

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.

The way that ones and zeros are represented in a digital circuit depends on how the circuit is constructed. Boolean algebra was first used to analyze relay circuits. In a relay circuit, a one is represented by a current flow and a zero is represented by no current. This corresponds to the open and closed state of a switch. In today's circuits, it is more common to represent ones and zeros as high and low voltages. These voltages are generated by opening and closing switches so as to connect a conductor to the power source or to ground. These high and low voltages produce different levels of charge on capacitors in the circuit. These charge levels are also used to represent ones and zeros.

There is no fixed assignment between high and low voltages and ones and zeros. Sometimes a high voltage is used to represent a one, and sometimes a low voltage is used for that purpose. In most cases we will assume that a high voltage represents a one, but we will choose the opposite point of view when it is convenient to do so.

Design Automation: Logic Simulation

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 \bar{a} . The expression a' is generally referred to as the *complement* of 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 combining AND, OR, and NOT functions. To see that this is true, consider a Boolean function f with n inputs. Suppose f has the value one for the input combination $C = (a_1 = b_1, a_2 = b_2, \dots, a_n = b_n)$. We first create the term $a_1 a_2 \dots a_n$, in which we AND all n input variables together. Then if $b_i = 0$ for some $1 \leq i \leq n$, we replace a_i with its complement, a_i' . We make this replacement for every b_i that is equal to zero, and leave the remaining input variables uncomplemented. The resulting term will have the value 1 for the input combination C and the value 0 for every other input combination. We go through the input combinations for which f has the value 1, and create a term for each. We then OR these terms together to obtain a Boolean expression for the function. A term that contains every input variable either complemented or uncomplemented is called a *minterm*.

Consider the following example.

a_1	a_2	a_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

For this function we have four input combinations that are equal to 1, so we must create four minterms. These are $a_1' a_2' a_3'$, $a_1' a_2 a_3$, $a_1 a_2' a_3'$, and $a_1 a_2 a_3$. We OR these together to create an expression for $f = a_1' a_2' a_3' + a_1' a_2 a_3 + a_1 a_2' a_3' + a_1 a_2 a_3$. For each Boolean function f , there are many Boolean expressions that represent it. The above expression is not necessarily the simplest expression for f .

The laws of Boolean algebra can be used to simplify Boolean expressions. The following table lists some of these laws. 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	$a0 = 0a = 0$ $1 + a = a + 1 = 1$
Commutative	$a + b = b + a$ $ab = ba$
Associative	$a(bc) = (ab)c$ $a + (b + c) = (a + b) + c$
Distributive	$a(b + c) = ab + ac$ $a + bc = (a + b)(a + c)$
Demorgan's Laws	$(a + b)' = a'b'$ $(ab)' = a' + b'$
Absorption	$ab + a = a$ $(a + b)a = a$

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

As mentioned above, 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 NOT is applied first, then all AND operations are performed, followed by the OR operations. The primes in the first equation apply only to the immediately preceding variable. The form of these equations is called *sum-of-products* form, which is the usual way of specifying Boolean functions.

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 and 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

Design Automation: Logic Simulation

is not. The term abc is unnecessary, because whenever abc is equal to 1, ab will also be equal to 1. (See the absorption law above.) 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 *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 an 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. If an implicant is not contained in or implied by another implicant, then it is called a *prime implicant*. For the function q , ab is a prime implicant, but abc is not. The process of minimization attempts to obtain an expression with as few implicants as possible, and implicants with as few variables as possible. A minimized expression will consist of nothing but prime implicants. Unfortunately, the minimization process is known to be NP-Complete. Nevertheless, there are several approximation algorithms that produce acceptable results in a reasonable amount of time.

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.

		C		B	
		00	01	11	10
0		1	0	0	1
1		1	0	0	0

Figure 1-3. A Simple Karnaugh Map.

The function corresponding to this Karnaugh map is $a'b' + b'c'$. Xxx shows the correspondence between variables and areas of the Karnaugh map. Prime implicants can be found by maximizing the area under consideration, gathering as many ones together as possible. Smaller areas lead to prime implicants with more variables. The first six maps show the areas corresponding to the individual variables, while the last two show intersections of two areas.

