Fundamentals of Digital Systems

by

Robert M. Laurie

This text material is a work in progress.
Please contact me with suggestions and corrections:

Robert M. Laurie
Electrical & Computer Engineering
Michigan Technological University
Houghton, Michigan 49931 USA

boblaurie@yahoo.com

Table of Contents

Table of Figures

Chapter 1. Introduction

Digital computers have brought about the information age that we live in today. Computers are important tools for humankind in that they can locate and process enormous amounts of information very quickly and efficiently. They allow us to utilize our mathematical disciplines to the fullest. In one second a computer can perform calculations that would take a person months to do by hand. However, computers are not creative and do only what we tell them. The list of instructions that tells the computer what to do is called a computer program.

System reliability, fast performance, and efficient information storage and retrieval are major factors in the acceptance and use of digital computer systems. The high reliability of computer systems is due largely to the fact that all data is in a digital format. Digital computers are designed such that digital formatted data can be processed quickly and efficiently.

1.1. Digital Logic States

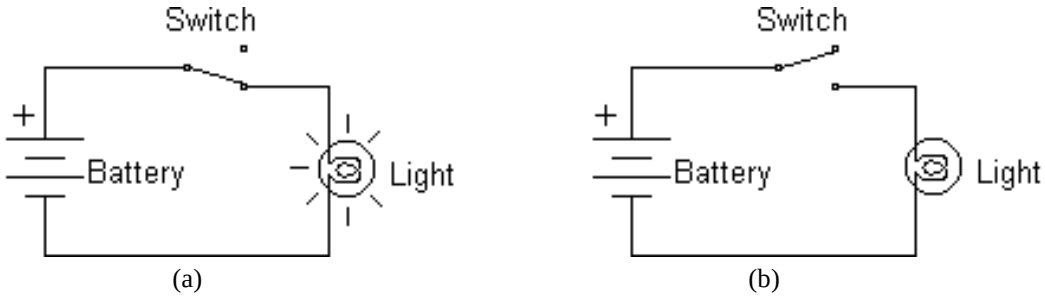
A computer is made up of many digital circuit modules that pass information in the form of *digital signals*. These signals can represent either program instructions or data to be processed. A digital signal can be considered a *logic variable* in that it can have only one of two possible values at any moment in time. These values are called *logic states*. Figure 1.1 describes the common notation for logic states.

Figure 1.1 Common Notation for Logic States

True	On	Closed	Yes	1	High	2 to 5 Volts
False	Off	Open	No	0	Low	0 to 1 Volt

Two possible logic states can be represented by the opening and closing of the switch in the circuit of Figure 1.2. When a logic state is TRUE, the switch is CLOSED and the light goes ON. When a logic state is FALSE, the switch is OPENED and the light goes OFF. A question that can be answered with YES or NO can also be said to have two logic states.

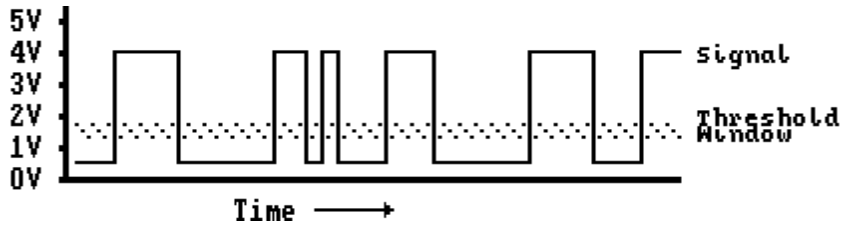
Figure 1.2 Simple Digital Circuit Using a Switch



(a) (b)

Digital circuit inputs and outputs are logic variables whose states are usually represented in binary (base 2) notation as a '1' or '0'. A digital signal is characterized as a time variant signal, which is in either a HIGH or LOW State. Transition time between HIGH and LOW states is minimized which results in a waveform as depicted in Figure 1.3. The primary constraint for a digital signal is whether the signal is above or below a specified threshold window. This threshold window is commonly between 1 and 2 Volts. Any signal which is greater than 2 Volts is considered logic '1' or HIGH state, and any signal which is less than 1 Volt is considered logic '0' or LOW state. Within the threshold window (between 1 and 2 Volts) the Logic State of the signal is undefined.

Figure 1.3 Digital Signal Representation



To perform logic operations, a device is required which can function as a switch with two possible states. Generally, it is desirable to have this switching occur as fast as possible. Electronics is currently the fastest, most compact, and least expensive way to implement a digital switch. Therefore, computer design is usually considered in the realm of electrical engineering.

Digital switches are implemented using an electronic device called a transistor. Using integrated circuit chip technology, it is possible to integrate millions of transistors on a single silicon chip. The rapid advances in digital electronics technology has led to the relatively inexpensive and compact computer systems that we use today.

The binary (Base 2) number system is often used to represent the states of several related logic variables at a specific instant in time. Figure 1.4 describes the binary equivalent for the decimal (Base 10) numbers zero through fifteen. Only two possible values can exist for each binary digit, either '0' or '1'. A single binary digit is called a *bit* and a group of eight bits is called a *byte*. A group of four bits as shown in Figure 1.4, is called a *nibble*.

Figure 1.4 Binary to Decimal Conversion Table (0 through 15)

Binary	Decimal	Binary	Decimal
0 0 0 0	00	1 0 0 0	08
0 0 0 1	01	1 0 0 1	09
0 0 1 0	02	1 0 1 0	10
0 0 1 1	03	1 0 1 1	11
0 1 0 0	04	1 1 0 0	12
0 1 0 1	05	1 1 0 1	13
0 1 1 0	06	1 1 1 0	14
0 1 1 1	07	1 1 1 1	15

Note that the binary number is zero-filled to four bits, just like the decimal number is zero-filled to two digits. A thorough description of the binary number system, including conversion methods, is presented in Chapter 5.

1.2. Modularity

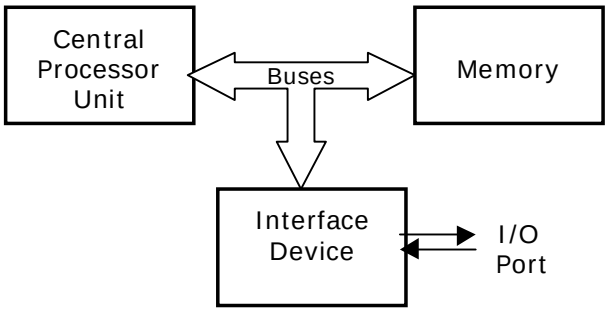
Two major aspects in the design of any computer system are hardware and software. *Hardware* is the physical computer, while *software* consists of the computer programs required to make the computer perform desired functions. Hardware and software must function together in a working computer system.

Modularity is one of the most important concepts in computer engineering and is applied in both hardware and software. Modularity is the process by which once something is designed to perform a desired function; it may be used as a functional block or module to implement the same function in a future design. Modularity avoids the "re-inventing of the wheel" syndrome, and minimizes design time. Modularity can best be utilized by partitioning a large task into simpler functional blocks or modules. This partitions the design into many manageable tasks that can usually be solved separately using previously designed modules. These modules are then interconnected to perform the desired function.

Hardware modularity has led to the rapid advances in computer technology. A computer can be viewed as a group of functional modules, with each module consisting of a hierarchy of smaller sub-modules. All modules are constructed using transistors in the form of integrated circuits. Figure 1.5 illustrates a block diagram of a computer system. All digital computer systems, whether small microprocessor based computers or large super computers, have this same structure. The major components are memory, a CPU, and I/O ports. Memory is used to store binary data that can represent either information or program instructions. The Central Processing Unit (CPU) is used to process data as specified by the instructions of the program. Input/output (I/O) ports are used to transfer information between interface devices and peripherals that may be in the form of a keyboard, display, printer, or disk drives. Figure 1.5 illustrates all functional blocks interconnected using circuit paths called buses.

Software modularity is utilized when a computer program is being written. A program can be divided into many functional modules or *functions*. Each function can be written and debugged separately. Thus, software development employs modularity principles by partitioning a large program into several smaller and more manageable tasks.

Figure 1.5 Computer System Block Diagram



Chapter 2. Combinational Logic

The hardware of a computer system is made of digital logic devices that are interconnected to form a digital circuit. Digital logic is divided into two categories: combinational logic and sequential logic. Combinational logic devices respond with a fixed set of transformation rules, which specify the state of all outputs for every combination of input states. For combinational logic the outputs are a function of only the present state of the inputs. For sequential logic the output is a function of both the present state of inputs and the past state of the output.

2.1. Logic Gates

A *logic gate* is a device that uses a fixed set of rules to transform a set of logical variable inputs into a single logical variable output. Although the inputs and output may vary with time, the transformation rules for a logic gate are time invariant. The output is dependent only on the states of the inputs at that instant.

Logic gates can be connected together to form complex digital circuits, called *combinational networks*. These networks can transform a large set of digital inputs into a large set of outputs. When studying combinational networks, past behavior is not important; what is important are the rules that specify the state of the outputs as a function of all combinations of input states.

All digital circuits, including the largest of computers, are built from three primary logic gates. These primary gates are called *NOT* (or Inverter), *AND*, and *OR*. The inputs and output of a logic gate are logic variables that are in either a one or '0' state. The transformation rules for a gate are specified in a truth table. The *truth table* specifies the state of the output for all possible combinations of input states. Figure 2.1 contains the truth tables for the NOT, AND, and OR gates. Also shown is the digital circuit drawing symbols and Boolean algebra symbols for each of these three gates.

The NOT gate is often called an *inverter* since its output state will be the inverse of the input state. The circuit symbol for the NOT gate is a triangle with a circle on the output, as shown in Figure 2.1a. The Boolean algebra symbol for the NOT operation is an inversion bar placed over the input logic variable. The NOT operation is defined for only one input, while the AND and OR operations require more than one input.

The AND operation for two inputs is depicted by the truth table and logic symbol shown in Figure 2.1b. The AND gate output is '1' only if all inputs are '1'; otherwise, the output is '0'. The Boolean algebra symbol for the AND operation is the same as multiplication in algebra of real numbers.

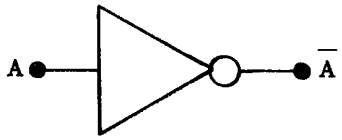
The OR operation for two inputs is depicted by the truth table and logic symbol illustrated in Figure 2.1c. The OR gate output will be '1' if any input is in the '1' state. Therefore, the only case for which the output is '0' occurs when all inputs are '0'. The Boolean algebra symbol for the OR operation is represented by a plus symbol.

Three additional gates, the *NAND*, *NOR*, and *XOR* (Exclusive OR) are simple combinations of the three primary gates NOT, AND, and OR. Figure 2.2 contains the truth tables for the NAND, NOR, and XOR. Also shown, are the digital circuit symbols and Boolean algebra symbols for each of these three gates.

Figure 2.1 Truth Tables for Primary Gates. (a) NOT, (b) AND, and (c) OR

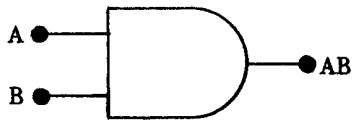
(a) NOT (Inverter)

A	$\overline{A}$ = NOT A
0	1
1	0



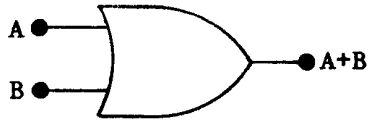
(b) AND

A B	AB = A AND B
0 0	0
0 1	0
1 0	0
1 1	1



(c) OR

A B	A+B = A OR B
0 0	0
0 1	1
1 0	1
1 1	1



The NAND gate can be considered an abbreviation for NOT-AND. The truth table for a NAND gate is the same as an AND gate with its output inverted (NOT). The NAND gate can be drawn as a digital circuit with an AND gate output connected to a NOT gate input. The NAND gate is such a common logic gate that it is usually drawn as an AND gate with a circle on the output of the gate as shown in Figure 2.2a. This circle is called an inversion circle and represents a NOT operation performed on the specified output. The Boolean algebra symbol for a NAND operation is represented with the inversion of the AND operation or $\overline{AB}$.

The NOR gate is similarly a NOT-OR operation which is represented as an OR gate with its output inverted. This can be verified by examining the truth table of Figure 2.2b. The circuit symbol for a NOR gate is an OR gate with an inversion circle on the output. The NOR operation is represented in Boolean algebra as $\overline{A+B}$.

The Exclusive OR (XOR) operation is defined by the following statement: If an odd number of inputs are '1' then the output is a '1'; otherwise, the output is '0'. The circuit symbol for an XOR gate is shown in Figure 2.2c. It is similar to an OR gate with an additional arc drawn across the input side of the gate. The Boolean algebra symbol for the Exclusive OR operation is a plus sign enclosed in a circle denoted by $A \oplus B$.

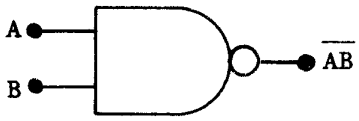
More than two inputs may be used for the AND, NAND, OR, NOR, and XOR gates. A gate with a total of n inputs will have 2^n possible combinations of these n inputs. Therefore, when constructing the truth table for a gate with n inputs, 2^n rows must exist which represent

all possible combinations of input states. This is illustrated by the truth table in Figure 2.3 for a 3-input AND gate. It is best to use the binary counting scheme described in Figure 1.4 to account for all possible combinations of input states.

Figure 2.2 Truth Tables for (a) NAND, (b) NOR, and (c) XOR Gates.

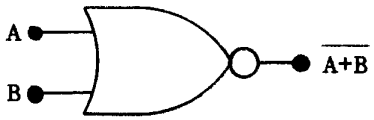
(a) **NAND**

A B	$\overline{AB}$ = A Not AND B
0 0	1
0 1	1
1 0	1
1 1	0



(b) **NOR**

A B	$\overline{A+B}$ = A Not OR B
0 0	1
0 1	0
1 0	0
1 1	0



(c) **XOR (Exclusive OR)**

A B	A⊕B = A XOR B
0 0	0
0 1	1
1 0	1
1 1	0

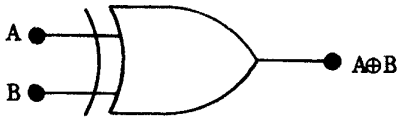
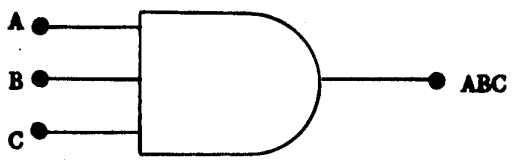


Figure 2.3 Truth Table for a 3-Input AND Gate

AND

A B C	ABC = A AND B AND C
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



2.2. Boolean Algebra

Boolean algebra is a branch of mathematics for which the values of all variables are either ‘0’ or ‘1’. There are three primary Boolean algebra operations which consist of the logic operations NOT, AND, and OR. Since binary (base 2) numbers can represent all logic values in digital circuits, Boolean algebra can be applied for analysis and synthesis of digital circuits. Boolean symbols were used for the output variable in the previously described truth tables. These symbols are summarized in Figure 2.4.

