gate, or the number of wire cross-overs implied by the design. Such considerations are critically important, and a great deal of computer science and electrical engineering expertise focus on them. Another issue we did not address at all is the physical implementation of gates and chips using the laws of physics, e.g. the role of transistors embedded in silicon. There are of course several such implementations, each having its own characteristic (speed, power requirements, production cost, etc.) Brief discussions of these issues appear in the textbooks mentioned above. More comprehensive treatments of the technological implementations of basic gates usually require some background in electronics and physics and can be found in advanced undergraduate textbooks like [TBD] and [TBD].

1.5 Build It

Objective: Implement the 15 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 main tool that you will use in this project is the hardware simulator supplied by the book web site. Each gate should be implemented in the HDL formalism understood by this simulator, as specified fully in appendix A.

In order to streamline and manage this construction project, we supply 45 files, as follows. For each one of the 15 gates mentioned in the chapter, we provide a skeletal .hdl program with a missing implementation part. In addition, for each gate we provide a .tst script file that tells the hardware simulator how to test it, along with the correct output file that this script should generate, called .cmp or "compare file". All these files are packed in one file named project1.zip. Your job is to complete the missing implementation parts of all the .hdl programs.

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

Tip: As a rule, one should implement the gates in the order in which they are presented in the book. At the same time, lack of .hdl versions of one gate or another should not delay the construction of more advanced chips that rely on them, since the simulator features built-in versions of all the chips described in the book. For example, consider the skeletal `Mux.hdl` program provided with this project. Suppose that for one reason or another you did not complete the implementation of this program, but you still want to use the `Mux` chip as a part in other chip designs, using statements like `Mux(a=in,...)`. This is not a problem, thanks to the following convention. If the simulator fails to find a Mux.hdl file in your working directory, it automatically invokes the built-in chip implementation of Mux, which is pre-supplied with the simulator. This built-in Mux implementation -- a Java class stored in the simulator's BuiltIn directory -- has the same interface and functionality as those of the Mux chip described in the book. Thus, if you want the simulator to ignore the HDL definition of one of the chips in your working directory, say Mux.hdl, simply rename it to something like Mux.later.

Steps: We recommend to proceed in the following order:

1. Read Appendix A (Hardware Description and Simulation);
2. Download and install the supplied Hardware Simulator;
3. Download and go through the Hardware Simulator Tutorial;
4. Create a new directory called project1 on your personal computer;
5. Download the project1.zip file and extract it to the project1 directory;
6. Build and simulate all the chips.