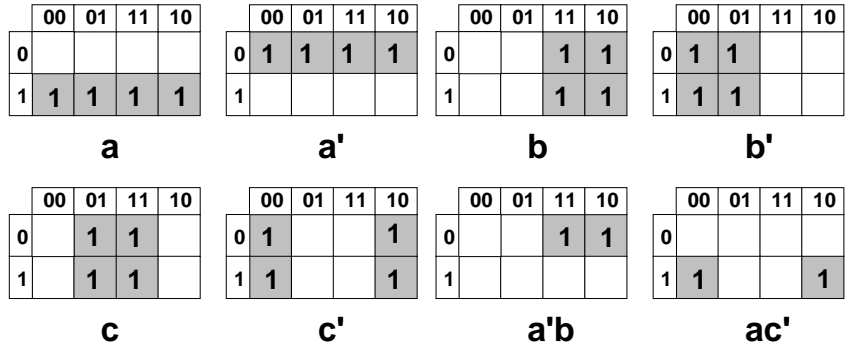


Figure 1-4 illustrates a 4-input Karnaugh map.

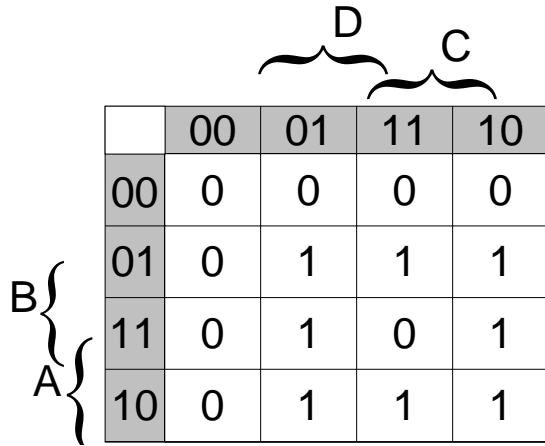


Figure 1-4. A Four-Variable Karnaugh Map.

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.

Xxx shows how the areas of a 4-variable Karnaugh map correspond to various prime implicants. Note that the 4-variable map wraps in both the horizontal and vertical directions. As with the 3-variable map, the prime implicants with more than one variable correspond to intersections of the single-variable areas.

Design Automation: Logic Simulation

	00	01	11	10
00				
01				
11	1	1	1	1
10	1	1	1	1

a

	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11				
10				

a'

	00	01	11	10
00				
01	1	1	1	1
11	1	1	1	1
10				

b

	00	01	11	10
00	1	1	1	1
01				
11				
10	1	1	1	1

b'

	00	01	11	10
00			1	1
01			1	1
11			1	1
10			1	1

c

	00	01	11	10
00	1	1		
01	1	1		
11	1	1		
10	1	1		

c'

	00	01	11	10
00		1	1	
01		1	1	
11		1	1	
10		1	1	

d

	00	01	11	10
00	1			1
01	1			1
11	1			1
10	1			1

d'

	00	01	11	10
00			1	1
01				
11			1	1
10				

bc

	00	01	11	10
00	1			1
01				
11				
10	1			1

b'd'

	00	01	11	10
00				
01				
11		1	1	
10				

abd

	00	01	11	10
00				1
01				
11				
10				

a'b'cd'

There are versions of the Karnaugh map for 5 and 6 variables, but for functions with more than 4 variables, the Quine-McClusky method is more effective. 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. Variable names are not important during the minimization process, the minterms can be represented as strings of ones and zeros. The algorithm builds a complete collection of implicants by combining minterms and other implicants. Let us assume that we are minimizing an n -input function. The minterms of the function represent the complete set of n -variable implicants. During the first step of the algorithm, pairs of minterms are combined to form $(n-1)$ -variable implicants. The next step combines pairs of $(n-1)$ -variable implicants to form $(n-2)$ -variable implicants. Any implicant that has been combined with another implicant is tagged as being a non-prime implicant. Once the algorithm reaches a point where no new implicants can be created, the set of untagged implicants constitutes the complete set of prime implicants for the function. Two k -variable implicants can be combined into a $(k-1)$ -variable implicant if they are identical in all bit positions except one. The $(k-1)$ -variable implicant is identical to the two k -variable implicants with the differing position replaced by an x, which represents “don’t care.” If the k -variable implicants contain x’s, they must not appear in the differing position. For example, consider the two minterms 01110 and 01100. These can be combined into 011x0. The two

minterms 01010 and 00110 cannot be combined, because they differ by two bit positions. The implicants 01xx0 and 01xx1 can be combined because the x's are in the same position and they differ only in the last position.

