SUPPLEMENTARY CHAPTER 1

AN INTRODUCTION TO DIGITAL COMPUTER LOGIC

"I keep telling you Gwendolyth, you'll never attract today's kids that way."

S1.0 INTRODUCTION

Many students are curious about the inner workings of the computer. Although understanding the computer's circuitry is not essential to working with computers, doing so is satisfying, for it reduces the mystery of computers; it also eliminates any idea that the computer is a "magical box" to be feared and respected. Instead, you get to see that the computer is actually nothing more than a rather simple collection of digital switches—more like a toy for adults to play with!

Computers are built up from integrated circuits. Each integrated circuit in a computer serves a specialized purpose within the computer. For example, there is an integrated circuit that represents the CPU, another that provides an interface to the external bus, another that manages memory, another that manages DMA (see Chapter 9), and so forth.

Q1

The integrated circuits themselves are made up of transistors, resistors, capacitors, and other electronic components that are combined into circuits. The primary component of interest to us is the transistor. A single integrated circuit may have thousands, or even millions of transistors. The CPU chip in the module shown in Figure S1.1 contains approximately million transistors in an area of less than

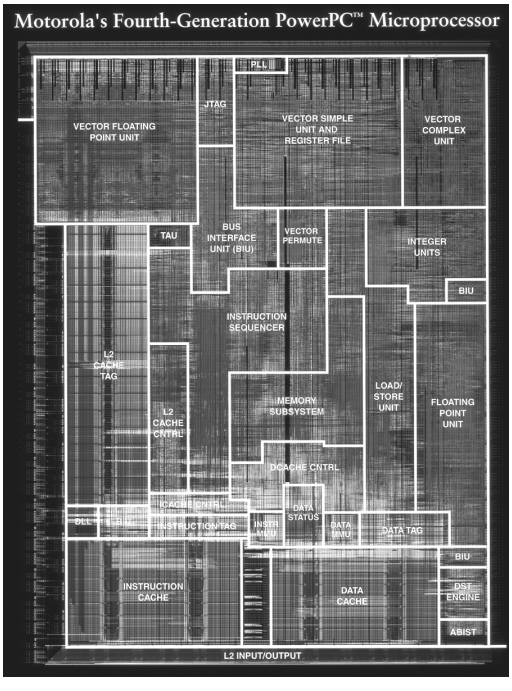
Transistors can act as amplifiers or switches. The transistors in your television set and stereo are used mostly as amplifiers. Except for a few specialized devices such as modems, virtually all the circuitry in computers is digital in nature: the ON and OFF positions of transistor switches serve to represent the 1s and 0s of binary digital circuits. In the computer, these transistor switches are combined to form **logic gates**, which represent values in Boolean algebra. Boolean algebra is the basis for computer logic design and transistors the means for implementation.

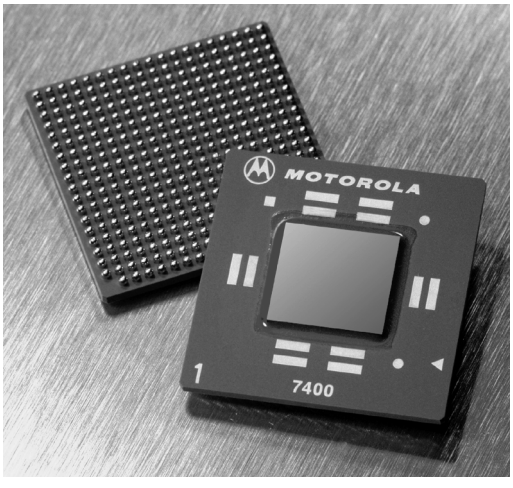
Digital circuits are used to perform arithmetic, to control the movement of data within the computer, to compare values for decision making, and to accomplish many other functions. The digital logic that performs these functions is called **combinatorial logic**. Combinatorial logic is logic in which the results of an operation depend only on the present inputs to the operation. For the same set of inputs, combinatorial logic will always yield the same result. As an example, arithmetic operations are combinatorial. For a given set of inputs and the add operation, the resulting sum will always be the same, regardless of any previous operations that were performed.

