Chapter 1

A Review of logic design

1.1 Boolean Algebra

Despite the complexity of modern-day digital circuits, the fundamental principles upon which they are based are surprisingly simple. Boolean Algebra [Bool54] provides the basis for all digital circuitry from the simplest traffic light controller to the most complex super-computer. The elements of Boolean algebra are the values 0 and 1, along with a set of laws for combining these values. Originally, Boolean algebra concerned itself not with ones and zeros, but with the values True and False. The connection between Boolean equations and the logic of switching circuitry was first recognized by Claude Shannon [Shan38]. Because we are concerned with digital circuits and binary arithmetic, we will use the convention of using zeros and ones instead of the values True and False. At times we may use the terms Low and High voltage instead of zero and one.

The three basic operations of Boolean algebra are NOT, AND and OR. AND and OR are binary operators, while not is a unary operator. (In digital design, Unary and Binary operators are more commonly called one-input and two-input functions.) If *a* and *b* are Boolean variables, then *a* AND *b* is written *ab*, *a* OR *b* is written *a*+*b*, and NOT a is written either a' or $\overline{a}$. Figure 1-1 gives the rules for evaluating these three functions.

And	0	1
0	**0**	**0**
1	**0**	**1**

Or	0	1
0	**0**	**1**
1	**1**	**1**

Not	
0	**1**
1	**0**

Figure 1-1. The Basic Boolean Functions.

These three functions are Universal in the sense that any Boolean function can be computed by performing AND, OR, and NOT functions. There are many algebraic laws that these functions obey, Figure 1-2 lists some of the most important.

Classification	Law
Identity	$a1=1a=a$ $a+0=0+a=a$
Dominance	$a0=0a=0$ $1+a=a+1=1$
Commutativity	$a+b=b+a$ $ab=ba$
Associativity	$a(bc)=(ab)c$ $a+(b+c)=(a+b)+c$
Distributive	$a(b+c)=ab+ac$ $a+bc=(a+b)(a+c)$
Demorgan's Laws	$(a+b)' = a'b'$ $(ab)' = a'+b'$

Figure 1-2. Boolean Identities.

Because AND and OR are commutative and associative, it is meaningful to talk about AND and OR functions with three or more inputs. Regardless of how these functions are evaluated, there is no ambiguity about the result.

1.2 Boolean Expressions

Boolean expressions can be used to express complex Boolean functions in terms of the basic AND, OR and NOT functions. Two examples of such equations are given below.

$$s = ab'c + a'bc + abc' + abc$$
$$q = ab + bc + ac + abc$$

In the above equations, it is assumed that the AND operation has priority over the OR operation. All AND operations are performed first, followed by the OR operations. NOT is applied first, so the primes in the first equation apply only to the immediately preceding variable. These equations are expressed in Sum-of-Products form, which is the usual way of specifying Boolean functions in digital

In general, there are many different Boolean equations for the same function. Good design demands that the most efficient representation be found for each function. Reducing the number of terms in an equation also reduces the amount of hardware necessary to implement the Boolean function. By the same token, reducing the number of factors in a product reduces the complexity of the required hardware. Of the two equations given above, the equation for *s* is

minimal, but the equation for *q* is not. The term abc is unnecessary, because whenever *abc* is equal to 1, *ab* will also be equal to 1. Deleting the abc term gives the following minimal equation.

$$q = ab + bc + ac$$

Although this equation can be minimized by inspection and by the application of Boolean identities, more systematic methods are required to assure the quality of a digital design. The two most important systematic methods are Karnaugh maps [KARN53] and Quine-McClusky minimization [Quin52] [MCCL56].

1.3 Minimization of Boolean Equations

In a Boolean expression such as the following, the terms *ab, bc, ac,* and *abc*, are known as the prime implicants of the function *q*.

$$q = ab + bc + ac + abc$$

Note that the term *abc* contains all input variables, while the term *ab* does not. If a prime implicant contains all input variables in complemented or uncomplemented form, it is called a minterm. A function of *n* inputs has at most 2^n minterms. The process of minimization attempts to obtain an expression with as few prime implicants as possible, and prime implicants with as few variables as possible. Unfortunately, the minimization process is known to be NP-Complete. Nevertheless, there are several approximation algorithms that produce acceptable results.