When comparing implicants, it is not necessary to compare each k -variable implicant to every other k -variable implicant. If implicant I_1 has j ones and differs from I_2 by only one bit position, then I_2 must have either $j-1$ ones or $j+1$ ones. If we sort the minterms into ascending order by the number of ones, and break them into groups based on the number of ones, then we need to compare the elements of group m with the elements of group $m+1$, but no other comparisons are necessary. Since an x can only match another x, we can group implicants with x's by the position of the x's and the number of ones.

Consider the example of xxx. The minterm 0011 can be combined with 0111 and 1011 giving 0x11 and x011. The minterm 1100 can be combined with 1101 and 1110 giving 110x and 11x0. All four minterms 0111, 1011, 1101, and 1110 can be combined with 1111 to give x111, 1x11, 11x1, and 111x. Technically, each of these new eight implicants belongs in its own category, because there are no two implicants with the same number of ones and x's in the same position. We can combine x011 and x111 to form xx11, 0x11 and 1x11 to form xx11, 11x0 and 11x1 to form 11xx, and 110x and 111x to form 11xx. Eliminating the duplicates we have xx11 and 11xx. These two cannot be combined with anything, so they are prime implicants of the function. All of the other implicants will have been flagged as non-prime, because they can be combined with another implicant.

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

Figure 1-5. Implicant Table.

Once all prime implicants of the function have been found, it is necessary to select a collection of them that covers all the minterms of the function. The coverage information is recorded along with each implicant so that the minterms can be checked off as the prime implicants are selected. In the example of xxx, x011 covers 0011 and 1011, while xx11 covers 1111, 1011, 0111, and 0011. In general, the minterm coverage of the prime implicants will overlap, so there may be several ways to select a set of prime implicants that covers all minterms.

The selection of an appropriate minimal set of prime implicants is a complex process that is known to be NP-complete. However there are several heuristics that will produce a minimal Boolean expression for the vast majority of functions. One heuristic is to select the prime implicant that will increase minterm coverage by the largest amount. In case of a tie, the prime implicant with the fewest variables will be chosen. This will tend to avoid overlapping prime implicants until it is impossible to do otherwise.

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.

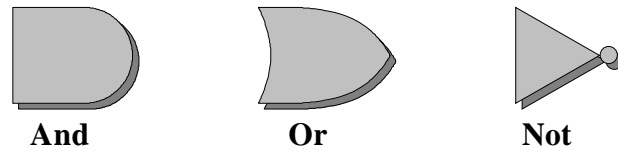


Figure 1-6. Basic Logic Symbols.

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

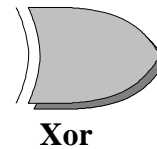


Figure 1-7. The Exclusive Or Function.

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*.

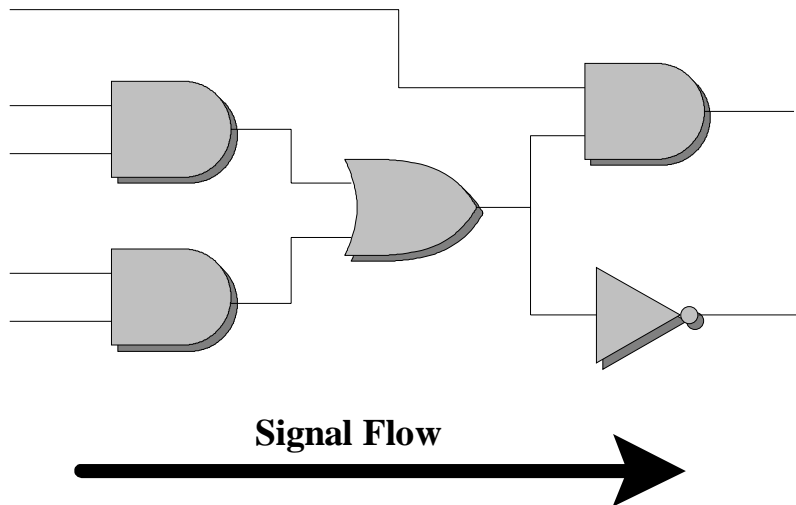


Figure 1-8. A Simple Logic Diagram.