Digital circuits can also be used to perform operations that depend on both the inputs to the operation and the result of the previous operation. Digital circuits can store the result or state of an operation and use that result as a factor the next time the operation is performed. Each time the operation is performed, the result will be a function of the present inputs and the previous state of the circuit. Digital logic that is dependent on the previous state of an operation is called **sequential logic**. An example of sequential logic is a counter. Each time the counter operation is performed, the result is the sum of the previous result plus the counting factor. The counter continues to hold the state—in this case, the current count—for use the next time the operation is performed. Computers incorporate both combinatorial and sequential logic.

FIGURE S1.1

The Motorola MPC 7400 PowerPC CPU





S1.1 BOOLEAN ALGEBRA

The digital computer is based on **Boolean algebra**. Boolean algebra describes rules that govern constants and variables that can take on two values. These can be represented in many different ways: true or false, on or off, yes or no, 1 or 0, light or dark, water valve open or shut, to indicate a few possible representations. (Yes, there have been attempts to build hydraulic, water-based computers!)

The rules that govern the ways in which Boolean constants and variables are combined are called **Boolean logic**. There are a number of logical rules, but these can all be derived from three fundamental operations, the operations of AND, OR, and NOT. Boolean logic rules can be described as a formula, or by a **truth table**, which specifies the result for all possible combinations of inputs. Truth tables are the Boolean equivalent to additions and multiplication tables in arithmetic.

The Boolean AND operation can be stated as follows: The result of an AND operation is TRUE if and only if both (or all, if there are more than two) input operands are TRUE. This is shown in the truth table in Figure S1.2. Arbitrarily, we have assigned the value 0 to FALSE and the value 1 to TRUE. This is a normal way of describing Boolean algebra. If you prefer, you could use the value GREEN for true and RED for false, and note that the AND operation says that you can only go if both lights are green. The Boolean symbol for the

AND operation is a center dot: (•). The Boolean equation

$$C = A \bullet B$$

states that the Boolean variable C is true if and only if both A and B are true.

The Boolean OR operation, or more accurately, **INCLUSIVE-OR**, is stated as follows. The result of an INCLUSIVE-OR operation is TRUE if the values of *any* (one or more) of the input operands are true. The truth table for the INCLUSIVE-OR operation is shown in Figure S1.3. The Boolean symbol for the OR operation is a plus sign (+).

Therefore,

$$C = A + B$$

states that C is true if either A or B or both are true.

The Boolean **NOT** operation states that the result is TRUE if and only if the single input operand is FALSE. Thus, the state of the result of a NOT operation is always the opposite state from the input operand. Figure S1.4 shows the truth table for the **NOT** operation. The symbol for the **NOT** operation is a bar over the symbol:

$$C = \overline{A}$$

There is a fourth operation, the **EXCLUSIVE-OR**. The truth table for the EXCLUSIVE-OR operation is shown in Figure S1.5. The symbol for the EXCLUSIVE-OR operation is a plus sign within a circle:

$$C = A \oplus B$$

The EXCLUSIVE-OR operation is used less frequently than the others. It can be derived from the INCLUSIVE-OR, AND, and NOT operations as follows: the result of the EXCLUSIVE-OR operation is TRUE if either A or B is TRUE, but not both. Two ways to express this equivalence are

$$A \oplus B = (A + B) \bullet \overline{(A \bullet B)}$$

which can be read "A or B and not both A and B," or alternatively

$$A \oplus B = (A \bullet \overline{B}) + (B \bullet \overline{A})$$

which reads "either A and not B or B and not A."

FIGURE S1.2
Truth Table for AND Operation

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

FIGURE S1.3
Truth Table for INCLUSIVE-OR Operation

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

FIGURE S1.4
Truth Table for NOT Operation

A	C
0	1
1	0

FIGURE S1.5
Truth Table for EXCLUSIVE-OR Operation

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

It is useful to study this example for practice in the manipulation and reasoning of Boolean algebra.

There are a number of useful laws and identities that help to manipulate Boolean equations. Boolean algebra operations are associative, distributive, and commutative, which means that

$$A + (B + C) = (A + B) + C \quad \text{(associative)}$$
$$A \bullet (B + C) = A \bullet B + A \bullet C \quad \text{(distributive)}$$
$$A + B = B + A \quad \text{(commutative)}$$

These laws are valid for INCLUSIVE-OR, AND, and EXCLUSIVE-OR operations. Perhaps most useful are a pair of theorems called **DeMorgan's theorems**, which state the following:

$$\overline{A + B} = \overline{A} \bullet \overline{B} \text{ and } \overline{A \bullet B} = \overline{A} + \overline{B}$$

These laws and theorems are important because it is frequently necessary or convenient to modify the form of a Boolean equation to make it simpler to understand or to implement.