For functions with six or fewer inputs Karnaugh maps can be used. The Karnaugh map is a manual technique that expresses the function in a form that allows prime implicants to be easily found by inspection. The success of the technique depends, at least in part, on the skill of the person using it. Figure 1-3 illustrates the use of Karnaugh maps for functions with three inputs.

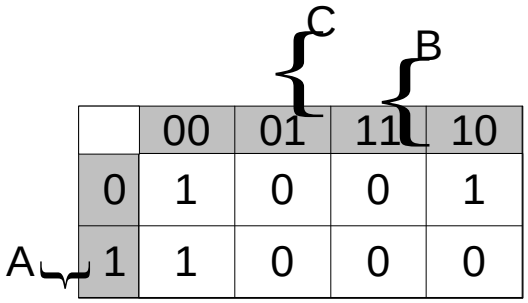


Figure 1-3. A Simple Karnaugh Map.

The function corresponding to this Karnaugh map is $a'b' + b'c'$. Figure 1-4 illustrates a 4-input Karnaugh map.

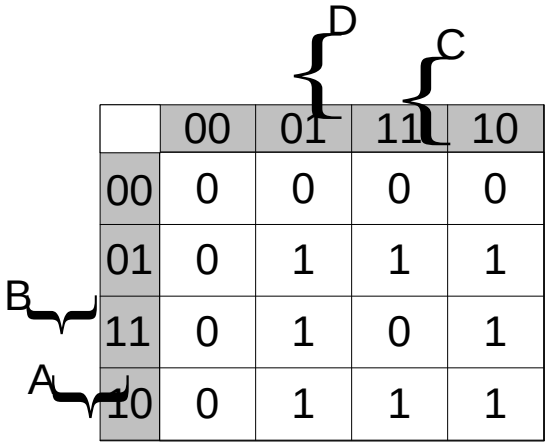


Figure 1-4. A Four-Variable Karnaugh Map.

The function represented by this Karnaugh map has two different minimal Boolean expressions, $ab'd + bc'd + a'bc + acd'$, and $ac'd + a'bd + bcd' + ab'c$. In general, a Boolean function may have several different minimal forms.

The Quine-McClusky technique can be used for Boolean functions with an arbitrary number of inputs, and is amenable to implementation on a computer. Starting with the truth table of a function one builds a list of minterms for the function. Since variable names are not important during the minimization process, the minterms can be represented as positional strings of ones and zeros. Using the set of minterms, one builds a complete collection of prime implicants of the function. Starting with an *n*-input function, the collection of minterms represents the complete set of *n*-variable prime implicants. From the set of n-variable prime implicants, a set of (n-1)-variable prime implicants can be constructed by comparing the n-variable prime implicants to one another. If two prime implicants are identical except for one bit, the two n-variable prime implicants can be combined into a single (n-1)-variable prime implicant by setting the differing bit to *Don't Care*. Don't Care bits are represented by an x in the table. Once a complete set of (n-1)-variable prime implicants has been constructed, it is possible to construct the set of (n-2)-variable prime implicants. When comparing two (n-1)-variable prime implicants, it is necessary for them to differ by only one bit, and neither of the differing bits can be a Don't Care. This process continues until no more prime implicants can be found. This process is illustrated in Figure 1-5.

Variables	Prime Implicants
4	0011, 0111, 1011, 1100, 1101, 1110, 1111
3	0x11, x011, x111, 1x11, 110x, 11x0, 11x1, 111x
2	xx11, 11xx
1	none

Figure 1-5. Prime Implicant Table.

Once all prime implicants of the function have been found, it is necessary to select a collection of prime implicants that cover all the minterms of the function. The selection of an appropriate minimal set of prime implicants is a complex process that is known to be NP-complete. For more information, the reader should refer to a basic book on logic design, or the many technical articles that have been written on the subject.

1.4 Logic Gates

A logic gate is a simple digital circuit that implements an elementary Boolean function. Because its electrical properties are well-known and predictable, its underlying structure can usually be ignored. The three basic types of gates are the **And** gate, the **Or** gate, and the **Not** gate, which correspond to the basic Boolean operations. **And** and **Or** gates are permitted to have an arbitrary number of inputs. Due to technological limitations, gates with a large number of inputs may be implemented as several smaller gates when the circuit is constructed. Figure 1-6 shows the symbols used for And, Or, and Not gates.