Figure 2.4 Boolean Symbols

$\overline{A}$ = NOT A = Complement A	$A \oplus B$ = A Exclusive-OR B
AB = A AND B	$\overline{A\ B}$ = A NAND B
$A+B$ = A OR B	$\overline{A+B}$ = A NOR B

Just as any other branch of mathematics, Boolean algebra has many identities that have been proven and can be used for simplification or to find equivalent expressions. The most common identities are listed in Figure 2.5. All variables in Boolean algebraic descriptions are logic variables. Therefore, the variables A, B, and C of Figure 2.5 have a value of either ‘1’ or ‘0’. Keeping this in mind, many of the results of these identities are fairly intuitive. The first five identities of Figure 2.5 are the fundamental identities of Boolean algebra. Using these identities, all other identities of Figure 2.5 can be proven using Boolean algebraic manipulation.

Once a Boolean algebra equivalency is proven it can be used as an identity. Equivalence in Boolean algebra is not the same as equality in algebra of real numbers. For example, consider an OR gate with both inputs connected to logic ‘1’. This could be written as 1 OR 1 = 1 or in Boolean algebraic form as 1 + 1 = 1. A common mistake during Boolean algebra manipulation is the improper use of the inversion bar. Note that $\overline{A}\ \overline{B} \neq \overline{A\ B}$.

DeMorgan's law is a very important identity that is used for manipulating inversion bars. DeMorgan's AND Law is written in Boolean algebraic form as $\overline{A\ B} = \overline{A} + \overline{B}$, and can be stated as NOT the quantity A AND B is equivalent to NOT A OR NOT B. DeMorgan's OR Law is an alternate form and is written algebraically as $\overline{A+B} = \overline{A}\ \overline{B}$. Both the AND and OR forms of DeMorgan's Law are equivalent, which is proven in Example 2.2d. DeMorgan's law can also be extended too more than two variables. By applying DeMorgan's law, equivalent gate symbols can be found as shown in

Figure 2.6. Note that inversion circles can be used to show an inversion of the input variables as well as the output.

Duality is a Boolean algebra principle describing once equivalence is proven a dual of this equivalence can be determined that is also a valid equivalence expression. The *dual* of a Boolean expression can be determined as follows: Replace all AND operations with OR operations and all OR operations with AND operations on both sides of the equivalency; then replace all 1's with 0's and 0's with 1's on both sides of the equivalency. The duality principle was applied in constructing the identity table of Figure 2.5. Note that the dual of the AND form of an identity is the OR form of the same identity.

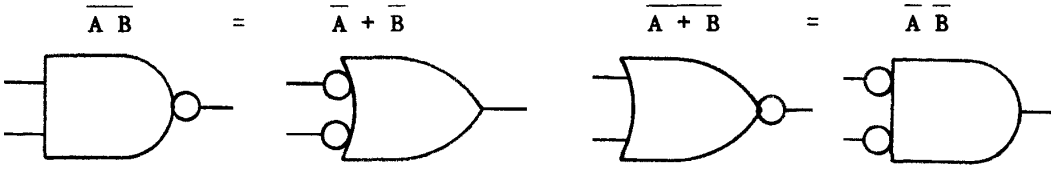
Substitution is the process by which a logic variable may be substituted for a Boolean expression or vise-versa. Substitution is frequently used when simplifying a Boolean expression. It can also be used to extend the identities of Figure 2.5 to a greater number of variables than what is specified.

Double Inversion is another identity. If a Boolean expression has two inversion bars over it, they cancel each other and both inversion bars can be removed. This relationship can be expressed in algebraic form as $\overline{\overline{A}} = A$ or $\overline{\overline{A\ B}} = A\ B$.
Note that $\overline{\overline{A}\ \overline{B}} \neq A\ B$.

Figure 2.5 Boolean Algebra Identities

Name	AND Form	OR Form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\ \overline{B}$
Absorption law	$A(A + B) = A$	$A + AB = A$
Inclusion law	$A(\overline{A} + B) = AB$	$\overline{A}B + B = A + B$

Figure 2.6 Gate Description for DeMorgan's Laws



2.3. Boolean Equivalence Verification

For any given combinational network, a Boolean expression can be written for each of the outputs in terms of the inputs. Using Boolean algebra, equivalent circuit designs can be found for optimizing a design or predicting the results for various situations.

Two methods may be employed to verify Boolean identities or find an alternate equivalent solution. One method uses a truth table and the other uses Boolean algebra.

2.3.1. Truth Table Verification

When constructing a truth table the output values must be found for all possible combinations of input states. For n input variables, 2^n rows will be required for the truth table. All logic variables are in either the '1' or '0' state. Example 2.1a through Example 2.1c illustrates the use of truth tables to prove equivalency of two Boolean expressions.

Example 2.1 Truth Table Verification

a) Verify: $1A = A$ (Identity AND Law)

A	1	1A
0	1	0
1	1	1

Verifies

b) Verify: $A + B = \overline{A}\,\overline{B}$ (De Morgan's OR Law)

A B	A + B	$\overline{A + B}$	$\overline{A}$	$\overline{B}$	$\overline{A}\,\overline{B}$
0 0	0	1	1	1	1
0 1	1	0	1	0	0
1 0	1	0	0	1	0
1 1	1	0	0	0	0

Verifies

c) Verify: $(A + B)(A + \overline{B}) = A$

A B	A + B	$A + \overline{B}$	$(A + B)(A + \overline{B})$
0 0	0	1	0
0 1	1	0	0
1 0	1	1	1
1 1	1	1	1

Verifies

Construction of the truth table begins by writing all possible combinations of input logic states in the left-most columns of the truth table. This can be done best by using the binary counting scheme of Figure 1.4 to account for all possible combinations of input variables. Then perform one primary logic operation (AND, OR, or NOT) by determining the output value of the logic operation for all combinations of input logic variables. Any column can be used as an input to perform additional logic operations, whether it is an output from a previous logic operation or an input logic variable. For clarity, vertical lines should separate all columns that represent results of logic operations. Continue performing logic operations in the truth table until two columns exist which represent the two Boolean

expressions on either side of the equality. If the columns match for all possible combinations of the inputs, the two Boolean expressions are said to be equivalent.

Example 2.1 demonstrates truth table verification to prove the Identity AND Law and DeMorgan's OR Law. Any of the identities of Figure 2.5 may be proven using truth tables. Example 2.1c describes a Boolean relationship, which is verified using a truth table. Once this relationship is proven to be equivalent, it may be used as an identity.

2.3.2. Boolean Algebra Verification

In mathematics, algebraic manipulation using proven identities can create equivalent expressions. The same procedure can be applied to Boolean algebra. Using the Boolean identities of Figure 2.5, substitution, and double inversion, a Boolean expression can be manipulated to find an alternate equivalent expression. Two expressions are equivalent when they generate the same results for all possible combinations of logic variable inputs. Equivalence can be proven using truth tables. However, for expressions with more than four logic variables, truth tables become tedious and Boolean algebra verification is often easier.

Example 2.2 Algebraic Verification

a) Verify: $A + A\,B = A$ (Absorption OR Law)

$= A\,1 + AB$ (Identity AND Law)

$= A\,(1 + B)$ (Distributive OR Law)

$= A\,1$ (Null OR Law)

$= A$ (Identity AND Law)

b) Verify: $A\bar{B}+B = A+B$ (Inclusion OR Law)

$= B+A\bar{B}$ (Commutative OR Law)

$= (B + A)(B + \bar{B})$ (Distributive AND Law)

$= (B + A)\,1$ (Inverse OR Law)

$= B + A$ (Identity AND Law)

$= A + B$ (Commutative OR Law)

c) Simplify: $X = \overline{A}\ \overline{B}\ C + A\ C + B\ C$

$= C\ (\overline{A}\ \overline{B} + B + A)$ (Distributive OR Law)

$= C\ (\overline{A} + B + A)$ (*Inclusion OR Law*)

$= C\ (1 + B)$ (Inverse OR Law)

$= C\ (1)$ (Null OR Law)

$X = C$ (Identity AND Law)

d) Show the two forms of De Morgan's Law are equivalent.

$\overline{A\ B} = (\overline{A} + \overline{B})$ (De Morgan's AND Law)

$\overline{\overline{A\ B}} = \overline{(\overline{A} + \overline{B})}$ (Invert Both Sides)

$A\ B = \overline{(\overline{A} + \overline{B})}$ (Double Inversion)

Set $C = \overline{A}$ $D = \overline{B}$ (Substitution)

$\overline{C}\ \overline{D} = \overline{C + D}$ (De Morgan's OR Law)

Example 2.2a and Example 2.2b demonstrate the use of Boolean algebra to verify the Absorption OR Law and Inclusion OR Law using other identities from Figure 2.5. Note that the identities of Figure 2.5 specify the form of equivalent expressions and not the actual input variables themselves. Example 2.2c illustrates the use of Boolean algebra to simplify an expression. After simplification is performed, one can see that output X is dependent only on the value of input C and is in fact equivalent to C. Example 2.2d demonstrates that both the AND and OR forms of DeMorgan's law are equivalent using Boolean algebraic manipulation.

2.4. Combinational Network Design

Design of combinational networks requires a working knowledge of Boolean algebra, truth table construction, and digital logic gate representations. It is essential to know how to go from one representation to another.

Several configurations of logic gates may have the same input/output characteristics; that is, each combination of input states produces the same output states. Two combinational networks with the same input/output characteristics are said to be equivalent. Equivalence is verified through truth table construction or Boolean algebra. Figure 2.7 illustrates two equivalent combinational networks. The networks are the digital circuits representing the Distributive AND Law of Figure 2.5.

2.4.1. Description to Digital Circuit Design

Once a description of the desired logic function is defined, a digital circuit can be designed to implement the function. The description may be a truth table or a verbal description. To understand the function of a digital circuit, the state of the outputs must be known for all possible combinations of inputs. Constructing the truth table, as discussed in Section 2.3.1, is one way to account for all possible combinations of inputs.

After the description is defined, the next step towards designing the digital circuit is to determine a Boolean expression that describes the desired function. The Sum of Products and Product of Sums methods are two procedures, which can be utilized to determine a valid Boolean expression directly from a truth table description. The induction method is used to determine a valid Boolean expression directly from a verbal description.

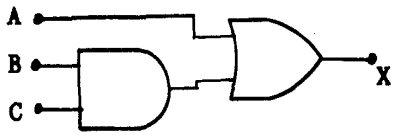
The truth table of Figure 2.7 is used as an example to demonstrate the sum of products method for the second output column (X=A+BC).

After a Boolean expression has been found to describe the logic function, the digital circuit can be designed directly. This is accomplished by using the gates discussed in Section 2.1 to implement each of the logic functions of the Boolean expression. Just as with algebra of real numbers, product operations (AND functions) are performed first and then sum operations (OR functions). Parentheses are used to specify a different order of operation and to group terms.

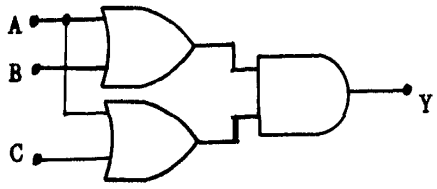
Two simple digital circuits are illustrated in Figure 2.7 for outputs X and Y. Examining the output X and Y columns of the truth table of Figure 2.7, one can verify that these are equivalent circuits.

Figure 2.7 Equivalent Combinational Networks

a) X=A+BC



b) Y=(A+B)(A+C)



A B C	B C	X = A + BC	A + B	A + C	Y = (A + B)(A + C)
0 0 0	0	0	0	0	0
0 0 1	0	0	0	1	0
0 1 0	0	0	1	0	0
0 1 1	1	1	1	1	1
1 0 0	0	1	1	1	1
1 0 1	0	1	1	1	1
1 1 0	0	1	1	1	1
1 1 1	1	1	1	1	1

2.4.1.1. Sum of Products (SOP) Method

The Sum of Products (SOP) method is a procedure, for determining a valid Boolean expression from a truth table description. This method considers only those rows of the truth table with logic '1' in the output column.

Examining the truth table of Figure 2.7, Output X equals '1' in the fourth row. This occurs when the inputs conditions are A=0, B=1, and C=1. Similarly, for the fifth row X equals '1' when A=1, B=0, and C=0. Boolean expressions can be written to express these relationships for the fourth and fifth rows as $\overline{A}BC=1$ and $A\overline{B}\overline{C}=1$. Likewise, Boolean expressions can be written for the sixth row as $A\overline{B}C=1$, the seventh row as $AB\overline{C}=1$, and eighth row as ABC=1. These Boolean expressions are called *minterms*. Minterms only exist for rows that have an output of logic 1. A Boolean expression can be written describing the entire truth table by OR'ing each of the five minterms together. This will result in the following Boolean expression:

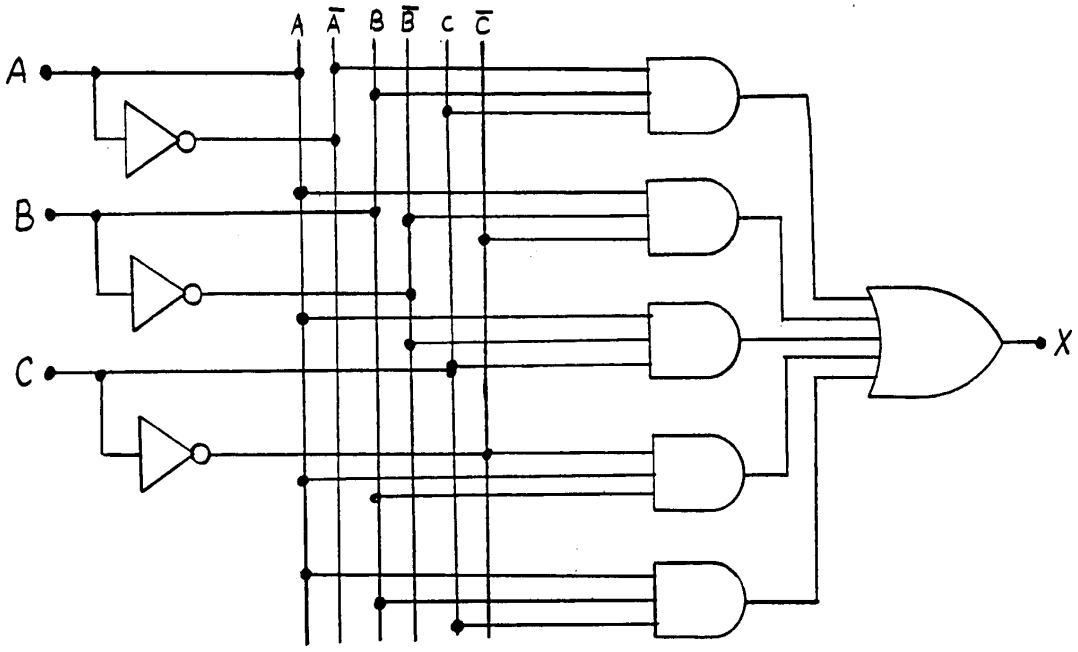
$$X = \overline{A}\,B\,C + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C + A\,B\,\overline{C} + A\,B\,C$$

Notice the form of the expression is a sum of products, hence the name of the method describes the resultant form.