S1.2 GATES AND COMBINATORIAL LOGIC

Many functions in a computer are defined in terms of their Boolean equations. For example, the sum of two single-digit binary numbers is represented by a pair of truth tables, one for the actual column sum, the other for the carry bit. The truth tables are shown in Figure S1.6. You should recognize the truth table for the sum as the EXCLUSIVE-OR operation and the carry as the AND operation. Similarly, the complement operation that is used in subtraction is just a Boolean NOT operation. These operations are combinatorial. They are true regardless of any previous additions or complements performed.

Combinatorial Boolean logic in a computer is implemented by using electronic circuits called **gates** or logical gates. Gates are constructed from transistor switches and other electronic components, formed into integrated circuits. A *small-scale* integrated circuit may contain half a dozen gates or so for building special Boolean logic circuits. The gates in a CPU are organized into a **very-large-scale integrated (VLSI) circuit** or chip. The drawn representations for logical gates are shown in Figure S1.7.

FIGURE S1.6

Truth Tables for the Sum of Two Binary Numbers

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

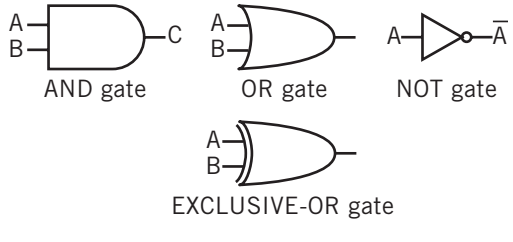
sum

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

carry

FIGURE S1.7

Standard Logic Gate Representations



It is not difficult to manipulate the Boolean algebra to show that combinatorial Boolean logic can be implemented entirely with a single type of gate, appropriately combined. Either

of the two gates shown in Figure S1.8 will fill the bill. The NAND gate is an AND operation followed by a NOT operation. The NOR operation is an INCLUSIVE-OR operation followed by a NOT operation. The small circle is used to indicate the NOT operation.

We can use DeMorgan's theorem to show that a NAND operation is the same as an OR operation performed on inverted inputs. For convenience, the NAND gate may also be drawn in the alternative form shown in the figure. (The same thing can be done with the NOR gate.) The advantage of doing so is shown in Figure S1.9. This logic drawing represents a pair of ANDs followed by an OR. Since two NOTs in succession cancel each other, the pair of circles in succession make it clear what is actually happening. The result in algebraic form is

$$Y = A \bullet B + C \bullet D$$

EXAMPLE

Just for fun, let's consider a practical application for the circuit in Figure S1.9. Figure S1.10 shows the same circuit with one modification: an additional NAND gate has been used to perform a NOT operation, so that only one of the AND gates in the AND-OR combination can be active at a time. If the *select* line is a "1," then the output of the upper NAND gate will reflect the inverse of whatever input is present at A. On the other hand, if the *select* line is a "0," the output of the lower NAND gate will reflect the inverse of whatever is present at B. Since the final NAND gate generates the OR operation of the inverted inputs, only the active AND operation gets passed through to the output. Therefore, Y represents either A or B, depending on the value of the select line.

For obvious reasons, this circuit is called a **selector circuit**. Since it can be used to switch the input back and forth between A and B, it is also sometimes called a **multiplexer**

FIGURE S1.8

NAND and NOR Gate Representations

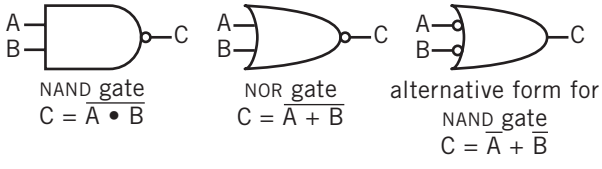


FIGURE S1.9

AND-OR Operation Made up of NAND Gates

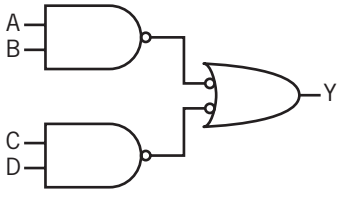
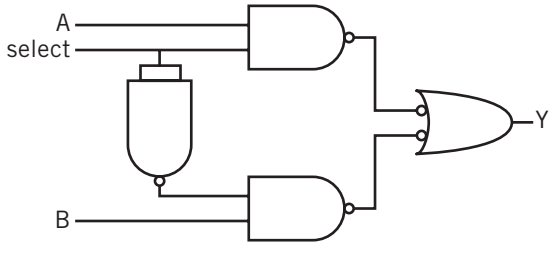


