C

APPENDIX

The Basics of Logic Design

I always loved that word, Boolean.

Claude Shannon
IEEE Spectrum, April 1992
(Shannon's master's thesis showed that the algebra invented by George Boole in the 1800s could represent the workings of electrical switches.)

C.1 Introduction

This appendix provides a brief discussion of the basics of logic design. It does not replace a course in logic design, nor will it enable you to design significant working logic systems. If you have little or no exposure to logic design, however, this appendix will provide sufficient background to understand all the material in this book. In addition, if you are looking to understand some of the motivation behind how computers are implemented, this material will serve as a useful introduction. If your curiosity is aroused but not sated by this appendix, the references at the end provide several additional sources of information.

Section C.2 introduces the basic building blocks of logic, namely, *gates*. Section C.3 uses these building blocks to construct simple *combinational* logic systems, which contain no memory. If you have had some exposure to logic or digital systems, you will probably be familiar with the material in these first two sections. Section C.5 shows how to use the concepts of Sections C.2 and C.3 to design an ALU for the MIPS processor. Section C.6 shows how to make a fast adder,

and may be safely skipped if you are not interested in this topic. Section C.7 is a short introduction to the topic of clocking, which is necessary to discuss how memory elements work. Section C.8 introduces memory elements, and Section C.9 extends it to focus on random access memories; it describes both the characteristics that are important to understanding how they are used in Chapter 4, and the background that motivates many of the aspects of memory hierarchy design in Chapter 5. Section C.10 describes the design and use of finite-state machines, which are sequential logic blocks. If you intend to read Appendix D, you should thoroughly understand the material in Sections C.2 through C.10. If you intend to read only the material on control in Chapter 4, you can skim the appendices; however, you should have some familiarity with all the material except Section C.11. Section C.11 is intended for those who want a deeper understanding of clocking methodologies and timing. It explains the basics of how edge-triggered clocking works, introduces another clocking scheme, and briefly describes the problem of synchronizing asynchronous inputs.

Throughout this appendix, where it is appropriate, we also include segments to demonstrate how logic can be represented in Verilog, which we introduce in Section C.4. A more extensive and complete Verilog tutorial appears elsewhere on the CD.

C.2 Gates, Truth Tables, and Logic Equations

The electronics inside a modern computer are *digital*. Digital electronics operate with only two voltage levels of interest: a high voltage and a low voltage. All other voltage values are temporary and occur while transitioning between the values. (As we discuss later in this section, a possible pitfall in digital design is sampling a signal when it not clearly either high or low.) The fact that computers are digital is also a key reason they use binary numbers, since a binary system matches the underlying abstraction inherent in the electronics. In various logic families, the values and relationships between the two voltage values differ. Thus, rather than refer to the voltage levels, we talk about signals that are (logically) true, or 1, or are **asserted**; or signals that are (logically) false, or 0, or are **deasserted**. The values 0 and 1 are called *complements* or *inverses* of one another.

asserted signal A signal that is (logically) true, or 1.

deasserted signal A signal that is (logically) false, or 0.

Logic blocks are categorized as one of two types, depending on whether they contain memory. Blocks without memory are called *combinational;* the output of a combinational block depends only on the current input. In blocks with memory, the outputs can depend on both the inputs and the value stored in memory, which is called the *state* of the logic block. In this section and the next, we will focus

only on **combinational logic**. After introducing different memory elements in Section C.8, we will describe how **sequential logic**, which is logic including state, is designed.

combinational logic
A logic system whose blocks do not contain memory and hence compute the same output given the same input.

sequential logic
A group of logic elements that contain memory and hence whose value depends on the inputs as well as the current contents of the memory.

Truth Tables

Because a combinational logic block contains no memory, it can be completely specified by defining the values of the outputs for each possible set of input values. Such a description is normally given as a *truth table*. For a logic block with n inputs, there are 2^n entries in the truth table, since there are that many possible combinations of input values. Each entry specifies the value of all the outputs for that particular input combination.

Truth Tables

EXAMPLE

Consider a logic function with three inputs, A, B, and C, and three outputs, D, E, and F. The function is defined as follows: D is true if at least one input is true, E is true if exactly two inputs are true, and F is true only if all three inputs are true. Show the truth table for this function.