The digital circuit can then be created from this sum of products expression as illustrated in Figure 2.8.

Figure 2.8 SOP Digital Circuit Drawing

$$X = \overline{A}\,B\,C + A\,\overline{B}\,\overline{C} + A\,\overline{B}\,C + A\,B\,\overline{C} + A\,B\,C$$



2.4.1.2. Product Of Sums (POS) Method

The Product Of Sums (POS) method is another procedure, for determining a valid Boolean expression from a truth table description. This method considers only those rows of the truth table with logic '0' in the output column.

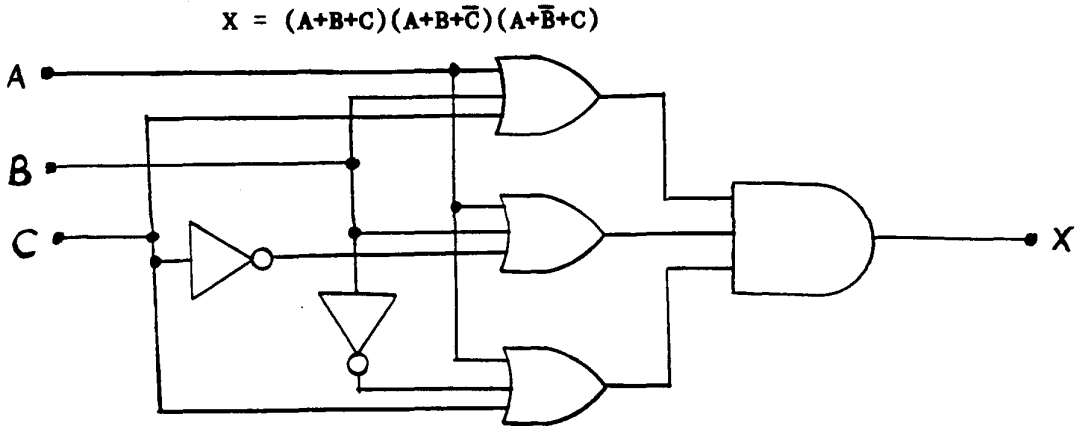
The first three rows of the truth table of Figure 2.7 have '0's in output column X. A *maxterm* can be written for each of these rows using the following procedure:
If the input is 1 for the specified row, the inverse of the input variable is used;
otherwise, if the input is 0, the input variable is used directly.
The input logic variables are then OR'ed together to form the maxterm. Only rows with an output of 0 will have a maxterm.

The maxterms for the truth table of Figure 2.7 are written as follows: The first row is (A+B+C), the second row is (A+B+$\overline{C}$), and the third row is (A+$\overline{B}$+C). The resulting maxterms can then be AND'ed together to form a valid Boolean expression for the truth table. This will result in the following expression, which is in product of sum form.

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

The digital circuit for this product of sums expression is shown in Figure 2.9.

Figure 2.9 POS Digital Circuit Drawing



2.4.1.3. Induction Method

The induction method describes the process of determining a Boolean expression directly from the verbal description. This is particularly useful for systems with more than four inputs because truth tables become cumbersome and the resulting sum of products and product of sums expressions can become lengthy. Output X in Figure 2.7 can be described verbally as:

X is 1 if A is 1, or if B and C are 1. Otherwise, X is 0.

From this verbal description the Boolean expression can be written directly:
X = A + B C

2.4.2. Digital Circuit Minimization

The digital circuits of Figures 2.7, 2.8, and 2.9, are all equivalent combinational networks since they all generate the same input/output relationships and are derived from the same truth table. The digital circuit of Figure 2.7a is the preferred choice since it requires the least number of logic gates for the implementation. When designing from the truth table it is best to use the sum of products method when fewer '1' s than '0's exist in the output column, and use the product of sums method when there are fewer '0's than '1' s.

Once a Boolean expression is found using either sum of products, product of sums, or induction, algebraic simplification of the expression is often possible using Boolean identities. This may greatly reduce the number of gates required to construct the circuit.

As an example, suppose a digital circuit must be designed which has two outputs X and Y; and must accomplish the following functions.

Output X is 1 if either E or F are 1 and D is 0. Otherwise X is 0.
Output Y is 1 if A and B and C are 1, or D is 0 and either B or C are 0, or if D is 1. Otherwise Y is 0.

This circuit has 6 inputs (A, B, C, D, E, and F) and two outputs (X and Y). Constructing the truth table would be tedious with 64 rows, and the resulting SOP or POS Boolean expressions would require much effort to reduce. For these reasons, the Boolean expression is found using induction from the verbal description and then simplifying the result.

Both the unsimplified and simplified digital circuits for this example are illustrated in Figure 2.10 and Figure 2.11. Note that inversion circles are shown on the inputs of some of the gates. These symbolize inverters.

$$X = (E + F)\,\overline{D}$$

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

$$= A\,B\,C + (\overline{B} + \overline{C}) + D \qquad \text{(Inclusion OR Law)}$$

$$= A\,B\,C + \overline{B\,C} + D \qquad \text{(De Morgan's AND Law)}$$

$$= A + \overline{B\,C} + D \qquad \text{(Substitution and Inclusion OR Law)}$$

Figure 2.10 Unsimplified Digital Circuit From Induction

$X = (E + F)\,\overline{D} \qquad Y = A\,B\,C + \overline{D}\,(\overline{B} + \overline{C}) + D$

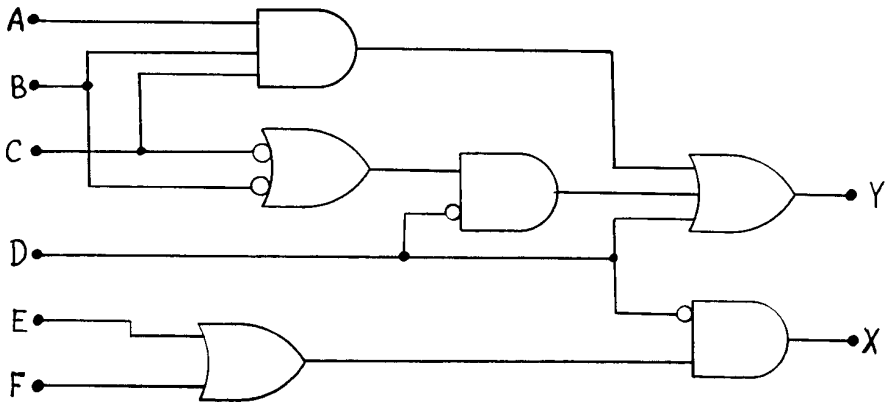
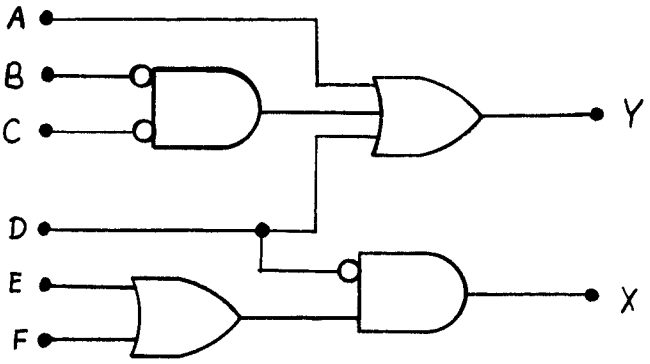


Figure 2.11 Simplified Digital Circuit from Induction

$X = (E + F)\,\overline{D} \qquad Y = A + \overline{B\,C} + D$



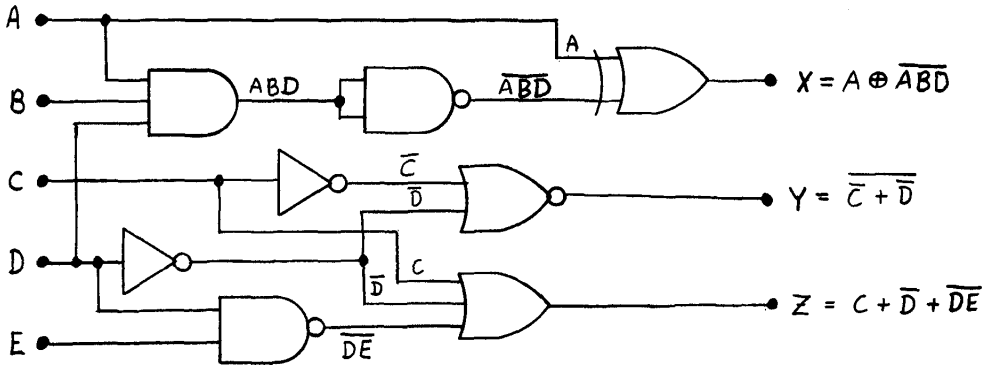
2.4.3. Boolean Expressions from Digital Circuit

Modification of existing digital circuits requires the ability to go from a digital circuit drawing to a Boolean expression. Two procedures for accomplishing this conversion are discussed in this section. The first procedure is used when Boolean expressions for all outputs of a digital circuit are required. The second procedure is for the case when the Boolean expression for only one output is needed.

The procedure for determining the Boolean expression for all outputs in a digital circuit is quite straightforward. Beginning from the input side of the digital circuit drawing, write the output Boolean expression as you progress through each gate. Then use the output equation from the preceding gate as the input to the next gate in the circuit. Continue writing Boolean expressions at the output of each gate until the Boolean expressions for all outputs of the digital circuit are determined. An example of this procedure is shown in Figure 2.12. Notice the 2-input NAND gate with both inputs tied together. A NAND gate in this

configuration will function as an inverter. Similarly, a NOR gate with all inputs tied together would also function as an inverter. This can be verified by examining the truth tables of Figure 2.2.

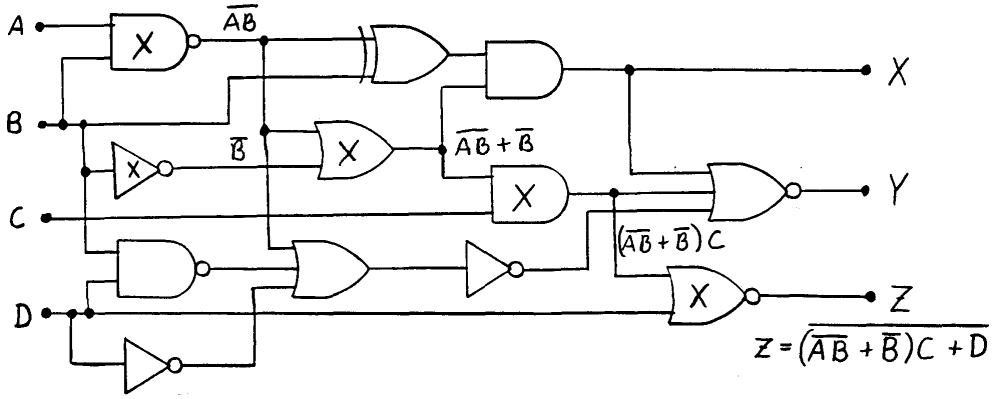
Figure 2.12 Boolean Expressions for All Outputs



Consider the case of a complex digital circuit with many outputs, and suppose the Boolean expression for only one output is needed. For this case, the first step is to trace through the circuit from the output of interest to the inputs, marking each gate that will affect this output. This is illustrated in Figure 2.13. After the gates are marked, proceed from the input side to the output of interest by writing the Boolean expression for the output of each affecting gate. This procedure will save considerable time when determining the Boolean expression of a single output in complex combinational networks.

Once the Boolean expression is determined for an output, Boolean algebra can then be used to reduce the expression to a simplified equivalent form.

Figure 2.13 Boolean Expression For One Output



2.5. Common Combinational Circuits

Once a digital circuit has been designed using individual gates to perform a specific function, it is often desirable to use the newly created circuit as a module for future designs. Discussed in this section are several commonly used combinational circuits that include

decoders, multiplexers, adders, and arithmetic logic units. These combinational circuits are used as building blocks to construct all computers.

2.5.1. Decoders

A *decoder* is a digital circuit with n inputs and 2^n outputs. Figure 2.14a illustrates a logic gate diagram that can be used to construct the 3 to 8 decoder. Figure 2.14b is a block diagram of the 3 to 8 decoder. Note that the block diagram contains all inputs and outputs illustrated in the logic gate diagram. After a digital circuit has been designed to perform a specific function, it can be considered as a functional module and is usually represented by the block diagram.