FIGURE S1.10

Selector Circuit



circuit. If we wanted to switch between two bytes of data, we would use eight identical selector circuits, one for each bit. One byte would be placed on the A input, the other on the B input. The same select signal would be connected to the select line on every circuit. What this shows you is that the logic circuits that make up a computer are relatively simple, but they look complicated because so many circuits are required to perform useful work.

■ ■ ■

Another important example of a combinatorial logic circuit is the arithmetic adder. In Figure S1.6 we showed you the truth tables for a simple adder. The NAND logic circuit that produces the desired outputs for a single bit is shown in Figure S1.11. This circuit is called a **half adder**. For practice, you should make sure that you can correlate the circuit to the formulas for a half adder.

The circuit in Figure S1.11 is called a half adder because in most cases a complete adder circuit must also handle a possible carry from the *previous* bit. Figure S1.12 shows a logic circuit for one bit of a **full adder**. To simplify the circuit, we have used the modified half adder enclosed in the dotted line; the use of $\overline{C}$ instead of C reduces the number of gates somewhat. The half adder circuit is represented in Figure S1.12 as a block in this drawing. This approach is a common solution to the problem of making logic drawings readable.

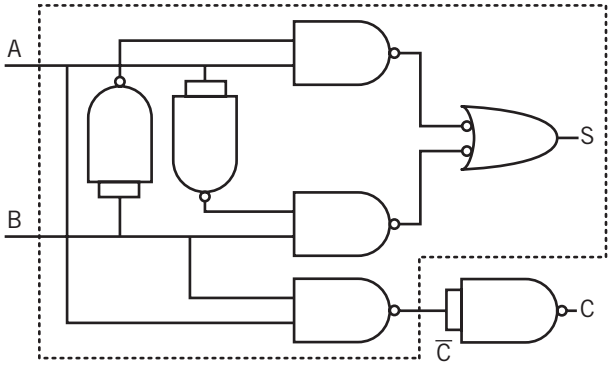
A 32-bit adder would be made up of 32 of these circuits. Because the carry ripples through each of the 32 bits, the adder is called a *ripple adder*. Modern logic designers use some tricks to speed up the adder by reducing the ripple effect of the carry bits, but the basic design of the 32-bit adder in a computer is as you see it here.

S1.3 SEQUENTIAL LOGIC CIRCUITS

Sequential logic circuits are circuits whose output is dependent not only on the input and the configuration of gates that make up the circuit, but on the previous state of the circuit as well. In other words, the state of the circuit is somehow stored within the circuit and used as a factor in determining the new output. The key to sequential logic circuits is the presence of memory within the circuit—not memory as you think of computer memory, but individual bits of memory that form part of the circuit itself. The **state** of the circuit is stored in these memory bits.

FIGURE S1.11

Half Adder



The basic memory element in a sequential logic circuit is called a **flip-flop**. The simplest flip-flop is made up of two NAND logic gates connected as shown in Figure S1.13. This circuit is called a set-reset flip-flop. A similar flip-flop can be built from NOR gates.

Suppose that $\overline{S}$ and $\overline{R}$ are both initially set to 1. Can you determine the two outputs? It turns out that you can't. All you can say is that one of them will be a 0 and the other will be

FIGURE S1.12

Full Adder

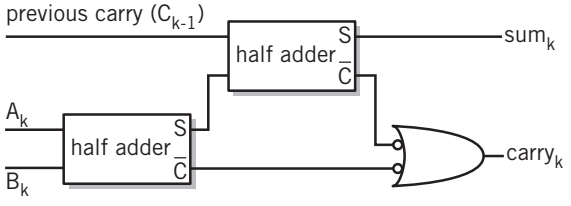
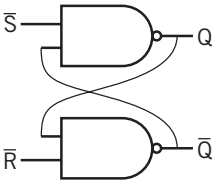


FIGURE S1.13

Set-Reset Flip-flop



a 1. You can see this by assuming the value for one output, determining the other output, and then verifying that everything in the circuit is self-consistent.