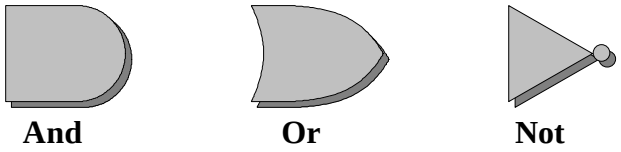


Figure 1-6. Basic Logic Symbols.

There are many other types of gates available in addition to the And, Or, and Not. One commonly used gate is the exclusive Or, which has the truth table and symbol illustrated in Figure 1-7. The exclusive Or gate is usually referred to as Xor.

Xor	0	1
0	**0**	**1**
1	**1**	**0**

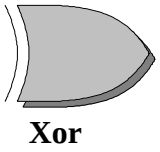


Figure 1-7. The Exclusive Or Function.

More complex functions can be constructed by connecting several gates with branching lines, as illustrated in Figure 1-8. The connections between two gates, or between a gate and the outside world are known as Nets. Signals that come from the outside world into the circuit are called Primary Inputs, while those that flow out of the circuit to the outside world are called Primary Outputs.

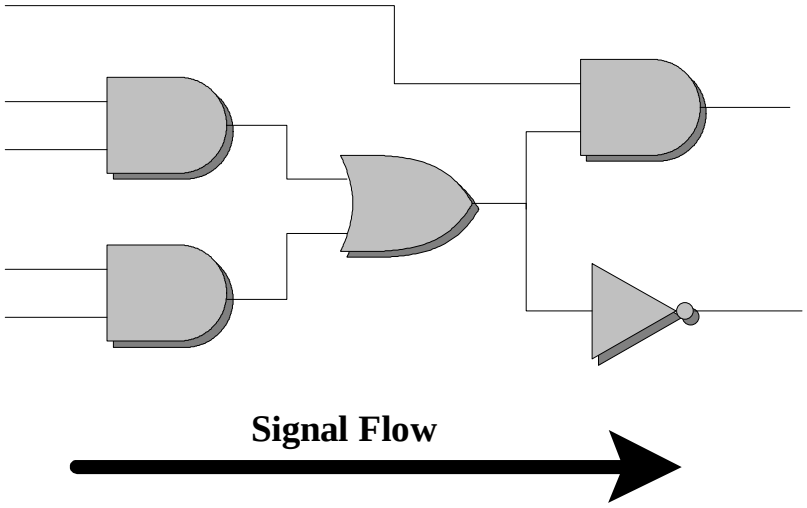


Figure 1-8. A Simple Logic Diagram.

In addition to the types already mentioned, there are many other types of gates found in digital design. The most common of these are Nand, Nor, Buffer, and Exclusive Nor. In fact, Nand and Nor gates are far more common than And and Or gates. Nand and Nor gates are identical to And and Or gates, except the outputs are inverted. A Buffer is a non-inverting amplifier, and an Exclusive Nor is an Exclusive Or with an inverted output. The symbols used for these gates are illustrated in the following diagram.

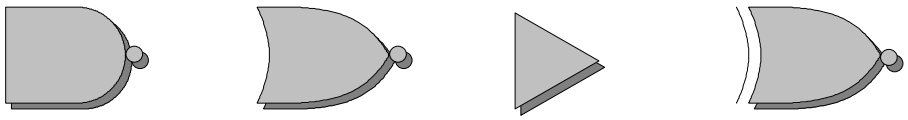


Figure 1-9. Additional Logic Symbols.

1.5 Combinational and Sequential Circuits

A circuit consisting of a combination of gates with no feedback loops or circular dependencies is known as a combinational circuit. Such a circuit implements a *n*-input, *m*-output Boolean function with no memory elements. To add memory functions to a circuit it is necessary to add feedback loops as illustrated in Figure 1-10. Circuits containing memory elements are known as Sequential Circuits.

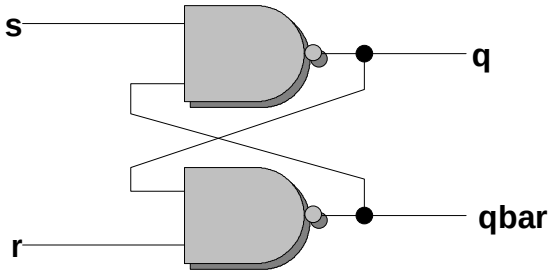


Figure 1-10. An RS Flip-Flop.