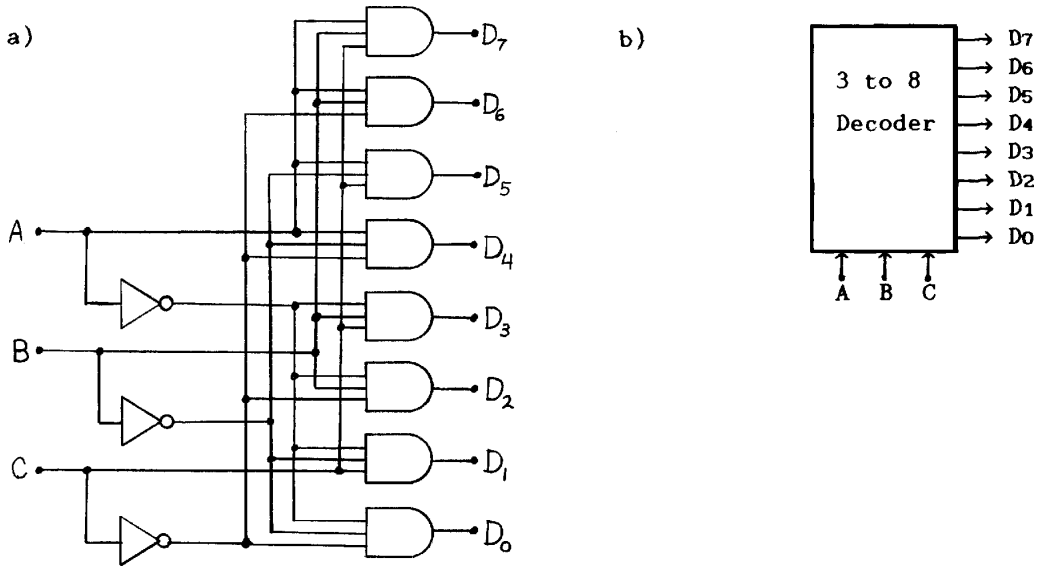
The decoder of Figure 2.14 functions such that one and only one output is in the 1 state, as selected by a binary code placed on select inputs A, B, and C. Since only one output can be in the 1 state, all other outputs will be in the 0 state.

Decoders are used to select one of 2^n devices with an n-bit code. The n-bit code is often called the address of the selected device. Note that only one device can be addressed at a time since only one decoder output will be in the 1 state. Decoders are used in computers to select one of several functions in the CPU and to select one of many memory locations.

Decoders can be constructed to generate either positive logic outputs or negative logic outputs. The decoder of Figure 2.14 is an example of a positive logic output decoder. Positive logic output decoders function such that the selected (or enabled) output is in the 1 state and all other outputs are in the 0 state (or disabled). Negative logic output decoders function such that the selected output is in the 0 state while all other outputs are in the 1 state. A negative logic output 3 to 8 decoder can be constructed by modifying the logic diagram of Figure 2.14. If the eight three-input AND gates are replaced with eight three-input NAND gates, then the decoder will have negative logic outputs.

Figure 2.14 3 to 8 Decoder

a) Digital Circuit b) Block Diagram



2.5.2. Multiplexers

A *multiplexer* is a combinational circuit in which one of several data inputs is selected and routed to a single output. Figure 2.15 is an example of a four data input multiplexer. The block diagram of Figure 2.15a illustrates the four data inputs (D_0 through D_3) of which one is selected and routed to the single output F. The data input is selected by applying a binary code on the select inputs A and B. Two select inputs will generate four unique binary codes. Therefore, any one of four data inputs may be selected when using two select inputs. For a general case multiplexer with n select inputs a maximum of 2^n data inputs could be available. A multiplexer with 4 select inputs may have a maximum of 16 data inputs.

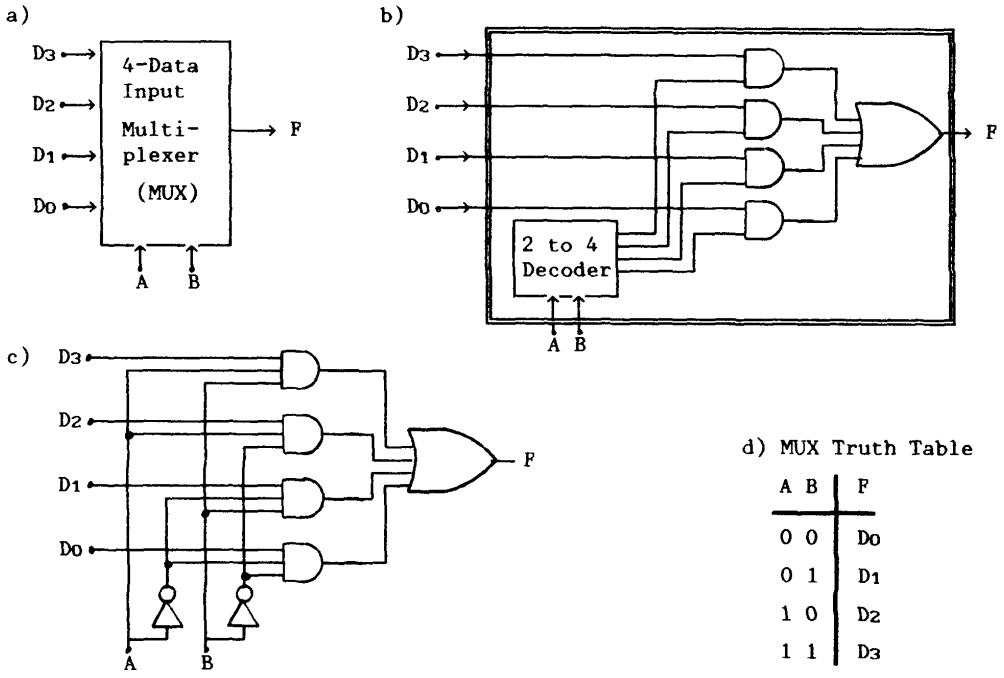
The 4-data input multiplexer can be constructed using a 2 to 4 decoder and logic gates as shown in Figure 2.15b. The decoder can be considered as a sub-module within the multiplexer module. A logic gate implementation for the 4-data input multiplexer is illustrated in Figure 2.15c. The reader is urged to verify the multiplexer operation. A functional truth table for the 4-data input multiplexer is shown in Figure 2.15d. Note that the output state is the state of the selected data input.

Multiplexers are generally used for data routing applications and can be considered as a data switch. As an example, consider the case of four computers that are connected to one printer. Only one computer can send data to the printer at a time; therefore, a 4-data input multiplexer is chosen as a data switch. A two bit binary code will be used to address each of the four computers.

A demultiplexer performs the opposite function. One data input is routed to several possible data outputs. Select inputs determine which data output transmits the data. A demultiplexer will have one data input, n select inputs, and 2^n data outputs.

Figure 2.15 4-Data Input Multiplexer

a) Block Diagram b) Digital Circuit c) Logic Gate Diagram d) Truth Table



d) MUX Truth Table

A	B	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

2.5.3. Binary Adders

Addition of binary numbers is accomplished using a digital circuit called an adder. Figure 2.16 illustrates a half adder which is used to add two single-bit binary numbers represented by logic variables A and B. The half adder circuit has two inputs A and B, and two outputs Sum and Carry.

The truth table for the half adder, shown in Figure 2.16a, describes binary addition of two single-bit binary numbers. When both inputs are 0, the sum is 0. If either input A or B is 1, but not both, the sum is 1. When both A and B are 1, the sum exceeds what can be shown with a single bit; therefore, the sum is 0 and the carry is set to 1. The carry output is 1 only when both inputs A and B are logical 1. Based on this description, the sum operation can be accomplished by using an Exclusive-OR gate and the carry operation can be performed using an AND gate.

Figure 2.16 Half Adder

a) Truth Table b) Logic Diagram

a)

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

b)

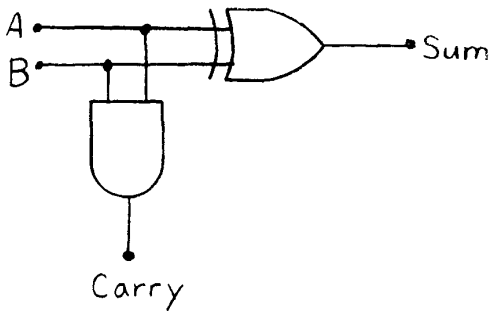


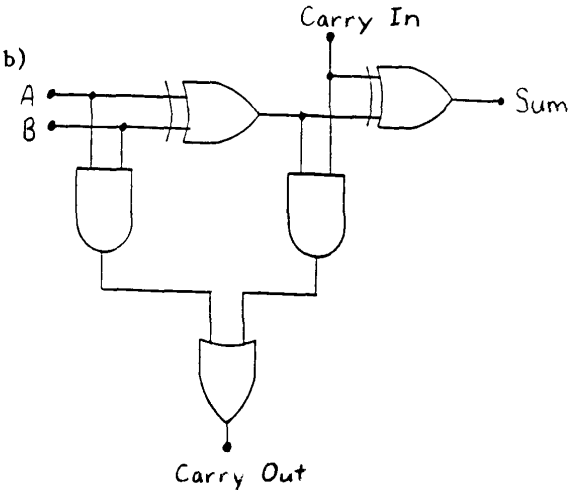
Figure 2.17 Full Adder

a) Truth Table b) Logic Diagram

a)

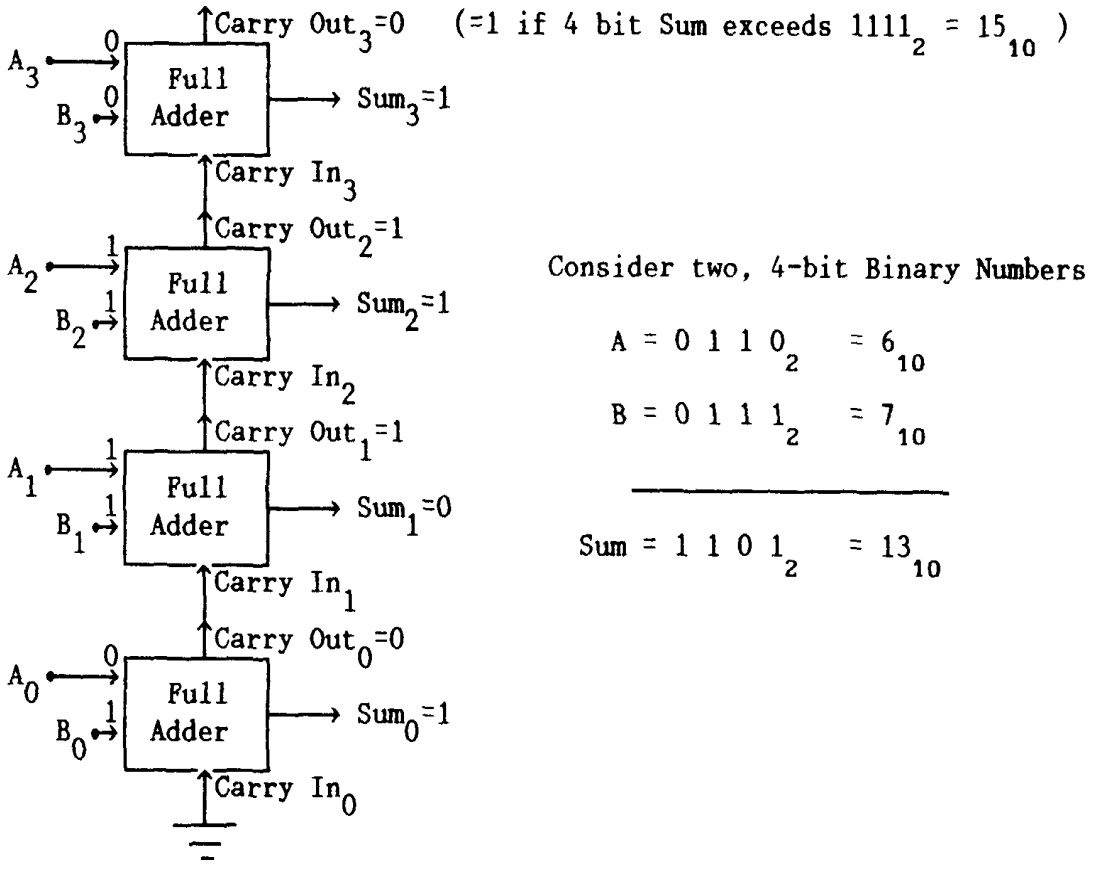
A	B	Carry In	Sum	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

b)



To utilize the carry for the next significant bit, a full adder is used for multi-bit addition. The truth table and logic diagram of the full adder is shown in Figure 2.17. The 3 inputs A, B, and Carry In are added together to generate the two outputs Sum and Carry Out. Figure 2.18 illustrates a digital circuit, which will perform binary addition on two four-bit binary numbers, which are represented by A_3 to A_0 and B_3 to B_0. A_3 is the most significant bit and A_0 is the least significant bit of the 4-bit binary number A. Four full adder circuits (shown as block diagrams) are utilized to construct this digital circuit. The Carry In of the least significant bit is connected to ground because a carry will not occur into the least significant bit. The Carry Out of the least significant bit is connected to the Carry In of the next significant bit as illustrated in Figure 2.18. The connections of Carry Out to Carry In of the next significant bit continue throughout the circuit. When the Carry Out of the most significant bit is 1, the sum exceeds what can be shown with the number of bits allotted for the sum. The Carry Out of the most significant bit is often called the Carry Flag.

Figure 2.18 Four Bit Binary Adder



2.5.4. Arithmetic Logic Units

The Arithmetic Logic Unit (ALU) is a device, which can perform several operations on two binary numbers. All computers contain an ALU, as a module within the Central Processing Unit. The ALU performs the arithmetic and logic operations specified by the instructions of a computer program.

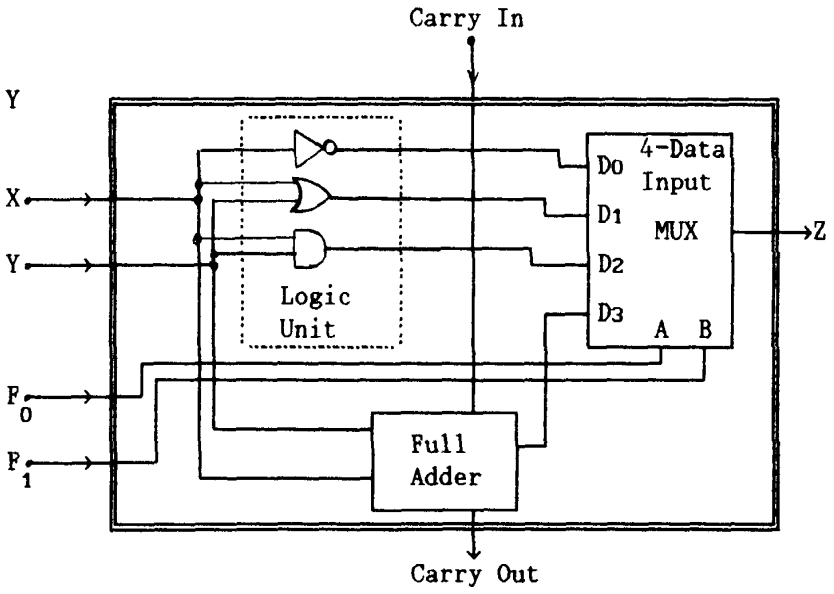
Figure 2.19 is an example of a single-bit 4-function ALU. Inputs X and Y are the two Boolean variables on which a particular operation is performed. Output Z is the result of the operation. The ALU shown will perform one of four functions (NOT, OR, AND, or SUM) on the inputs X and Y to generate the result Z. The function performed is determined by the values of functional inputs F_0 and F_1. For example, when the F_0 and F_1 inputs are 0 and 1 respectively, an OR operation will be performed on inputs X and Y. When both F_0 and F_1 are 1, a sum operation is performed and the Carry In input and Carry Out output will be utilized. The functional truth table of Figure 2.19 summarizes these operations.