For example, assume that the upper output in the figure is a 1. Then both inputs for the lower gate are 1s, and the $\overline{Q}$ output is a 0. This means that one of the inputs to the upper gate is a 0, which verifies that the upper output is a 1. Everything is self-consistent, and the circuit is stable as long as the $\overline{R}$ and $\overline{S}$ inputs remain at 1. (You might need to review the truth table for the NAND gate to convince yourself that the flip-flop works as we claim.)

Now suppose that the $\overline{R}$ input momentarily becomes a 0. This forces the output of the lower gate to a 1. The two upper inputs are now both 1s, so the Q output becomes a 0. The Q output will hold the lower output at 1, even after the $\overline{R}$ input returns to a 1. The flip-flop has switched states. It is now stable in the alternate state to the one that we began with. In other words, the flip-flop *remembers* which input was momentarily set to 0. (One ground rule: the logic surrounding this flip-flop must avoid situations where both $\overline{R}$ and $\overline{S}$ are 0 at the same time.)

There are other types of flip-flops as well. Some are designed to work on the basis of the 1 and 0 levels at the input. These types of flip-flops are sometimes called *latches*. Other flip-flops work on an input *transition*, called an *edge trigger*, the instantaneous change from 1 to 0 at an input, for example. A D flip-flop has a single data input. When the input marked Ck, for *clock*, is momentarily changed to 0 the Q output will take on the value present at the *D* input. The *preset* (P) and *clear* (Clr) inputs are used to initialize the flip-flop to a known value; they work independently of the D and *clock inputs*. A toggle flip-flop switches states whenever the T input momentarily goes to 0.

The equivalent of a truth table for a sequential circuit is called a **state table** or behavior table. The state table shows the output for all combinations of input and previous states. For edge-triggered flip-flops, the clock acts as a control signal. The new output occurs when the clock is pulsed except for preset and clear inputs, which affect the output immediately. The symbols and state tables for several types of flip-flops are shown in Figure S1.14.

Flip-flops of various types have many uses throughout the computer. Registers are made up of flip flops. They hold the results of intermediate arithmetic and logic operations. Flip-flops are used as counters, for the steps of a fetch-execute cycle, and for the program counter. Flip-flops control the timing of various operations. Flip-flops serve as buffers. Static RAM is also made up of flip-flops, although dynamic RAM uses a different storage technique.

EXAMPLE

This example is a simple illustration of the use of both sequential and combinatorial logic in a computer. The text in Chapter 7 points out that the copying of data from one register to another is an essential operation in the fetch-execute cycle. The logic shown in Figure S1.15 represents the essential part of implementing a register copying operation. Flip-flops A_1 through A_4 represent four bits of a register. Flip-flops B_1 through B_4 represent the corresponding bits of a second register. This circuit can be used to copy the data from register A to register B. If the signal marked *copy-A-to-B* is a 1 when the clock is pulsed, data will be copied from A to B. The *copy-A-to-B* signal would be controlled from a circuit that counts the steps in a particular instruction fetch-execute cycle, then turns on the signal when the copy is required.