The above circuit is an RS Flip Flop, which can be used to store a single bit of information. Setting input s to zero and input r to 1 will cause q to become 1 and qbar to become zero. Leaving r at 1 and setting s to 1 will cause q and qbar to retain their values. Setting s to 1 and r to zero will cause q to become zero and qbar to become 1. Again, leaving s at 1 and setting r to 1 will cause q and qbar to retain their values. Setting both s and r to zero may cause the circuit to become unstable. In particular, setting both inputs back to 1 will cause both q and qbar to oscillate between 1 and 0.

The RS flip-flop can be used as a basis for constructing other types of flip-flops. For example, a D flip-flop can be constructed as illustrated in Figure 1-11. The D flip-flop retains the last value appearing on its single input.

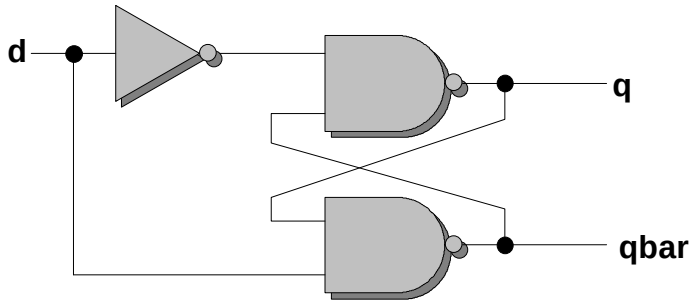


Figure 1-11. A D Flip-Flop.

To be useful, the D flip-flop is usually enhanced with a clock signal, so the state of the flip-flop can change only when the clock is active. Figure 1-12 illustrates such a flip-flop.

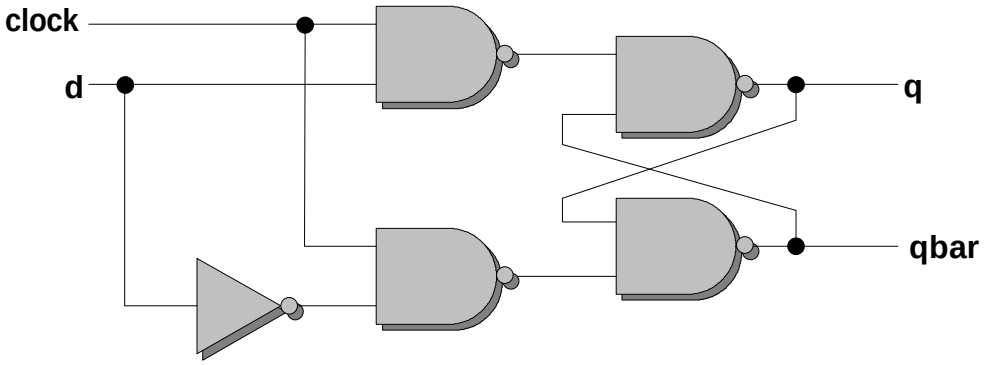


Figure 1-12. A Clocked D Flip-Flop.

Two additional types of flip-flops that are commonly used are JK flip-flops and T flip-flops. A JK flip-flop is identical to an RS flip-flop, except that setting both inputs to zero causes the q and qbar outputs to toggle. (Toggling is the term for changing the current state to its opposite value.) The T flip-flop has a single input which causes the output to toggle when it is set to 1. When the output is zero, the T flip-flop retains its current state. (Strictly speaking, such a flip-flop is known as an Active High T flip-flop. An Active Low T flip-flop toggles when its input is zero and retains its value when its input is one. The terms Active High and Active Low can also be used to refer to specific inputs.)

Although the D, JK, and T flip-flops can be implemented using simple gates, for efficiency they are usually implemented as special circuits. Because flip-flops are commonly used circuit elements, special symbols are used to represent them. Figure 1-13 gives the symbols for the four types of flip-flops discussed in this section.

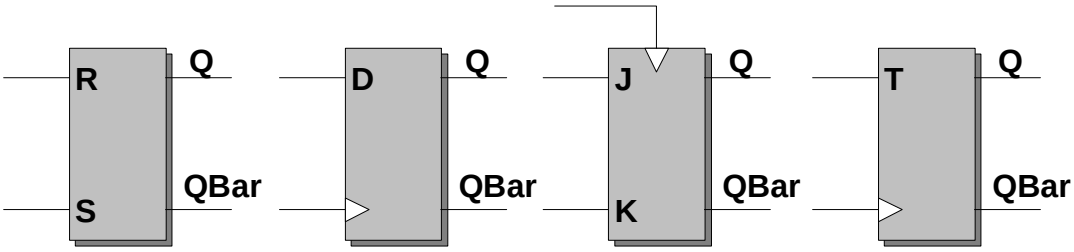


Figure 1-13. Logic Symbols for Flip-Flops.