The ALU of Figure 2.19 can be thought of as a module, which contains three sub-modules, namely a full adder, a logic unit, and 4- data input multiplexer. The full adder and logic units are used to perform sum and logic operations on the input variables. The multiplexer is used to route the output of the selected function to output Z.

A four-bit ALU could be constructed by cascading four single-bit ALU modules together much like the full adder circuit of Figure 2.18. All four functional inputs F_1 would be connected together and all F_0 inputs would be interconnected so that each ALU would perform the same function on each bit.

Figure 2.19 Single-Bit 4-Function ALU

F_0	F_1	Z
0	0	$\overline{X}$
0	1	X+Y
1	0	X Y
1	1	$\overline{X}$ plus Y



Problem Set

1. Make a conversion table from Decimal (Base 10) to Binary (Base 2) from zero to thirty-five. Zero fill the binary numbers to generate six bits.

2. Draw the following gates and construct the truth tables for these gates.
   a) 2-input AND Gate
   b) 2-input XOR Gate
   c) 2-input NOR Gate
   d) 3-input NAND Gate
   e) 3-input XOR Gate
   f) 4-input OR Gate
3. Draw the gates which represent the following Boolean symbols.
   a) $A+B$
   b) ABC
   c) $\overline{A+B+C}$
   d) $A \oplus B$
   e) $\overline{ABCD}$
4. Prove the following Boolean Identities using Truth Tables.
   a) $0A = 0$
   b) $1 + A = 1$
   c) $AA = A$
   d) $A(A + B) = A$
   e) $(A + B) + C = A + (B + C)$
   f) $A(\overline{A} + B) = AB$
   g) $A + BC = (A + B)(A + C)$
   h) $\overline{AB} = \overline{A} + \overline{B}$

5. Prove the following equivalencies using Boolean algebra.
   a) $(A + B)(A + C) = A + BC$
   b) $A(\overline{A} + B) = AB$
   c) $(A + \overline{B + C})\,\overline{A} = \overline{A+B+C}$
   d) $A\,B + C + \overline{A\,B}\,C\,D + C\,C = A\,B + C$
   e) $\overline{A\,\overline{B\,(B+C)}} = \overline{A} + B$
   f) $A\,\overline{B} + A\,\overline{C} + B\,C = A + B\,C$
   g) $\overline{\overline{A}\,\overline{B}}\,\overline{\overline{A}\,\overline{C}} = A + B + C$
   h) $\overline{(\overline{A} + \overline{B} + \overline{C})\,C} = A\,B + \overline{C}$

6. Simplify the following Boolean expression. Verify your answer using a truth table.

   $F = (A + B)\,\overline{(\overline{A}\,(\overline{B} + \overline{C}))} + \overline{A}\,\overline{B} + \overline{A}\,\overline{C}$

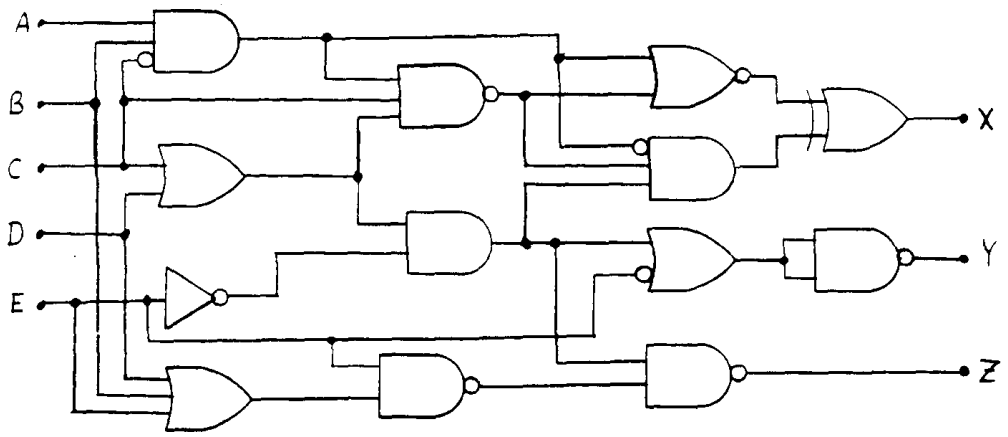
7. Write both sum of products and product of sums Boolean expressions for output Z using the truth table shown. Draw a logic diagram for both circuits.

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

8. Draw a logic diagram for the following Boolean Expression.

   $Z = \overline{(A \oplus B) + \overline{B\,C} + \overline{B\,C}\,\overline{C}}$

9. Write a Boolean Expression for Output Z.



10. Construct the truth table for the 3 to 8 decoder of Figure 2.13a.

11. Construct the logic diagram and truth table for each of the following combinational circuits.
    a) 2 to 4 Decoder with positive logic outputs.
    b) 2 to 4 Decoder with negative logic outputs.
    c) 8-Data Input Multiplexer.
    d) 4-Data Output De-Multiplexer (Unselected outputs will be in 0 state)

12. Construct a 4-bit adder using logic gates.

Chapter 3. Integrated Circuits

Digital circuit design is considered a high-level design method, because only the states of the inputs and outputs are important. Gates will function as expected as long as nominal analog circuit parameters are not exceeded. There are many advantages to using digital circuits over conventional analog circuits. These advantages include modularity, reliability, and noise immunity.

Several procedures have been discussed for designing a digital circuit from either verbal or truth table descriptions in Section 2.4.1. The next step is to use actual digital circuit components to construct the circuit.

Gates are not manufactured individually but are sold in packages containing several gates. These packages are called integrated circuits. Often the term integrated circuit is abbreviated IC or called by its nicknamed “chip”. Integrated circuits are available in several package styles. The two most common are the Dual In-line Package (DIP) and Surface Mount Technology (SMT) packages. Inside the integrated circuit package is a small silicon chip measuring less than 1/4-inch square. This chip contains the transistors and connecting circuits required for implementing the logic components. Several transistor technologies exist for fabrication of silicon chips as described in Section 3.3. .

Integrated circuits are classified by the number of gates fabricated on the integrated circuit. The classification is approximated by:

SSI (Small Scale Integrated) circuit:	1 to 10 gates
MSI (Medium Scale Integrated) circuit:	10 to 100 gates
LSI (Large Scale Integrated) circuit:	100 to 100,000 gates
VLSI (Very Large Scale Integrated) circuit:	> 100,000 gates

3.1. Dual In-Line Packages

A common integrated circuit package is the Dual In-line Package or DIP. Figure 3.1 illustrates mechanical views of a dual in-line package for a 14 pin DIP. DIPs are commonly available in 14, 16, 20, 22, 24, 28, 40, 64, and 68 pin arrangements with the pins always positioned in two parallel rows as shown in Figure 3.1. Package materials are usually plastic or ceramic, with the pins made of gold or tin plated metal. Electrical contact between the pins and the silicon chip is usually made using gold filament wires which are ultrasonically welded to conductive pads on the silicon chip. Proper orientation of the IC is determined by using either the notch shown in or by locating a small depression on the top of the IC, which specifies pin 1.

The standard dimensional units used for IC packages are inches (English System of Units), as shown in Figure 3.1, Figure 3.5, and Figure 3.6. Therefore, the physical layout of a digital circuit on a circuit board is also usually done using inches. Often, measurements are said to be in mils, which is an abbreviation for mil-inches or thousandths of an inch. It has become an industry standard to make the pin separation for DIP’s 100 mils or 0.100 inches.

One very common SSI circuit family is the 7400 Series integrated circuits, which are produced by a variety of semiconductor manufacturers. Functional views of several of these integrated circuits are illustrated in Figure 3.2. These ICs contain the basic logic gates described in Section 2.1. Specific pins on the integrated circuit package are connected

internally to the inputs and output of a gate, which has been fabricated on the chip. Power and ground are symbolized by Vcc and GND. Power usually comes from a 5 Volt source for most digital circuits. The following paragraphs describe four integrated circuit classifications and examples for each.

Small Scale Integrated Circuits (SSI) usually contain several gates inside one integrated circuit package. When describing the number of functional units in the integrated circuit, one usually uses the prefixes dual for two, triple for three, quad for four, and hex for six units in the package. For example the integrated circuits of Figure 3.2 would be described as follows:

7400 = Quad 2-Input NAND Gates,	7404 = Hex Inverters,
7411 = Triple 3-Input AND Gates,	7421 = Dual 4- Input AND Gates

To construct a digital circuit, integrated circuits are usually mounted on a printed circuit board. The ICs are interconnected using conductive wire-like circuit paths, which are etched on the circuit board when it is manufactured. The pins of the ICs and other electrical components are soldered to the circuit board to ensure good electrical contact and mechanical bonding. Standard DIP integrated circuits are usually classified as *through-hole component technology* as their pins must pass through holes in the circuit board before they are soldered.

Figure 3.1 14-Pin DIP Package

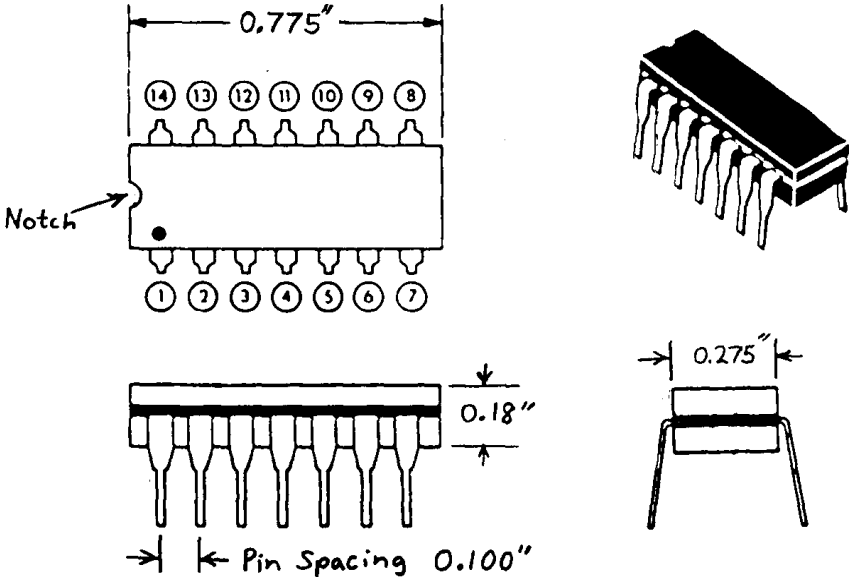


Figure 3.2 Functional Views of Several 7400 Series Integrated Circuits

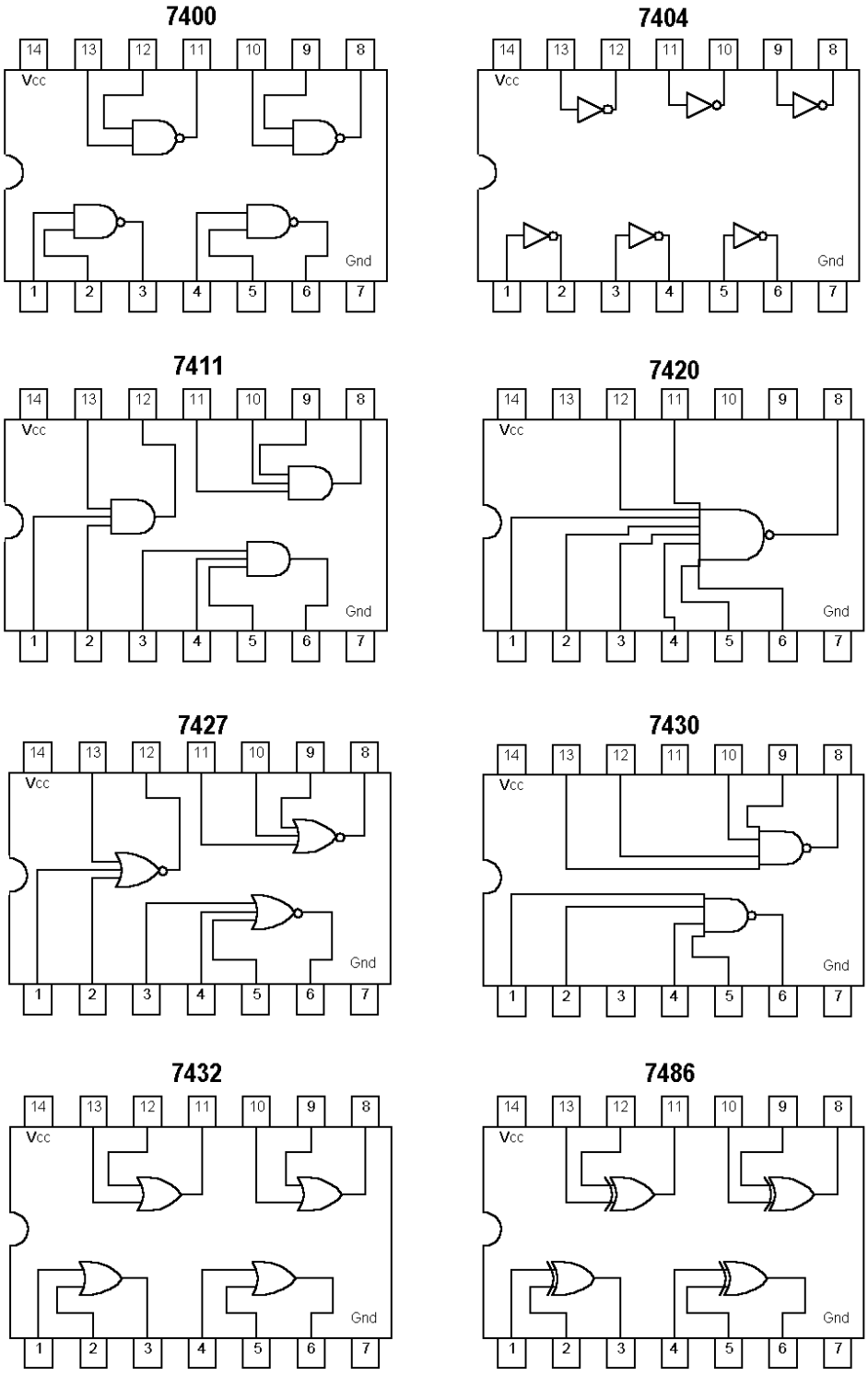
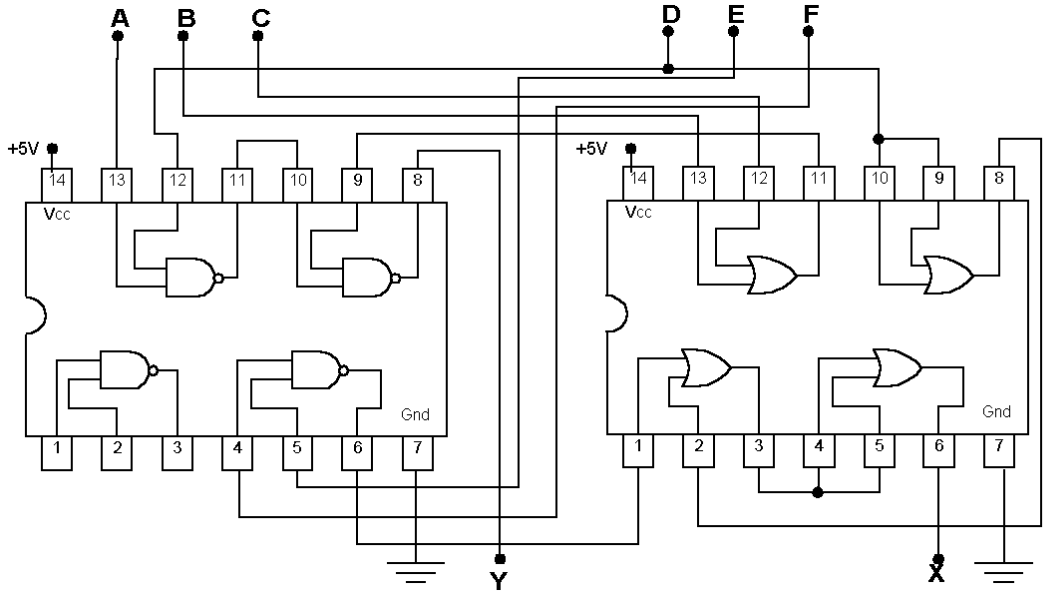