ANSWER

The truth table will contain $2^3 = 8$ entries. Here it is:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Truth tables can completely describe any combinational logic function; however, they grow in size quickly and may not be easy to understand. Sometimes we want to construct a logic function that will be 0 for many input combinations, and we use a shorthand of specifying only the truth table entries for the nonzero outputs. This approach is used in Chapter 4 and Appendix D.

Boolean Algebra

Another approach is to express the logic function with logic equations. This is done with the use of *Boolean algebra* (named after Boole, a 19th-century mathematician). In Boolean algebra, all the variables have the values 0 or 1 and, in typical formulations, there are three operators:

- The OR operator is written as +, as in $A + B$. The result of an OR operator is 1 if either of the variables is 1. The OR operation is also called a *logical sum*, since its result is 1 if either operand is 1.
- The AND operator is written as $\cdot$, as in $A \cdot B$. The result of an AND operator is 1 only if both inputs are 1. The AND operator is also called *logical product*, since its result is 1 only if both operands are 1.
- The unary operator NOT is written as $\overline{A}$. The result of a NOT operator is 1 only if the input is 0. Applying the operator NOT to a logical value results in an inversion or negation of the value (i.e., if the input is 0 the output is 1, and vice versa).

There are several laws of Boolean algebra that are helpful in manipulating logic equations.

- Identity law: $A + 0 = A$ and $A \cdot 1 = A$.
- Zero and One laws: $A + 1 = 1$ and $A \cdot 0 = 0$.
- Inverse laws: $A + \overline{A} = 1$ and $A \cdot \overline{A} = 1$.
- Commutative laws: $A + B = B + A$ and $A \cdot B = B \cdot A$.
- Associative laws: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.
- Distributive laws: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$.

In addition, there are two other useful theorems, called DeMorgan's laws, that are discussed in more depth in the exercises.

Any set of logic functions can be written as a series of equations with an output on the left-hand side of each equation and a formula consisting of variables and the three operators above on the right-hand side.

Logic Equations

EXAMPLE

Show the logic equations for the logic functions, *D*, *E*, and *F*, described in the previous example.

ANSWER

Here's the equation for *D:*

$$D = A + B + C$$

F is equally simple:

$$F = A \cdot B \cdot C$$

E is a little tricky. Think of it in two parts: what must be true for *E* to be true (two of the three inputs must be true), and what cannot be true (all three cannot be true). Thus we can write *E* as

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

We can also derive *E* by realizing that *E* is true only if exactly two of the inputs are true. Then we can write *E* as an OR of the three possible terms that have two true inputs and one false input:

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

Proving that these two expressions are equivalent is explored in the exercises.

In Verilog, we describe combinational logic whenever possible using the assign statement, which is described beginning on page C-23. We can write a definition for *E* using the Verilog exclusive-OR operator as `assign E = (A ^ B ^ C) * (A + B + C) * (A * B * C)`, which is yet another way to describe this function. *D* and *F* have even simpler representations, which are just like the corresponding C code: `D = A | B | C` and `F = A & B & C`.

Gates

gate A device that implements basic logic functions, such as AND or OR.

Logic blocks are built from **gates** that implement basic logic functions. For example, an AND gate implements the AND function, and an OR gate implements the OR function. Since both AND and OR are commutative and associative, an AND or an OR gate can have multiple inputs, with the output equal to the AND or OR of all the inputs. The logical function NOT is implemented with an inverter that always has a single input. The standard representation of these three logic building blocks is shown in Figure C.2.1.

Rather than draw inverters explicitly, a common practice is to add "bubbles" to the inputs or outputs of a gate to cause the logic value on that input line or output line to be inverted. For example, Figure C.2.2 shows the logic diagram for the function $\overline{\overline{A} + B}$, using explicit inverters on the left and bubbled inputs and outputs on the right.

Any logical function can be constructed using AND gates, OR gates, and inversion; several of the exercises give you the opportunity to try implementing some common logic functions with gates. In the next section, we'll see how an implementation of any logic function can be constructed using this knowledge.

NOR gate An inverted OR gate.

NAND gate An inverted AND gate.