In the above diagram, a white triangle is used to identify the clock input of the D, JK, and T flip-flops. In CMOS design, which is one of the most common implementation techniques used today, flip-flops must have two clock inputs and the two clocks must be the inverse of one another. Figure 1-14 is an example of a CMOS D flip-flop.

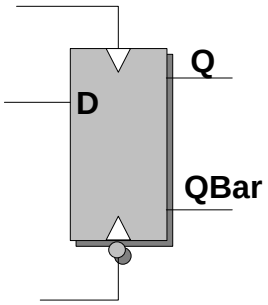


Figure 1-14. A CMOS Flip-Flop.

1.6 Synchronous and Asynchronous Circuits

Although the clock input of the D flip-flop is designed to synchronize the state-change of the flip-flop with the changes in the clock, such synchronization is

not guaranteed. If the D input changes several times while the clock is active, these changes will be transmitted through the gate to the output. If the flip-flop is embedded in a combinational circuit as illustrated in Figure 1-15, this behavior can cause problems.

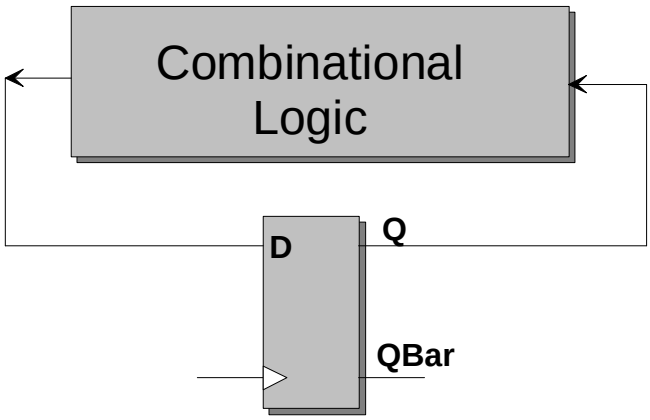


Figure 1-15. Sequential Circuits.

While the clock is active, the input of the D flip-flop is transmitted to the output Q. This will generally cause changes in the combinational logic, which may cause the D input to change. As long as the clock remains active, this change will be transmitted through the flip-flop to the output. This will cause the D input to return to its original value. The net result is that the Q output of the flip-flop will oscillate, and eventually be set to an unpredictable value when the clock becomes inactive. To synchronize the D flip-flop it is necessary to use two D flip-flops in a master-slave configuration, as illustrated in Figure 1-16.

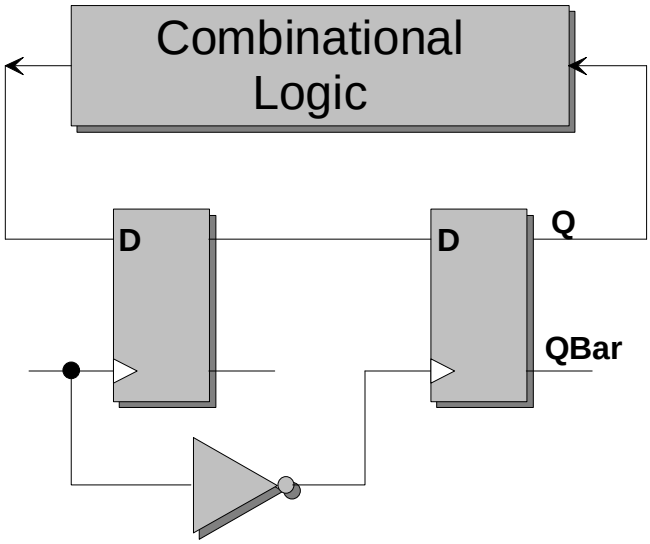


Figure 1-16. A Synchronous Sequential Circuit.

Circuits such as that in the preceding diagram are called synchronous circuits, because the changes in the master-slave flip-flop are synchronized with the clock.