Other types of gates that are encountered in digital design 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 And and Or gates with inverted outputs. 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 given in xxx.

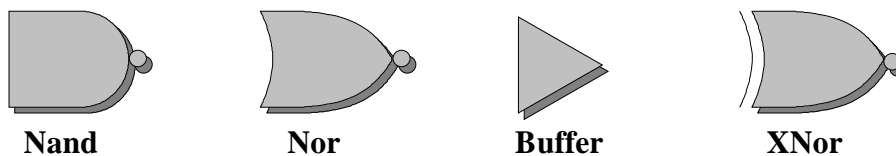


Figure 1-9. Additional Logic Symbols.

1.5 Combinational and Sequential Circuits

A circuit such as that of xxx, with no feedback loops or circular dependencies is known as a *combinational circuit*. Such a circuit implements a n -input, m -output Boolean function. The behavior of the circuit depends only on its inputs, not on the previous history of the circuit. For many circuits it is necessary to have memory elements that store information about previous inputs to the circuit. A register is an example of such a circuit. In most cases, memory elements are constructed using feedback loops such as that illustrated in Figure 1-10. Circuits containing memory elements are known as *sequential circuits*, because the output of the circuit depends on a *sequence* of inputs, not just the current inputs.

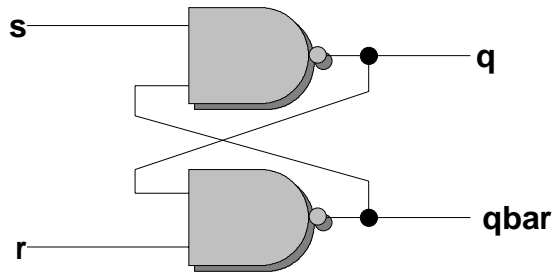


Figure 1-10. An RS Flip-Flop.

Xxx shows an RS Flip Flop, which can be used to store a single bit of information. The *s* input is used to *set* the value of *q* (set the value to 1) while the *r* input is used to *reset* the value of *q* (set the value to 0). Setting input *s* to zero and input *r* to one will cause *q* to become one and *qbar* to become zero. Setting *s* to one and *r* to zero will cause *q* to become zero and *qbar* to become one. Setting both *r* and *s* to one will cause *q* and *qbar* to retain their values. Setting both *s* and *r* to zero may cause the circuit to become unstable. Setting both inputs back to one will cause both *q* and *qbar* to oscillate between one and zero.

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.

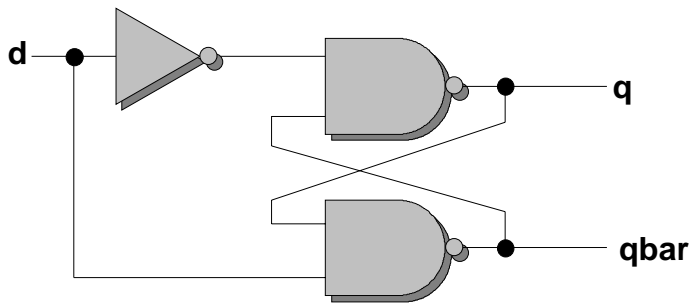


Figure 1-11. A D Flip-Flop.

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.