In fact, all logic functions can be constructed with only a single gate type, if that gate is inverting. The two common inverting gates are called **NOR** and **NAND** and correspond to inverted OR and AND gates, respectively. NOR and NAND gates are called *universal*, since any logic function can be built using this one gate type. The exercises explore this concept further.

Check Yourself

Are the following two logical expressions equivalent? If not, find a setting of the variables to show they are not:

- $(A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$
- $B \cdot (A \cdot \overline{C} + C \cdot \overline{A})$

FIGURE C.2.1 Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right. The signals to the left of each symbol are the inputs, while the output appears on the right. The AND and OR gates both have two inputs. Inverters have a single input.

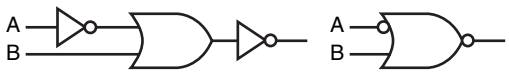


FIGURE C.2.2 Logic gate implementation of $\overline{\overline{A} + B}$ using explicit inverts on the left and bubbled inputs and outputs on the right. This logic function can be simplified to $A \cdot \overline{B}$ or in Verilog, `A & ~ B`.

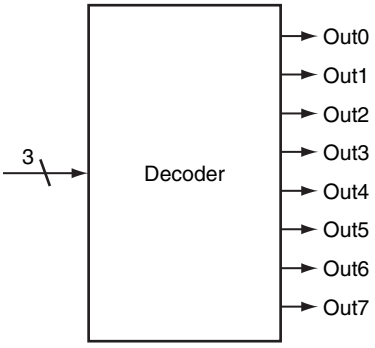
C.3 Combinational Logic

In this section, we look at a couple of larger logic building blocks that we use heavily, and we discuss the design of structured logic that can be automatically implemented from a logic equation or truth table by a translation program. Last, we discuss the notion of an array of logic blocks.

Decoders

One logic block that we will use in building larger components is a **decoder**. The most common type of decoder has an n-bit input and 2^n outputs, where only one output is asserted for each input combination. This decoder translates the n-bit input into a signal that corresponds to the binary value of the n-bit input. The outputs are thus usually numbered, say, Out0, Out1, . . . , Out$2^n - 1$. If the value of the input is i, then Out$i$ will be true and all other outputs will be false. Figure C.3.1 shows a 3-bit decoder and the truth table. This decoder is called a *3-to-8 decoder* since there are 3 inputs and 8 (2^3) outputs. There is also a logic element called an *encoder* that performs the inverse function of a decoder, taking 2^n inputs and producing an n-bit output.

decoder A logic block that has an n-bit input and 2^n outputs, where only one output is asserted for each input combination.



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

FIGURE C.3.1 A 3-bit decoder has 3 inputs, called 12, 11, and 10, and $2^3 = 8$ outputs, called Out0 to Out7. Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

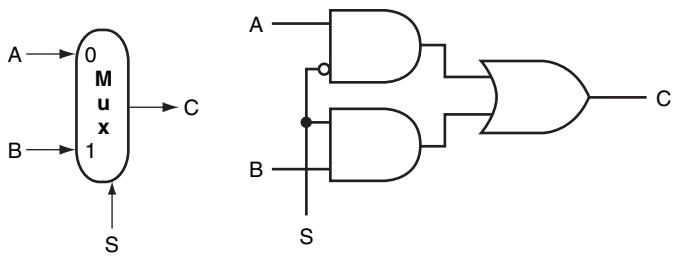


FIGURE C.3.2 A two-input multiplexor on the left and its implementation with gates on the right. The multiplexor has two data inputs (*A* and *B*), which are labeled *0* and *1*, and one selector input (*S*), as well as an output *C*. Implementing multiplexors in Verilog requires a little more work, especially when they are wider than two inputs. We show how to do this beginning on page C-23.

Multiplexors

selector value Also called **control value**. The control signal that is used to select one of the input values of a multiplexor as the output of the multiplexor.

One basic logic function that we use quite often in Chapter 4 is the *multiplexor*. A multiplexor might more properly be called a *selector*, since its output is one of the inputs that is selected by a control. Consider the two-input multiplexor. The left side of Figure C.3.2 shows this multiplexor has three inputs: two data values and a **selector (**or **control) value**. The selector value determines which of the inputs becomes the output. We can represent the logic function computed by a two-input multiplexor, shown in gate form on the right side of Figure C.3.2, as $C = (A \cdot \overline{S}) + (B \cdot S)$.