FIGURE S1.14

Several Types of Flip-flops

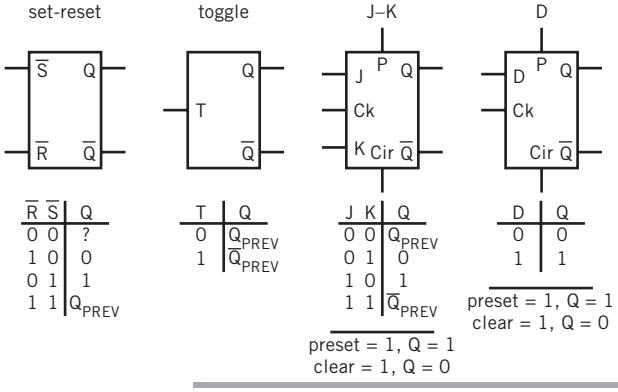
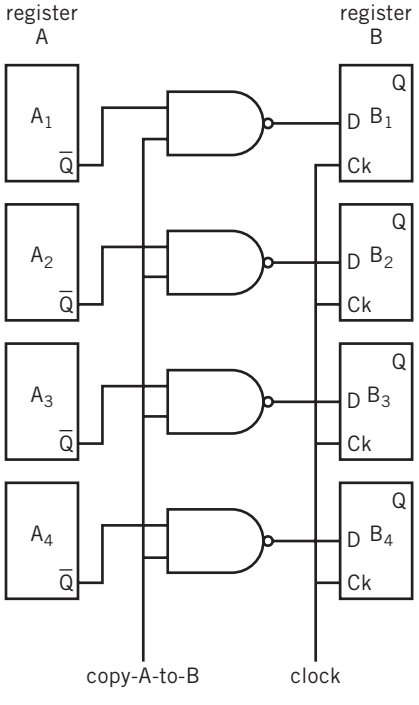


FIGURE S1.15

Logic to Copy Data from One Register to Another



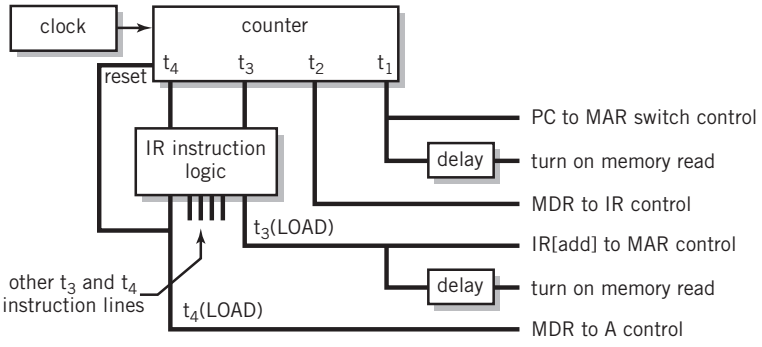
To carry this discussion a step further, consider the simplified hardware implementation of a LOAD instruction, shown in Figure S1.16. For this instruction, the clock pulse is directed to four different lines, each of which carries one of the clock pulses, in sequence, controlled by an instruction step counter. The first line, called t_1 in the diagram, closes the switches that transfer the data from the program counter to the memory address register for the first step of the fetch phase. The same pulse is delayed, then used to activate memory for a READ. The next clock pulse, t_2, connects the memory data register to the instruction register, completing the fetch phase.

Lines t_3 and t_4 will perform different operations depending on the instruction. The combination of bits in the op code portion of the instruction register determine the instruction being performed, and these are used together with the clock lines to determine which switches are closed for the execution portion of the instruction. The remainder of the operation can be seen in the diagram. (The incrementing of the program counter has been omitted in the diagram for simplicity.) The last time pulse is also used to reset the instruction step counter for the next instruction.

As you can see, the basic hardware implementation of the CPU is relatively straightforward and simple. Although the addition of pipelining and other features complicates the design, it is possible, with careful design, to implement and produce an extremely fast and efficient CPU at low cost and in large quantities.

FIGURE S1.16

Simplified Implementation of the Steps in a LOAD Instruction



SUMMARY AND REVIEW

The circuitry in a computer is made up of a combination of combinatorial and sequential logic. Computer logic is based on the rules of Boolean algebra, as implemented with logic gates. Sequential logic uses logic gates to provide memory. The output and state of a sequential logic circuit depends on its previous state as well as the current sets of inputs.

KEY CONCEPTS AND TERMS

AND
Boolean algebra
Boolean logic
combinatorial logic
DeMorgan's theorems
EXCLUSIVE-OR
flip-flop
full adder
gates
half adder
INCLUSIVE-OR
logic gate
multiplexer circuit
NAND
NOR
NOT
sequential logic
selector circuit
state
state table
truth table
very-large-scale integrated (VLSI) circuit

FOR FURTHER READING

Most general computer architecture textbooks have at least a brief discussion of digital logic circuits. Reasonable discussions can be found, for example, in Stallings [STAL05], Patterson and Hennesey [PATT07], and Tanenbaum [TAN05]. More detailed discussions can be found in Lewin [LEW83], Wakerly [WAKE05], or Mano [MANO07]. There are many other excellent choices as well.