Figure 3.3 IC Digital Circuit Drawing

$X = (E + F)\,\overline{D}, \quad Y = A + \overline{B\,C} + D$



Consider the digital circuit of Figure 2.11. To construct this circuit would require one 7400 Quad 2-Input NAND Gate IC, and one 7432 Quad 2-Input OR Gate IC. The IC digital circuit is designed by drawing lines, which represent circuit paths, between the pins of IC's. This is demonstrated in Figure 3.3 for the digital circuit diagram of Figure 2.11, which represents the Boolean expressions $X = (E + F)\,\overline{D}$ and $Y = A + \overline{B\,C} + D$. Note that the 3-input OR gate is implemented by using two 2-input OR gates. Inverters are implemented by connecting all inputs of a NAND gate together. An extra gate, contained in the 7432 IC, will not be used. Any extra gates or spares can be used for a future modification of the digital circuit. Power and ground connections are represented by connections to +5V and ⏚ Gnd symbols.

Medium Scale Integrated Circuits (MSI) are collections of interconnecting gates that are used as a module to perform a specific function. Generally, MSI chips are classified as those containing between 10 and 100 gates.

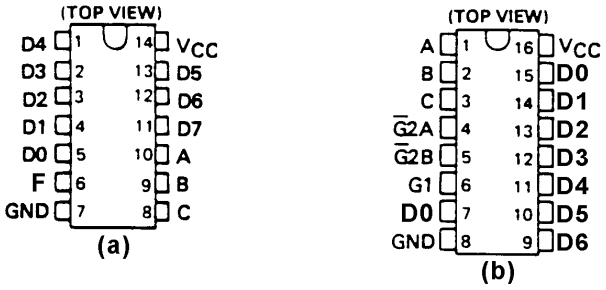
The MSI circuit for an 8-data input multiplexer is the 74152, which is shown in Figure 3.4a. Contained in this integrated circuit are the transistors required to implement the multiplexer function. Standard MSI integrated circuits are available for 4, 8, and 16 data input multiplexers.

Figure 3.4b illustrates an integrated circuit DIP for a 3 to 8 decoder with negative logic outputs (74138). The decoder integrated circuit (74138) has three additional inputs, G1, $\overline{G2A}$, $\overline{G2B}$ called gate or enable inputs. These inputs must be in the proper state to enable the selected output. Standard MSI decoder circuits are available for 2 to 4, 3 to 8, and 4 to 16 decoders. Specifics on these integrated circuits can be found in a manufacturer's data book.

Large Scale Integrated Circuits (LSI) are functional modules containing thousands of gates. LSI chips are generally used for large functions such as microprocessor chips or memory chips.

Very Large Scale Integrated Circuits (VLSI) are often used for integrating many LSI functions on a single chip. For example, VLSI microcomputers are manufactured with CPU, memory, and I/O ports all integrated on a single chip. It is always a challenge for LSI or VLSI chip designers to design the integrated circuit with a maximum amount of functionality requiring a minimum number of pins. VLSI components often come in Pin Grid Array (PGA) packages that are square and contain a matrix of pins, which connect to the circuit board. Pin grid arrays are available with several hundred pins on a single package.

Figure 3.4 MSI Dual In-Line Packages



3.2. Surface Mount Packages

Surface Mount Technology (SMT) describes a component packaging style in which components mount directly to the surface of a circuit board. Unlike standard DIP's in which IC pins must pass through a circuit board to be soldered, surface mount component pins are small metallic pads that mate with pads on the surface of the circuit board. Solder paste is used to initially bond the SMT component pins to the circuit board pads. The solder paste is then heated until reflow occurs and the electrical connection is made between the SMT component and the circuit board.

Pads can be spaced much closer than through-holes on a printed circuit board. Therefore, the pin spacing and size of SMT components are usually much smaller than an equivalent through-hole component DIP package. Pin spacing of 50 mils has become an industry standard for SMT components.

SMT components have several packaging styles, of which two are illustrated in Figure 3.5 and Figure 3.6. The D package is basically a DIP, but it is approximately one-fourth the size of a through-hole DIP.

A chip carrier is an SMT component package that is square and contains pins on all four sides. Figure 3.6 is an example of a 28-pin chip carrier package.

SMT components generally require less then one-fourth the area on a circuit board. Therefore, they allow at least four times the functionality to be placed on the same sized circuit board.

Figure 3.5 Surface Mount Technology DIP Style

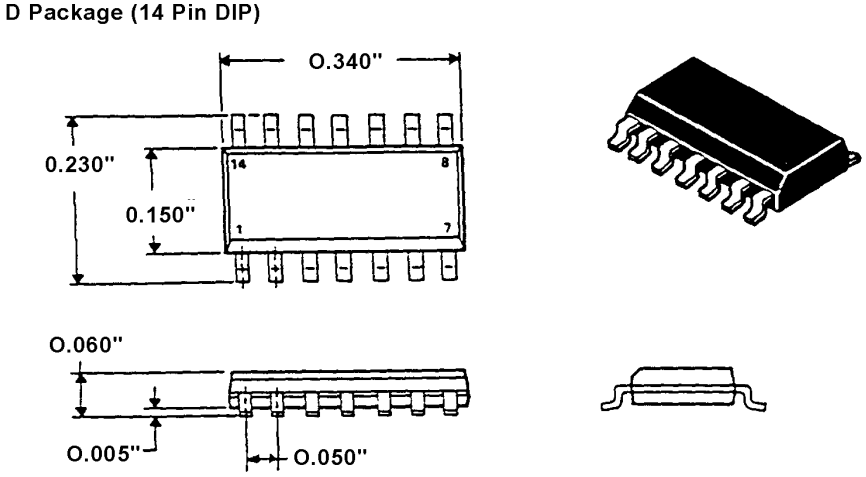
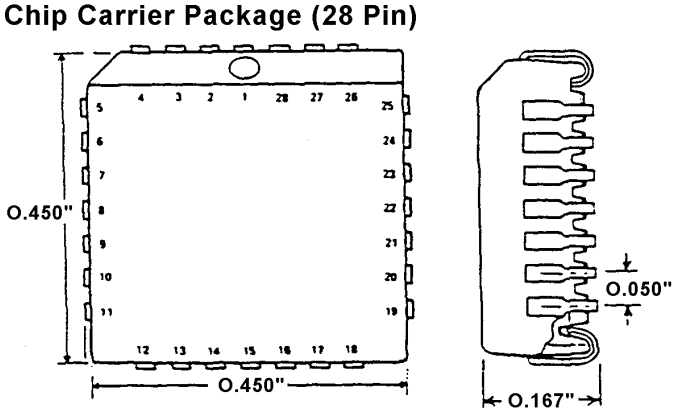


Figure 3.6 Surface Mount Technology Chip Carrier Style



3.3. Integrated Circuit Technologies

Integrated circuit chips can contain literally millions of gates all integrated together on a single chip. The active electronic device used to make each gate is the *transistor*. Several different transistor technologies exist for making integrated circuits. Each technology offers various advantages over other technologies, and advances are continually being made. Figure 3.7 describes the primary attributes of several common technologies used today, namely NMOS, CMOS, TTL, and ECL. The *density* parameter refers to the number of transistors or gates, which can be fabricated per unit area on a silicon chip. Of the technologies listed NMOS has the highest packing density and is the primary technology used when designing VLSI circuits. Power describes the amount of power consumed per gate on the chip, of which CMOS is the lowest. Speed refers to the propagation delay or switching time for a gate on the chip. There will always be a finite delay between when the

inputs of a gate change and the output stabilizes to the new value. Of the four technologies, ECL is the fastest and has the lowest propagation delay.

Figure 3.7 Properties of Several IC Technologies

Technology	Density	Power	Speed
NMOS	Very high	Medium	Medium
CMOS	Medium	Low	Medium
TTL	Low	High	High
ECL	Low	Very high	Very high

Analog circuit parameters are the voltage levels and currents that are associated with logic gate inputs and outputs. Up to now the assumption has been made that all logic gates can be interconnected without regard to analog circuit parameters. However, analog circuit parameters are important should the designer need to mix circuit technologies (i.e. TTL, CMOS, ECL) or connect many logic gate inputs to a single output. Figure 3.8 describes analog circuit parameters for several common logic families. Using these analog circuit parameters we can determine if logic families are compatible and how many logic gates can be interconnected for each family. Examples 3.1 and 3.2 demonstrate the analysis procedure.

Gates of the same technology will operate at compatible voltage levels. However, there is a limit to the number of gate inputs that can be connected to a single gate output. This limit is called the fan-out limit, which is a function of the input currents (I_{IH} and I_{IL}) and the output currents (I_{OH} and I_{OL}).

Example 3.1 describes the procedure used to determine the fan-out limit for NMOS gates. The output current must be greater than or equal to the input current requirements for both the high and low states. A gate output will source current in the high state and sink current in the low state, as depicted by the Figure 3.9a. The first step in the procedure is to determine how many inputs can be driven high, by comparing I_{OH} with I_{IH} and solving for M, as described by Step 1 of Example 3.1. Then determine the number of inputs that can be pulled low, by comparing I_{OL} with I_{IL} and solving for N, as shown in Step 2 of Example 3.1. The smaller value of M and N determines the number of inputs that can be driven with a single output. The quantities for M and N should always be rounded down to the nearest whole number.

Next consider the problem of interconnecting gates of two different technologies. Both the voltages and currents must be checked to determine if the output of one technology is compatible with the inputs of the other technology. Referring back to the threshold window concept described in Chapter 1, the gate output must provide a voltage above the input threshold window in the logic 1 state and below the input threshold window in the logic 0 state. This evaluation is performed in steps 1 and 2 of Example 3.2 by determining if V_{OH} is greater than or equal to V_{IH}, and if V_{OL} is less than or equal to V_{IL}. Next consider the compatibility of the currents for the two technologies by performing the same evaluation as was done in Example 3.1. This is accomplished in steps 3 and 4 of Example 3.2 by determining if I_{OH} is greater than or equal to I_{IH}, and if I_{OL} is greater than or equal to I_{IL}. The number of inputs which can be connected to an output is the smaller value of M and N. The

quantities for M and N should always be rounded down to the nearest whole number. If either M or N is less than 1, then the two technologies are not compatible. To be compatible, all voltages and current conditions must be met for the interconnecting gate technologies.

Example 3.1: For NMOS logic gates, what is the maximum number of inputs that can be driven by an output?

Step 1) Compare I_{OH} with I_{IH} $200 \times 10^{-6} = M (2.5 \times 10^{-6})$ Solving M = 80
Therefore, one output can source current for 80 inputs in the high state

Step 2) Compare I_{OL} with I_{IL} $1.6 \times 10^{-3} = N (2.5 \times 10^{-6})$ Solving N = 640
Therefore, one output can sink current for 640 inputs in the low state

Step 3) The smaller value of M and N determines the number of inputs which can be driven with a single output. The quantities for M and N should always be rounded down to the nearest whole number. For this problem M is less than N.

Answer: One output can drive up to 80 inputs.

Figure 3.8 Electrical Characteristics of Four Semiconductor Technologies

Characteristic	Technology			
	TTL	ECL	NMOS	CMOS
V_{OL} (volts)	0.4	-1.60	0.4	0.1
V_{OH} (volts)	2.4	-0.74	2.4	4.9
V_{IL} (volts)	0.8	-1.45	0.8	1.2
V_{IH} (volts)	2.0	-0.74	2.0	3.5
I_{OL} (amperes)	16×10^{-3}	50×10^{-3}	1.6×10^{-3}	0.4×10^{-3}
I_{OH} (amperes)	400×10^{-6}	50×10^{-3}	200×10^{-6}	500×10^{-6}
I_{IL} (amperes)	1.6×10^{-3}	0.3×10^{-6}	2.5×10^{-6}	1×10^{-6}
I_{IH} (amperes)	40×10^{-6}	265×10^{-6}	2.5×10^{-6}	1×10^{-6}

Note: The above values are typical and variations exist between gates of the same family.

Example 3.2: Can the output of a CMOS gate drive a TTL gate input?

Step 1) Compare High Level Voltages.
Is CMOS V_{OH} greater than or equal to TTL V_{IH} ?
YES 4.9 V $\exists$ 2.0 V

Step 2) Compare Low Level Voltages.
Is CMOS V_{OL} less than or equal to TTL V_{IL} ?
YES 0.1 V # 0.8 V

Step 3) Compare High Level Currents.
Is CMOS I_{OH} greater than or equal to TTL I_{IH} ?
YES 500×10^{-6} A $\exists$ 40×10^{-6} A, and M = 12.5.
Therefore, one output can source current for 12 inputs.