Multiplexors can be created with an arbitrary number of data inputs. When there are only two inputs, the selector is a single signal that selects one of the inputs if it is true (1) and the other if it is false (0). If there are n data inputs, there will need to be $\lceil \log_2 n \rceil$ selector inputs. In this case, the multiplexor basically consists of three parts:

1. A decoder that generates n signals, each indicating a different input value
2. An array of n AND gates, each combining one of the inputs with a signal from the decoder
3. A single large OR gate that incorporates the outputs of the AND gates

To associate the inputs with selector values, we often label the data inputs numerically (i.e., 0, 1, 2, 3, . . . , $n - 1$) and interpret the data selector inputs as a binary number. Sometimes, we make use of a multiplexor with undecoded selector signals.

Multiplexors are easily represented combinationally in Verilog by using *if* expressions. For larger multiplexors, *case* statements are more convenient, but care must be taken to synthesize combinational logic.

Two-Level Logic and PLAs

As pointed out in the previous section, any logic function can be implemented with only AND, OR, and NOT functions. In fact, a much stronger result is true. Any logic function can be written in a canonical form, where every input is either a true or complemented variable and there are only two levels of gates—one being AND and the other OR—with a possible inversion on the final output. Such a representation is called a *two-level representation*, and there are two forms, called **sum of products** and *product of sums*. A sum-of-products representation is a logical sum (OR) of products (terms using the AND operator); a product of sums is just the opposite. In our earlier example, we had two equations for the output *E:*

sum of products A form of logical representation that employs a logical sum (OR) of products (terms joined using the AND operator).

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

and

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot \mathrm{C} \cdot \overline{\mathrm{A}})$$

This second equation is in a sum-of-products form: it has two levels of logic and the only inversions are on individual variables. The first equation has three levels of logic.

Elaboration: We can also write *E* as a product of sums:

$$E = \overline{(\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{C} + B) \cdot (\overline{B} + \mathrm{C} + A)}$$

To derive this form, you need to use *DeMorgan's theorems*, which are discussed in the exercises.

In this text, we use the sum-of-products form. It is easy to see that any logic function can be represented as a sum of products by constructing such a representation from the truth table for the function. Each truth table entry for which the function is true corresponds to a product term. The product term consists of a logical product of all the inputs or the complements of the inputs, depending on whether the entry in the truth table has a 0 or 1 corresponding to this variable. The logic function is the logical sum of the product terms where the function is true. This is more easily seen with an example.

EXAMPLE

Sum of Products

Show the sum-of-products representation for the following truth table for D.

Inputs			Output
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

ANSWER

There are four product terms, since the function is true (1) for four different input combinations. These are:

$$\overline{A} \cdot \overline{B} \cdot C$$

$$\overline{A} \cdot B \cdot \overline{C}$$

$$A \cdot \overline{B} \cdot \overline{C}$$

$$A \cdot B \cdot C$$

Thus, we can write the function for D as the sum of these terms:

$$D = (\overline{A} \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C)$$

Note that only those truth table entries for which the function is true generate terms in the equation.

programmable logic array (PLA) A structured-logic element composed of a set of inputs and corresponding input complements and two stages of logic: the first generating product terms of the inputs and input complements, and the second generating sum terms of the product terms. Hence, PLAs implement logic functions as a sum of products.

minterms Also called **product terms**. A set of logic inputs joined by conjunction (AND operations); the product terms form the first logic stage of the programmable logic array (PLA).

We can use this relationship between a truth table and a two-level representation to generate a gate-level implementation of any set of logic functions. A set of logic functions corresponds to a truth table with multiple output columns, as we saw in the example on page C-5. Each output column represents a different logic function, which may be directly constructed from the truth table.

The sum-of-products representation corresponds to a common structured-logic implementation called a **programmable logic array (PLA)**. A PLA has a set of inputs and corresponding input complements (which can be implemented with a set of inverters), and two stages of logic. The first stage is an array of AND gates that form a set of **product terms** (sometimes called **minterms**); each product term can consist of any of the inputs or their complements. The second stage is an array of OR gates, each of which forms a logical sum of any number of the product terms. Figure C.3.3 shows the basic form of a PLA.