READING REVIEW QUESTIONS

S1.1 What are the three fundamental operations in Boolean algebra?

S1.2 What are the two possible results from a Boolean formula?

S1.3 What is a *truth table*? What is the algebraic equivalent?

S1.4 Show the truth table for:

$$C = \overline{A + B}.$$

S1.5 Show the truth table for:

$$D = A + (B \bullet C).$$

S1.6 **a.** Show the DeMorgan's theorem equivalent for:

$$\overline{A + B \bullet C}$$

b. Show the DeMorgan's theorem equivalent for:

$$\overline{A} + \overline{B \bullet C}$$

S1.7 Explain what is meant by "combinatorial logic." What are the electronic circuits that implement combinatorial logic called?

S1.8 Draw four standard combinatorial logic representations.

S1.9 Draw a logic representation for:

$$D = A + B + C$$

S1.10 Draw a logic representation for:

$$D = \overline{A} + B + \overline{C}$$

S1.11 A circuit whose output table takes on the value of one of several inputs is called a _____.

S1.12 What is the "state" of a circuit? Explain the relationship between sequential logic and "circuit state."

S1.13 Explain what a "state table" shows.

S1.14 Carefully explain how the logic in Figure S1.15 works.

EXERCISES

S1.1 **a.** Verify using truth tables that both equivalence equations for the EXCLUSIVE-OR operations are valid.

b. Do the same using DeMorgan's theorem.

S1.2 Show the truth table for the following Boolean equation:

$$Y = A + A \bullet B$$

Look at the result. What general rule for reducing Boolean equations can you deduce from the result?

S1.3 Reduce the following equations to a simpler form

a. $Y = A + 1$

b. $Y = A + 0$

c. $Y = A \bullet 1$

d. $Y = A \bullet 0$

S1.4 Show the truth table for the following Boolean equation:

$$Y = A + A \bullet \overline{B} \bullet C + A \bullet B \bullet \overline{C}$$

S1.5 One easy way to construct a logic gate implementation from a truth table is to recognize that the output is the OR of every row that has a 1 as the result. Each row is the AND of every column that has a 1 in it. Given the following Boolean expression:

$$Y = ((A \bullet \overline{B} + \overline{C}) + B \bullet (\overline{A} \bullet B \bullet C)) \bullet (\overline{B} + C)$$

determine the truth table; then implement the result using NAND gates. You may use three input NAND gates if necessary.

S1.6 Show a selector circuit implementation made up of NOR gates.

S1.7 The sum output from the half adder in Figure S1.11 is implemented from the equation

$$S = A \bullet \overline{B} + \overline{A} \bullet B$$

An alternate representation for the sum is

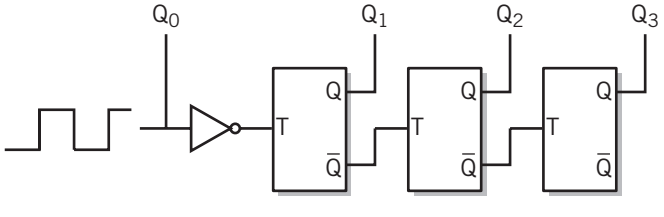
$$S = \overline{((A \bullet B) + \overline{A}) \bullet ((A \bullet B) + \overline{B})}$$

a. Show, using either truth tables or algebraic manipulation, that these two representations are equivalent.

b. Use the latter form to develop a NAND gate implementation that requires only five gates to produce both the sum and carry.

S1.8 A decoder is a combinatorial logic circuit that produces a separate output for every possible combination of inputs. Each output is a 1 only for that particular combination. A decoder with three inputs, A, B, and C, would have eight outputs, for 000, 001, 010, ... Implement a logic decoder circuit for three inputs.

S1.9 Consider the sequential logic circuit shown in the accompanying figure together with an input that consists of an alternating sequence of 0s and 1s as shown. Assume that the initial state of this circuit produces an output that is all 0s. Show the next six output states. In one word, what does this circuit do?



S1.10 Design a circuit that would serve as a four-stage *shift register*. A shift register shifts the input bits one bit at a time, so that the output from each stage represents the previous output from the previous stage.

Queries in Chapter 1

Q1. pls verify x-ref

Q2. pls verify x-ref