Sequential circuits containing unclocked flip-flops are called asynchronous circuits. It is possible for a circuit to contain both clocked and unclocked flip-flops. If the unclocked elements are not contained in a logic loop, the circuit will behave as if it were a synchronous circuit, otherwise the circuit is considered to be asynchronous.

1.7 Edge Triggered Flip-Flops and Registers

The master-slave scheme illustrated in Figure 1-16 shows one method of synchronizing flip-flop changes with the clock. Synchronizing changes in a flip-flop with changes in one or more inputs is called Edge Triggering. RS and D flip-flops are not generally assumed to be edge triggered, and hence require synchronizing mechanisms such as that given in Figure 1-16. On the other hand JK and T flip-flops are generally assumed to be edge triggered. Edge triggering is necessary for these flip-flops to guarantee that the flip-flop toggles in a controlled manner.

Suppose a circuit is designed using active high T flip-flops. When the input of the flip flop is set to one, it must toggle once and only once. To toggle the flip-flop again, it is necessary to set the input back to zero, and set it to one once again. In other words, the toggling of the flip-flop must be triggered by the change in the input from zero to one. If the change is triggered by the value of the input, the output of the flip-flop will oscillate until the input is reset to zero.

Most edge-triggered flip-flops are clocked, with all changes synchronized with the change in the clock. For such flip-flops it is necessary to hold the other inputs constant while the clock is active. If these inputs change while the clock is active, the behavior of the flip-flop can be unpredictable.

When several flip-flops are combined to hold a multi-bit value, the result is a register. As Figure 1-17 illustrates, a register has a single, multi-bit input called a bus. (The number next to the diagonal line gives the number of bits in the input.) At a minimum, a register has a load signal which is used to copy a value from the input bus to the register. The register may also have a separate clock. The register will usually have a multi-bit output, and may have several additional control inputs. In particular there may be inputs which shift the register contents to the right or left, or inputs to increment and decrement the register value. Such inputs need to be edge triggered to produce predictable behavior. Registers are assumed to be synchronous unless otherwise specified.

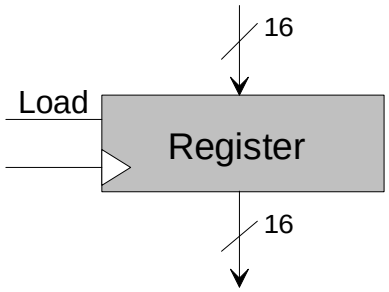


Figure 1-17. The Register Symbol.

1.8 Tristate Logic

A simple gate such as a NOT gate has two states, output zero and output one. Electrically, a zero output is generated by connecting the output to the ground terminal of the power supply. Similarly the one output is generated by connecting the output to the positive terminal of the power supply. Some gates allow the output to have a third state, which is generated by disconnecting the input from both power terminals. Gates with this capability are called tristate gates, and the third state is called the high-impedance state. Unlike the zero and one states, the high-impedance state is not a true logic state. A connection in the high-impedance state will, initially, appear to be in either the one or zero state. After a few milliseconds the connection will drift, usually toward the zero state. Tristate gates can be used to create high-speed multiplexors with a minimal amount of logic. Figure 1-18 illustrates the symbols used to represent tristate buffers. Tristate buffers may be inverting or non-inverting.

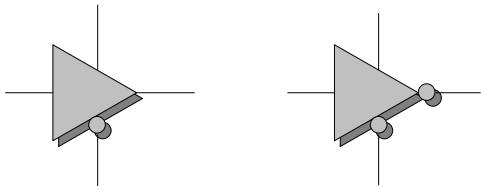


Figure 1-18. Tristate Buffers.

These two gates are the CMOS versions, which require both the true and inverted values of the control signal. Another type of tristate gate is the transmission gate. The transmission gate is functionally identical to the non-inverting tristate buffer, but does not amplify the signal passing through it. Although transmission gates are simpler than tristate buffers, they are less frequently used than tristate buffers, because the signal tends to degrade after passing through several transmission gates. The symbol for the transmission gate is illustrated in Figure 1-19. This is the CMOS version, which requires both the true and inverted values of the control signal.