Step 4) Compare Low Level Currents.
Is CMOS I_{OL} greater than or equal to TTL I_{IL} ?
NO 0.4×10^{-3} A $\not\exists$ 1.6×10^{-3} A. and N = 0.25.
Therefore, one output can not sink current for one input.

Step 5) Have all four conditions been met?
No, The CMOS Output Cannot Pull The TTL Input Low.

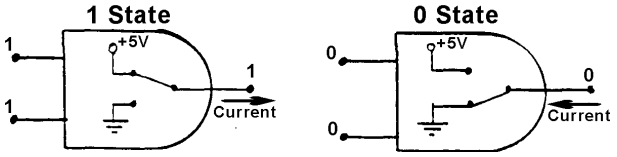
3.4. Device Outputs

When designing computers it is often desirable to have multiple digital logic device outputs connected to the same conductive path. This minimizes both the amount of interconnecting wires between integrated circuit pins and the complexity of the digital circuits. Circuit paths that have more than one output connected to them and are used to pass binary information are classified as *buses*. Only one output may have access or "talk" on the bus at a time.

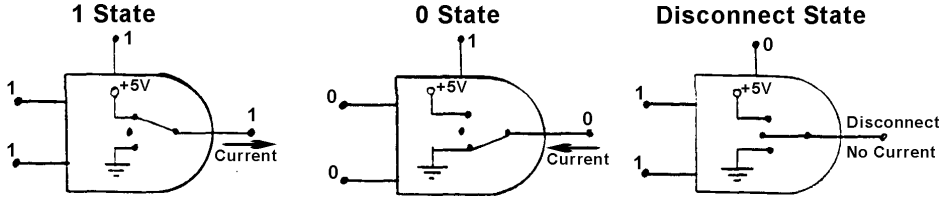
Standard logic devices have transistor switch outputs as depicted in Figure 3.9a. In the ‘1’ state, the output sources current, with current flowing from the output to the connecting inputs. This drives the inputs to the high or ‘1’ state. In the logic ‘0’ state, the output sinks current, such that current flows from the connecting inputs into the output. This will drive all inputs to the ‘0’ state. This standard configuration is often referred to as a totem-pole output. Totem-pole outputs cannot be connected together to form buses. When the output states of two connected totem-pole outputs are different, the low state output may damage the high state output. This problem is remedied by using either tri- state or open-collector output gates.

Figure 3.9 Switch Analogies for Output Devices

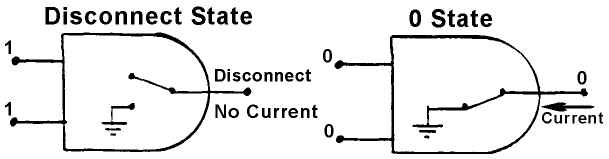
a) Standard Gate Output



b) Tri-state Gate Output



c) Open-Collector Gate Output



3.4.1. Tri-state Gates

The tri-state gate has transistor switch outputs, which can be modeled as a 3-position switch as depicted in Figure 3.9b. Each position of the switch represents a different state (either 0, 1, or Disconnect). Notice that an additional input is provided on the top of the gate, which is called the control input. The control input is used to selectively disconnect or drive the output. The truth table for a tri-state NAND gate is shown in Figure 3.10a. When the Control input is 1, the gate functions normally as a 2-input NAND gate. However, when the Control input is 0, the gate output is effectively disconnected. This disconnect state is often referred to as the high-impedance state (High-Z State).

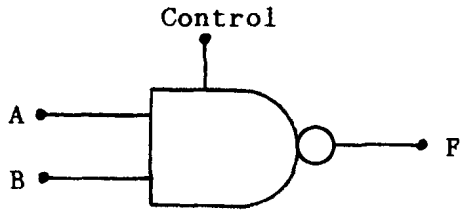
Figure 3.10b illustrates a tri-state buffer with a negative logic control input. The tri-state buffer is symbolized as a triangle pointing to the output (similar to the NOT gate without the inversion circle). As described by the truth table, the output of the tri-state gate is disconnected when the control input is in the 1 state. When the control input is in the 0 state, the output state is the same value as the input A.

When a bus is constructed using tri-state gates, only one tri-state gate can be active at a time. Inactive outputs must be disconnected from the bus by forcing their control inputs to the inactive state.

Figure 3.10 Tri-state Gate Truth Table and Symbolic Representation

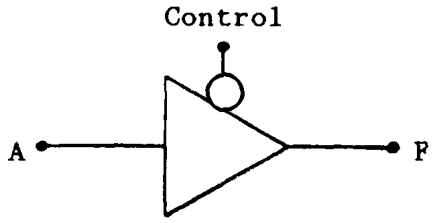
(a)

Control	A	B	F
0	0	0	Disconnect
0	0	1	Disconnect
0	1	0	Disconnect
0	1	1	Disconnect
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



(b)

Control	A	F
0	0	0
0	1	1
1	0	Disconnect
1	1	Disconnect



3.4.2. Open-Collector Gates

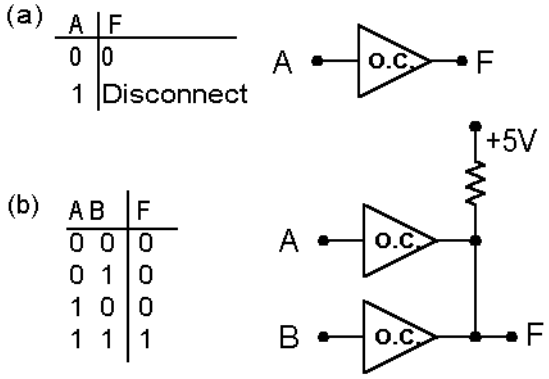
Open-collector gates function as standard gates, but have two possible output values either Disconnect or the 0 state. The output of an open-collector gate contains only one switching transistor which switches the output to ground as shown in Figure 3.9c. Current cannot flow out of an open-collector output in the Disconnect state; therefore, $I_{OH} = 0$ and the output is effectively disconnected. When the output is in the 0 state, the output will sink current to ground and all connecting inputs will be driven to the 0 state. The truth tables for open-collector gates have the same form as standard gates; however, when the output is 1 for a standard gate it will be disconnected for an open-collector gate. The truth table for an open-collector buffer is shown in Figure 3.11a.

Since the output of an open collector gate cannot source current in the 1 state (as a standard output), an external voltage source must be connected to the output. This voltage source is required to drive any connecting inputs to a 1 state voltage level (i.e. 2 to 5 volts). A pull-up resistor must be connected between the voltage source (i.e. +5V) and the open-collector output. The pull-up resistor is used to limit the current entering the output in the low state.

Illustrated in Figure 3.11b is a simple circuit composed two buffers, which are used to generate an output bus F. The circuit has two buffers with open-collector outputs and one pull-up resistor. If either or both inputs A and B are in the 0 state, then at least one open-collector output will sink current and drive output bus F to the 0 state. Only when both inputs A and B are in the 1 state, will no current flow and output F will be pulled up to the 1 state. The truth table for this circuit is shown in Figure 3.11b. As depicted by the truth table, this circuit is often called a wire-AND circuit.

Open-collector gates can be used to construct buses, with more than one output connected to the same conductive path. Inactive open-collector outputs must remain in the disconnect state while the active output transmits its digital signal. In Figure 3.11b, if the buffer connected to Input A were active and the other buffer inactive, then Input B must be in the 1 state while Input A transmits data.

Figure 3.11 Open-Collector Buffers Configured as Wire-AND Circuit



3.4.3. Drivers

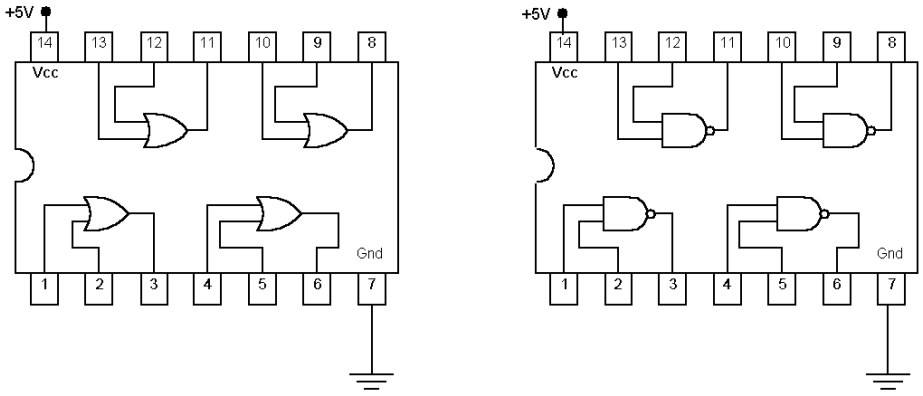
Drivers are used as an output device when the application requires current and voltages that exceed those available with standard logic devices. Drivers are used as output devices for applications such as lights, displays, and relays.

High voltage open-collector output gates are often used as drivers for such applications as turning on light emitting diodes (LED). These gates will sink more current than will standard gates, because their output transistors are designed to handle more current.

When using drivers, it is important to examine both the input compatibility and the output rating for both voltage and currents.

Problem Set

1. Using the integrated circuits shown, draw lines between the pins of the DIPs to construct a digital circuit which will implement the Boolean expression
   $X = A + \overline{B\,C} + \overline{D}$ for Output X based on inputs A, B, C, and D. The power and ground connections are drawn.



2. Using the integrated circuits shown in Figure 3.2, draw circuits for the following Boolean expressions.
   a) $A + B\,C$
   b) $A\,(B + C)\,D$
   c) $A\,\overline{B}\,C + B\,\overline{C} + A$
   d) $(A \oplus B) \oplus C$

3. A digital circuit is designed in which power consumption is minimized. Which one of the integrated circuit technologies should be considered first?

4. A digital circuit is designed which requires very high speed switching. Which integrated circuit technologies should be considered first?

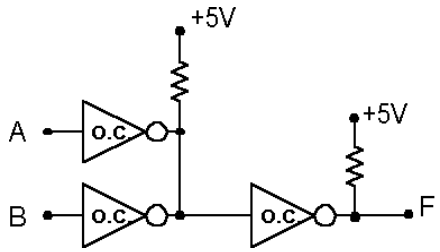
5. A VLSI digital circuit is designed which requires a very high density of circuitry to be placed on a single silicon chip. Which integrated circuit technologies should be considered first?

6. Using the table of Figure 3.8, determine if an NMOS output is compatible with a TTL input. If they are compatible, determine how many TTL inputs can be driven by a single NMOS output.

7. Determine how many CMOS inputs can be driven by a single CMOS output.

8. Draw the logic gate symbol and truth table for the following tri-state gates.
   a) Tri-state Inverter
   b) Tri-state 2-Input OR Gate
   c) Tri-state 3-Input NAND Gate

9. Draw the truth table for the following circuit which uses Inverters with open collector outputs. Why is this circuit called a Wire-ORed circuit?

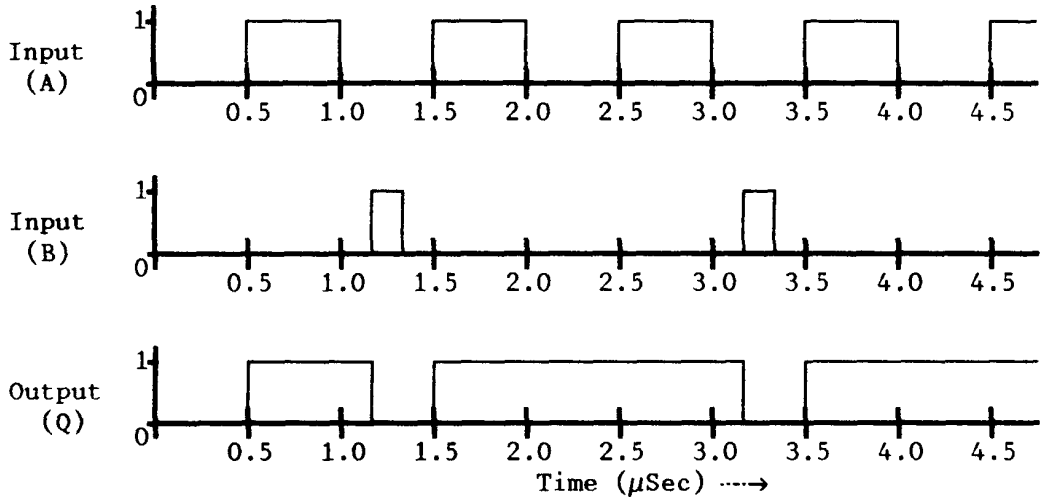


Chapter 4. Sequential Logic

Sequential logic devices are used to construct sequential circuits such as computer memories, counters, frequency dividers, and data converters. Sequential circuits differ from combinational circuits because their outputs are determined not only by the current state of the inputs but also by past behavior.

Timing diagrams describe the state of input and output signals of a digital circuit as they vary with time. Timing diagrams are used to analyze sequential circuits because sequential logic devices are a function of not only the current state of the inputs, but also the past behavior of the output. Figure 4.1 illustrates a timing diagram for a sequential circuit with two inputs (A and B) and one output (Q). Time is located on the horizontal axis while the state of the signal is shown on the vertical axis. The logic variables (A, B, and Q) are represented as waveforms in either the 1 or 0 state as they vary in time.

Figure 4.1 Timing Diagram



Most sequential logic circuits utilize a clock signal to synchronize components. Typically, clock signals are either a periodic square wave (Input A of Figure 4.1) or a periodic pulse waveform (Input B of Figure 4.1), for which the period is constant and the transition time between states is minimized. The period of the waveform is the time required to complete one full cycle. In Figure 4.1, Input A has a period of 1.0 μSeconds and Input B has a period of 2.0 μSeconds. Often the clock signal is specified not by its period, but by its frequency. *Frequency* is specified in units of cycles per second, which is usually called Hertz (Hz) after the 19th century physicist Heinrich Rudolf Hertz. Computer circuits have very high clock frequencies, so the terms Kilo-Hertz (KHz), Mega-Hertz (MHz), and Giga-Hertz (GHz) are used to specify units of thousand cycles per second, million cycles per second, and billion cycles per second respectively.