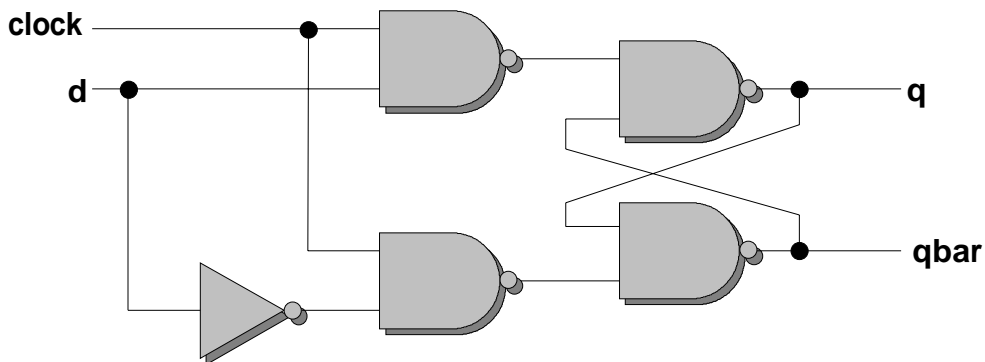


Figure 1-12. A Clocked D Flip-Flop.

Two additional types of flip-flops that are commonly encountered 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, which means that the output values invert. 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.

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.

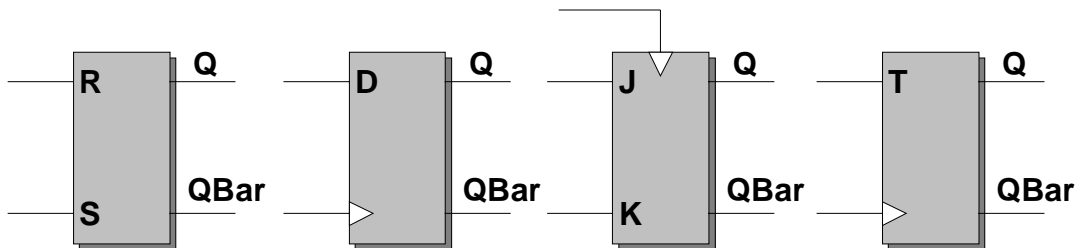


Figure 1-13. Logic Symbols for Flip-Flops.

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.

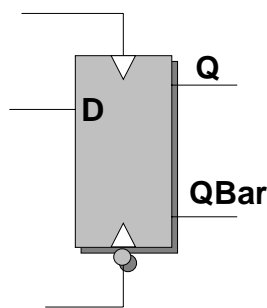


Figure 1-14. A CMOS Flip-Flop.

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

Design Automation: Logic Simulation

embedded in a combinational circuit as illustrated in Figure 1-15, this behavior can cause problems.

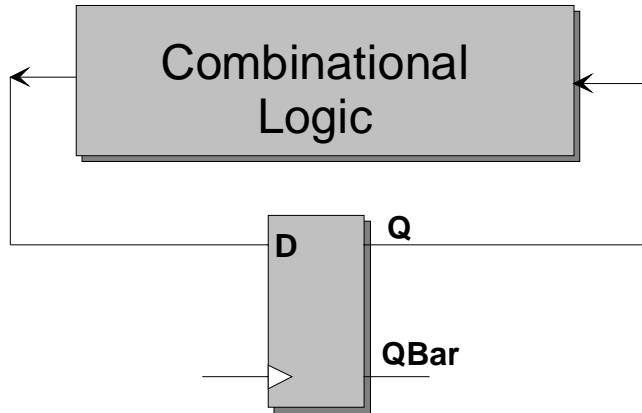


Figure 1-15. Sequential Circuits.

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.

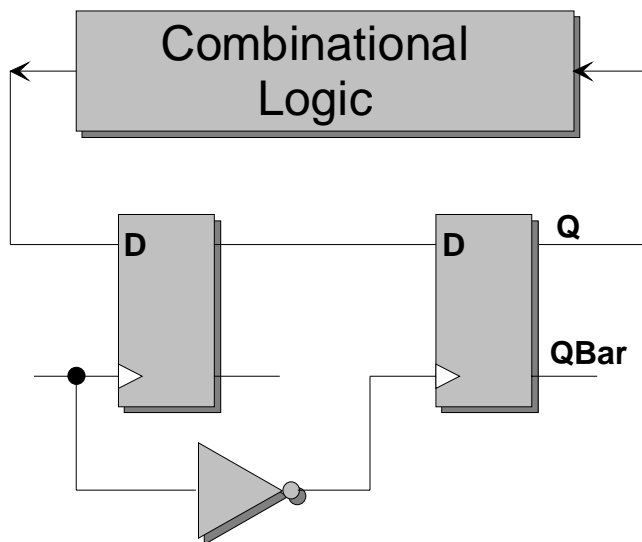


Figure 1-16. A Synchronous Sequential Circuit.