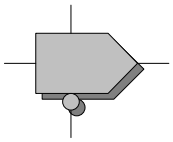
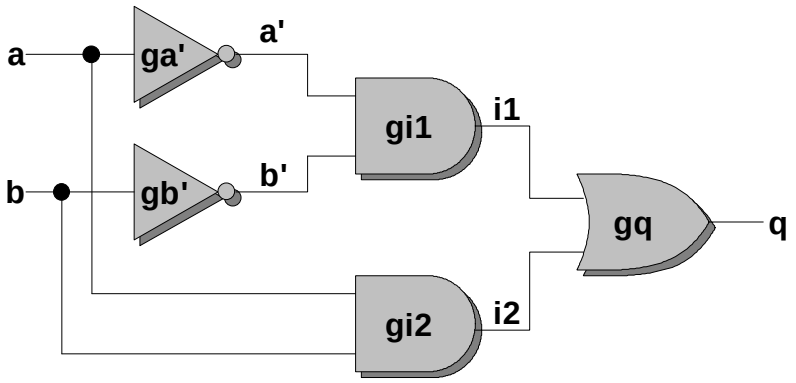


Figure 1-19. The Transmission Gate.

1.9 Specification Languages

Before a circuit can be simulated, it must be converted to machine-readable form. There are many commercial tools that can be used to draw a logic diagram and simulate it. The drawing tools are usually called *schematic capture* tools, and are generally separate from the simulation programs. Although schematic capture tools are popular, the process of creating a schematic is slow, and complex schematics may difficult to create and difficult to read. As an alternative to schematic capture, there are a number of tools that allow circuits to be specified textually. Over the years, a number of standard languages have been developed, two of which are *VHDL* and *VERILOG*. VHDL was created by the U.S. Department of Defense, while VERILOG was created by Cadence Corporation. Other, simpler languages have been developed for educational purposes. One of these is the *FHDL* language, which will be used extensively in this book. Figure 1-20 gives an example of a circuit, and its FHDL description.




```
Test1:      circuit
            inputs      a,b
            outputs     q
gq:         or          (i1,i2),q
gi2         and         (a,b),i2
gi1:        and         (aprime,bprime),i1
gaprime:    not         a,aprime
gbprime:    not         b,bprime
            endcircuit
```

Figure 1-20. Text-Based Circuit Descriptions.

Before a circuit can be simulated, it must be translated from text into data-structures. The mechanisms for doing this are part of the theory of parsing and programming languages, and are beyond the scope of this book. We will assume that parsing software is available for the FHDL language, and that it produces data structures similar to those illustrated in Figure 1-21 and Figure 1-22. As these tables illustrate, it is necessary to treat each gate and each net in the circuit as a separate entity.

Gate Table				
Index	*Name*	*Type*	*Inputs*	*Outputs*
0	gq	or	3,4	2
1	gi2	and	0,1	4
2	gi1	and	5,6	3
3	gaprime	not	0	5
4	gbprime	not	1	6

Figure 1-21. Machine Readable Gate Descriptions.

Net Table				
Index	*Name*	*Type*	*Fanout*	*Value*
0	a	PI	1,3	0
1	b	PI	1,4	0
2	q	PO		0
3	i1		0	0
4	i2		0	0
5	aprime		2	0
6	bprime		2	0

Figure 1-22. Machine Readable Net Descriptions.

1.10 Summary

The first step in creating a simulator is a thorough understanding of the objects to be simulated. In logic simulation the objects are gates and nets. In some cases more complex objects such as registers, multiplexors and adders can be simulated directly. In other cases it is necessary to model these objects as collections of simpler gates. Regardless of the types of gates that are supported, it is necessary to have a thorough understanding of the function of each gate.

Although gates are the active elements of a circuit, it is important not to ignore the nets used to connect them. Since gates have no value, simulator output consists of a set of net values. It is necessary to model the primary inputs and outputs of a circuit using some mechanism that allows external input and output. It may also be necessary to have some mechanism for internally generating primary input values.

As later chapters will explain, the primary difficulty in logic simulation is not simulating gates or nets, but the scheduling of such simulations. Nevertheless, the gate simulation code is a major portion of each simulator. The following chapters will explain how this code is scheduled to produce a correct simulation of a circuit.