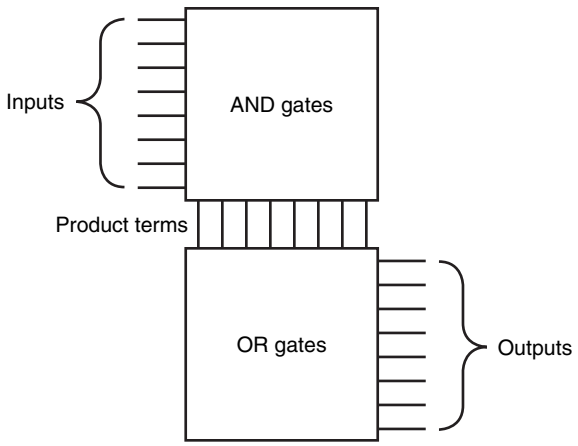


FIGURE C.3.3 The basic form of a PLA consists of an array of AND gates followed by an array of OR gates. Each entry in the AND gate array is a product term consisting of any number of inputs or inverted inputs. Each entry in the OR gate array is a sum term consisting of any number of these product terms.

A PLA can directly implement the truth table of a set of logic functions with multiple inputs and outputs. Since each entry where the output is true requires a product term, there will be a corresponding row in the PLA. Each output corresponds to a potential row of OR gates in the second stage. The number of OR gates corresponds to the number of truth table entries for which the output is true. The total size of a PLA, such as that shown in Figure C.3.3, is equal to the sum of the size of the AND gate array (called the *AND plane*) and the size of the OR gate array (called the *OR plane*). Looking at Figure C.3.3, we can see that the size of the AND gate array is equal to the number of inputs times the number of different product terms, and the size of the OR gate array is the number of outputs times the number of product terms.

A PLA has two characteristics that help make it an efficient way to implement a set of logic functions. First, only the truth table entries that produce a true value for at least one output have any logic gates associated with them. Second, each different product term will have only one entry in the PLA, even if the product term is used in multiple outputs. Let's look at an example.

EXAMPLE

PLAs

Consider the set of logic functions defined in the example on page C-5. Show a PLA implementation of this example for *D*, *E*, and *F*.

ANSWER

Here is the truth table we constructed earlier:

Inputs			Outputs		
A	*B*	*C*	*D*	*E*	*F*
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Since there are seven unique product terms with at least one true value in the output section, there will be seven columns in the AND plane. The number of rows in the AND plane is three (since there are three inputs), and there are also three rows in the OR plane (since there are three outputs). Figure C.3.4 shows the resulting PLA, with the product terms corresponding to the truth table entries from top to bottom.

Rather than drawing all the gates, as we do in Figure C.3.4, designers often show just the position of AND gates and OR gates. Dots are used on the intersection of a product term signal line and an input line or an output line when a corresponding AND gate or OR gate is required. Figure C.3.5 shows how the PLA of Figure C.3.4 would look when drawn in this way. The contents of a PLA are fixed when the PLA is created, although there are also forms of PLA-like structures, called *PALs*, that can be programmed electronically when a designer is ready to use them.

read-only memory (ROM) A memory whose contents are designated at creation time, after which the contents can only be read. ROM is used as structured logic to implement a set of logic functions by using the terms in the logic functions as address inputs and the outputs as bits in each word of the memory.

programmable ROM (PROM) A form of read-only memory that can be programmed when a designer knows its contents.

ROMs

Another form of structured logic that can be used to implement a set of logic functions is a **read-only memory (ROM)**. A ROM is called a memory because it has a set of locations that can be read; however, the contents of these locations are fixed, usually at the time the ROM is manufactured. There are also **programmable ROMs (PROMs)** that can be programmed electronically, when a designer knows their contents. There are also erasable PROMs; these devices require a slow erasure process using ultraviolet light, and thus are used as read-only memories, except during the design and debugging process.

A ROM has a set of input address lines and a set of outputs. The number of addressable entries in the ROM determines the number of address lines: if the