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 unlocked flip-flops are called asynchronous circuits. It is possible for a circuit to contain both clocked and unlocked flip-flops. If the unlocked 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.

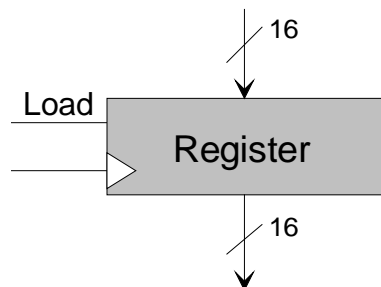


Figure 1-17. The Register Symbol.

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 output 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 multiplexers with a minimal amount of logic. Figure 1-18 illustrates the symbols used to represent tristate buffers. Tristate buffers may be being inverting or non-inverting.

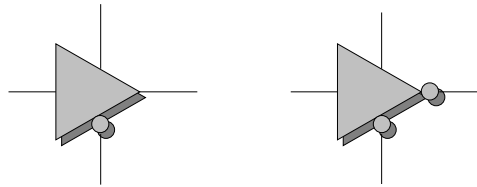


Figure 1-18. Tristate Buffers.

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, 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.

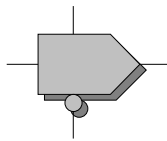
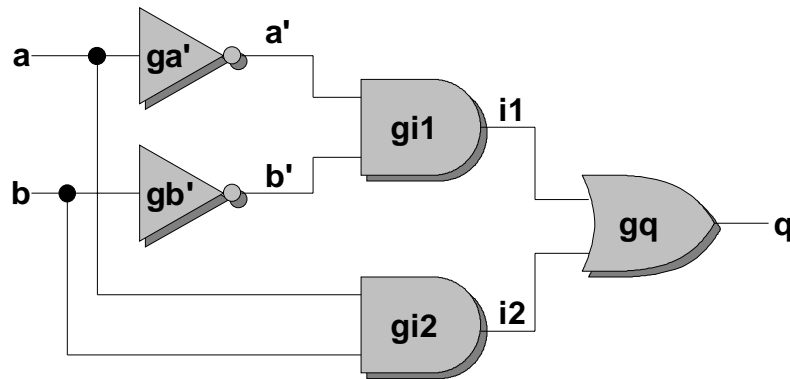


Figure 1-19. The Transmission Gate.

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. Both are now defined by IEEE standards[ref]. Other, simpler languages have been developed for educational purposes. One of these is the Functional Hardware Description Language (*FHDL*), 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 converted into data structures that contain the gates, the connections between them, their types, and their connectivity. 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 parsers are available for FHDL or whatever language we are using, and that they 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				
<i>Index</i>	<i>Name</i>	<i>Type</i>	<i>Inputs</i>	<i>Outputs</i>
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				
<i>Index</i>	<i>Name</i>	<i>Type</i>	<i>Fanout</i>	<i>Value</i>
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, multiplexers 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 reading external inputs and writing external outputs.

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

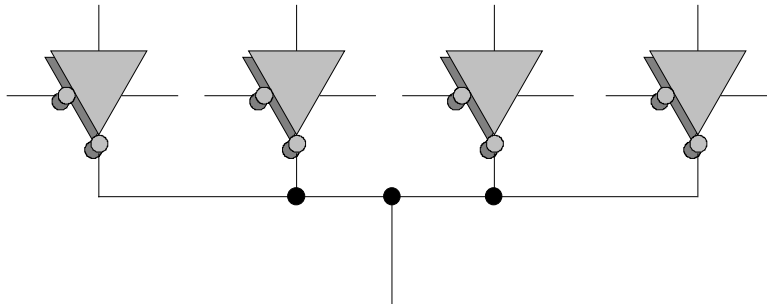
Design Automation: Logic Simulation

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

Inputs				Outputs	
R	S	Q _{old}	Q _{bar} _{old}	Q _{new}	Q _{bar} _{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.



1.1 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.