1.11 Exercises

1. Prove the identities given in Figure 1-2. (This can be done by evaluating both sides of the equations for all combinations of input values.)
2. Evaluate the following Boolean equations for all 16 input combinations.

$$q = a(b+c)+cd$$
$$r = (a+b)(c+d)$$
$$t = ab+bc+cd+ad$$
$$u = abc+cd(ab+cd)$$

3. Find a minimal Boolean expression for the following function.

Inputs	Val	Inputs	Val	Inputs	Val	Inputs	Val
0,0,0,0	0	0,1,0,0	0	1,0,0,0	0	1,1,0,0	0
0,0,0,1	0	0,1,0,1	0	1,0,0,1	0	1,1,0,1	0
0,0,1,0	0	0,1,1,0	0	1,0,1,0	0	1,1,1,0	1
0,0,1,1	1	0,1,1,1	1	1,0,1,1	1	1,1,1,1	1

4. Find a minimal Boolean expression for the following function.

Inputs	Val	Inputs	Val	Inputs	Val	Inputs	Val
0,0,0,0	1	0,1,0,0	1	1,0,0,0	1	1,1,0,0	1
0,0,0,1	1	0,1,0,1	0	1,0,0,1	0	1,1,0,1	1
0,0,1,0	0	0,1,1,0	0	1,0,1,0	0	1,1,1,0	1
0,0,1,1	0	0,1,1,1	0	1,0,1,1	0	1,1,1,1	1

5. Find the prime implicants of the following function.

Inputs	Val	Inputs	Val	Inputs	Val	Inputs	Val
0,0,0,0	0	0,1,0,0	0	1,0,0,0	1	1,1,0,0	0
0,0,0,1	1	0,1,0,1	1	1,0,0,1	1	1,1,0,1	1
0,0,1,0	0	0,1,1,0	0	1,0,1,0	1	1,1,1,0	1
0,0,1,1	0	0,1,1,1	0	1,0,1,1	1	1,1,1,1	1

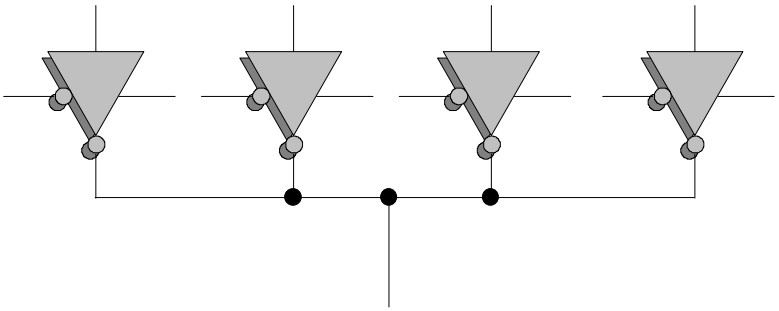
6. Give the logic diagram for each of the expressions listed in Exercise 2.
7. Give the logic diagram for the functions of Exercises 3 and 4.

8. Complete the following Table for the RS Flip-Flop.

Inputs				Outputs	
R	**S**	**Q_{old}**	**$Qbar_{old}$**	**Q_{new}**	**$Qbar_{new}$**
0	0	0	0		
0	0	0	1		
0	0	1	0		
0	0	1	1		
0	1	0	0		
0	1	0	1		
0	1	1	0		
0	1	1	1		
1	0	0	0		
1	0	0	1		
1	0	1	0		
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		

9. Complete a table such as that shown in Exercise 8 for a clocked D flip-flop.

10. Using Tristate gates, it is possible to construct a 4x1 multiplexor using only 4 gates. Show how to construct a similar circuit using And, Or, and Not gates. Note that the control inputs of this circuit are of the form (1,0,0,0), (0,1,0,0), (0,0,1,0), and (0,0,0,1) rather than (0,0), (0,1), (1,0), (1,1). You may assume that the complements of all control inputs are already available.



References

[Bool54] Boole, G. *An Investigation of the Laws of Thought*, New York, Dover Pub. 1954.

[Shan38] Shannon, C. E. "A Symbolic Analysis of Relay and Switching Circuits." Trans. of the AIEE, Vol. 57, (1938), 713-23.

[KARN53] Karnaugh, M. "A Map Method for the Synthesis of Combinational Logic Circuits," *Trans. AIEE, Comm. and Electronics*, Vol. 72, Part I (November 1953), 593-99.

[Quin52] Quine, W. V., "The Problem of Simplifying Truth Functions," *Am. Math. Monthly*, Vol. 59, No. 8, (October 1952), 521-31.

[MCCL56] McCluskey, E. J., Jr. "Minimization of Boolean Functions," *Bell System Tech. J.*, Vol. 35, No. 6, (November 1956), 1417-44.