The frequency of the waveform is found by taking the reciprocal of its period. The frequencies of inputs A and B are determined to be 1.0 MHz and 500 KHz, as shown below:

$$\text{Frequency of A} = f_A = \frac{1}{\text{Period}} = \frac{1}{1.0 \times 10^{-6}\ \text{Sec}} = 1.0 \times 10^{6}\ \frac{\text{Cycles}}{\text{Second}} = 1.0\ \text{MHz}$$

$$\text{Frequency of B} = f_B = \frac{1}{\text{Period}} = \frac{1}{2.0 \times 10^{-6}\ \text{Sec}} = 500 \times 10^{3}\ \frac{\text{Cycles}}{\text{Second}} = 500\ \text{KHz}$$

4.1. Sequential Logic Devices

Sequential logic devices utilize an input signal called a *clock* input to trigger themselves at specific moments in time. The output of the device can only change when a trigger condition occurs. Trigger conditions can be either level sensitive or edge sensitive.

Level sensitive sequential components are called latches, and trigger in the one state. The D-latch is an example of a level triggered device.

Edge sensitive sequential components are called flip-flops. Three flip-flops will be discussed: J-K, T, and D flip-flops. Flip-flops are either positive or negative edge triggered. A positive edge occurs when the clock signal changes from the '0' to '1' state, as shown in Figure 4.1 for Input A at times 0.5, 1.5, 2.5, 3.5, and 4.5 microseconds. A negative edge occurs when the clock changes from the '1' to '0' state, as shown in Figure 4.1 for Input A at times 1.0, 2.0, 3.0, and 4.0 microseconds. Flip-flop inputs are read at the instant a transition edge occurs. The output may change state the instant after the transition edge occurs.

4.1.1. J-K Flip-Flop

The J-K flip-flop has three inputs (J, K, and C) and two outputs (Q and $\overline{Q}$), as shown in Figure 4.2a. The output $\overline{Q}$ is always the opposite state of output Q.

Transition tables describe what happens to the latch or flip-flop output after a clock trigger condition occurs. An example of a transition table is shown if Figure 4.2b. The output after a clock trigger condition (Q^+) is a function of both the current state of the inputs at that instant and the state of the output before the trigger condition occurs (Q^-).

The transition table looks like a truth table, except the output Q is a function of both the current state of the inputs and the past state of the output. The output column of the transition table, denoted by Q^+, is the value of the output (Q) after some instant in time. The symbol Q^- denotes the output value before that instant in time. The output can only change at the instant after a clock trigger condition has occurred; otherwise, the other inputs are ignored and the output cannot change.

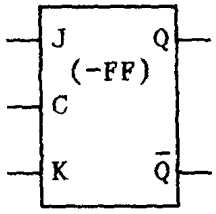
The transition table for the J-K flip-flop is shown in Figure 4.2b. The J and K inputs are examined the instant the clock trigger edge occurs, in order to determine the state of output Q the instant after the clock transition edge.

When the J and K inputs are both '0' (at a clock trigger edge) the output Q remains unchanged. If J=0 and K=1, then output Q is reset to '0'. If J=1 and K=0, then output Q is set to one. If both J and K inputs are in a one state at the clock trigger edge, then output Q will change to the opposite state.

This condition is described by the last row of the transition table. The symbol $\overline{Q}^-$ of the output column signifies that if the output Q was a one before the trigger edge, it becomes a '0' after; and a '0' before the trigger edge results in a one after. This is often described as an output toggle.

Figure 4.2 J-K Flip-Flop (- Edge Triggered)

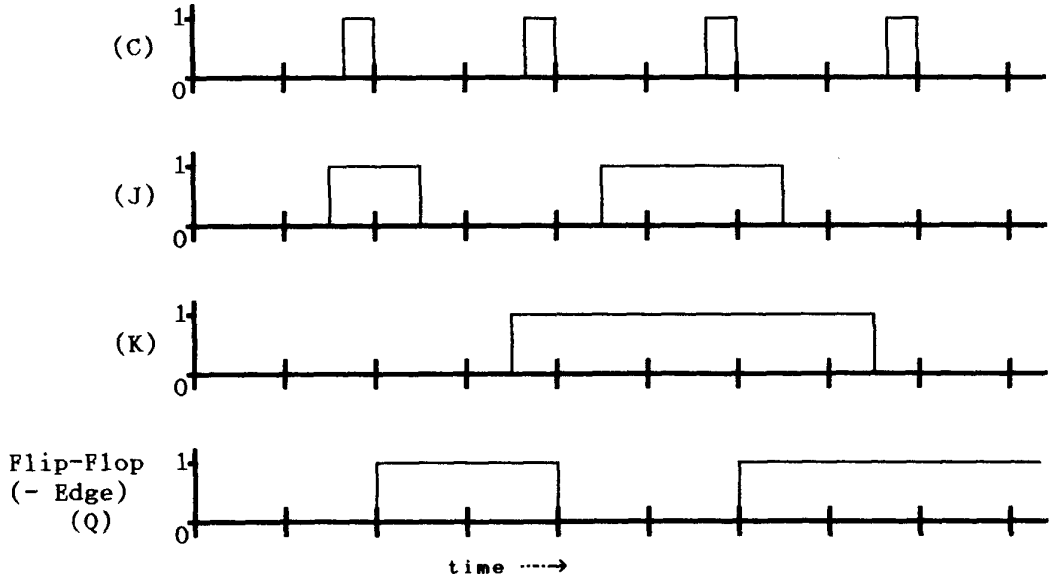
a) Device Symbol b) Transition Table



J	K	Q+
0	0	Q-
0	1	0
1	0	1
1	1	$\overline{Q}$-

J-K flip-flops come in either negative edge or positive edge triggered varieties. If the flip-flop is positive edge triggered, it is denoted by the symbol "+FF". If the flip-flop is negative edge triggered, it is denoted by the symbol "-FF". A timing diagram for a negative edge triggered J-K flip-flop is illustrated in Figure 4.3. As with logic gates, some time delay does exist between the time when the inputs are read and the time the output changes state. This is called the *propagation delay* of the flip-flop.

Figure 4.3 Timing Diagram for the J-K Flip-Flop



4.1.2. T Flip-Flop

The T flip-flop (Toggle flip-flop) has one input (T) and two outputs (Q and $\overline{Q}$), as shown by its device symbol in Figure 4.4b. It can be constructed using a J-K flip-flop with both the J and K inputs connected to logic '1' state and using the clock input as the T input.

T flip-flops are either positive or negative edge triggered. The output (Q) will toggle states only when a trigger edge appears on the T input. If the output (Q) was '1', it becomes '0', and if it was a '0', it becomes '1'.

Otherwise, when the trigger edge is not present, the output state will not change.

The timing diagram of Figure 4.5 illustrates the operation of a positive edge triggered T flip-flop for a non-periodic waveform. When the T input of a T flip-flop is connected to a periodic square wave, it operates as a divide by 2 frequency divider with the output waveform being half the frequency of the input.

Figure 4.4 The T Flip-Flop (+Edge Triggered)

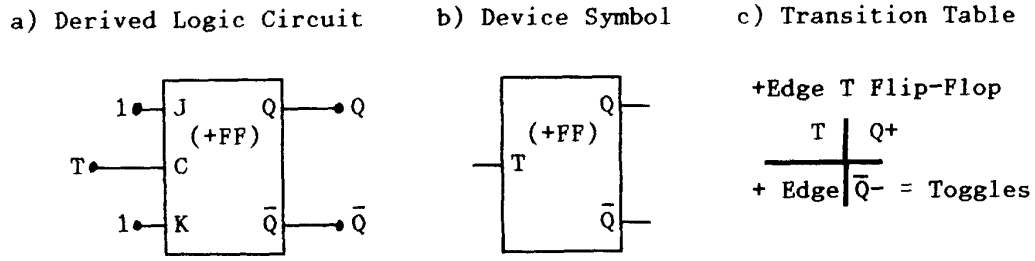
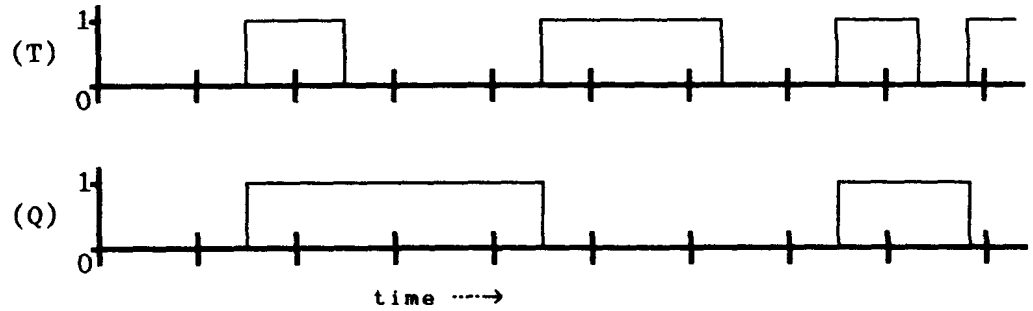


Figure 4.5 Timing Diagram for a Positive Edge Triggered T Flip-Flop



4.1.3. D Flip-Flop and Latch

The D flip-flop (Data flip-flop) can be used to store one bit of data. It is constructed by using a J-K flip-flop with an inverter connected between inputs J and K, as illustrated in Figure 4.6a. The D flip-flop has a data input (D), a clock input (C), and two outputs (Q and $\overline{Q}$). When a trigger edge occurs on the clock input, the state of output Q will become the state of input D. Therefore, the state of the D input is stored at output Q until the next clock trigger edge occurs. This relationship is depicted in the transition table of Figure 4.6c. Timing diagrams for both positive edge and negative edge triggered D flip-flops are illustrated in Figure 4.7.

D latches differs from a D flip-flops in that they are level triggered. The transition table for the D latch is the same as the D flip-flop. When the clock input is in the 1 state the Q output of a D latch will be the same state as the D input. When the clock input is in the 0 state the data at the output will be "latched" and will not change. A timing diagram for the D latch is illustrated in the last diagram of Figure 4.7.

4.1.4. Preset And Clear Inputs

Flip-flops commonly have preset and clear inputs, as illustrated in Figure 4.8. These inputs are shown on the top and bottom of the flip-flop and are symbolized with an inversion circle by the labeled input. Sometimes these inputs are abbreviated such that $\overline{R}$ = Clear and $\overline{S}$

= Preset. Preset and clear inputs are *unclocked* negative logic inputs and take priority over all other inputs.

When the Preset or $\overline{S}$ input is in the '0' state, the output Q is set to '1'. When the Clear or $\overline{R}$ input is in the '0' state, the output Q is cleared to '0'. At no time should both the clear and preset inputs be activated simultaneously, as the output state is then undefined.

Figure 4.6 D Flip-Flop (- Edge Triggered)

a) Derived Logic Circuit b) Device Symbol c) Transition Table

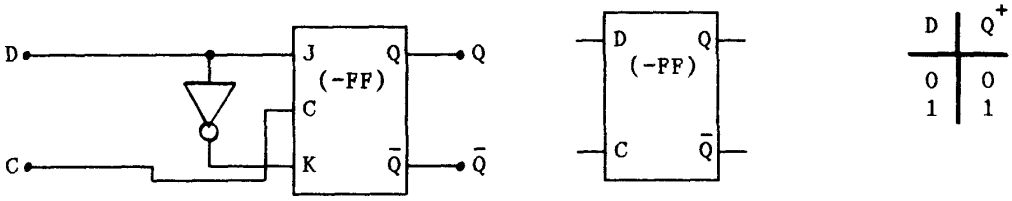


Figure 4.7 Timing Diagram for the D Flip-Flop and D Latch

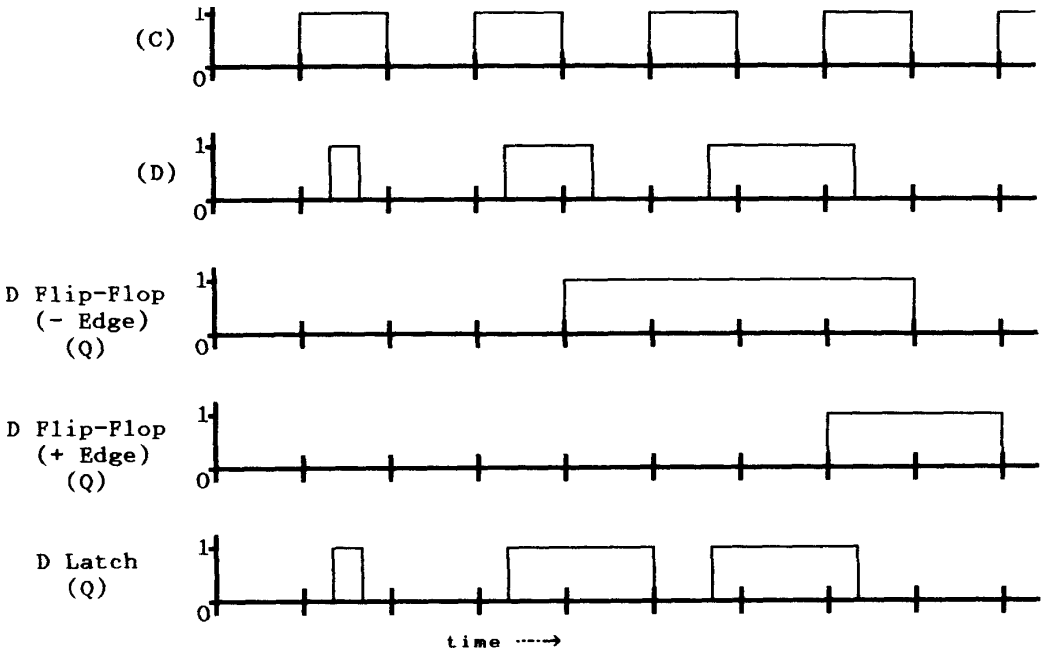
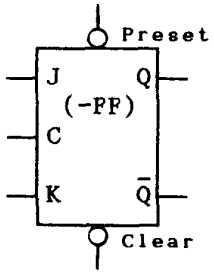


Figure 4.8 J-K Flip-Flop with Preset and Clear Inputs



4.2. Timing Diagram Construction For Sequential Circuits

Timing diagrams have been constructed for single flip-flops and latches in Figures 4.1 through 4.7. Given timing diagrams for the inputs, the output timing diagram can be drawn for any sequential logic device. The procedure for making a timing diagram is outlined below:

Step 1: Determine the type of sequential logic device, including whether it is level triggered, or positive or negative edge triggered.

Step 2: Mark triggered clock edges (flip-flop) or trigger area (latches) where a transition of output may occur.

Step 3: Determine the state of all inputs to the sequential logic device just before a trigger condition occurs.

Step 4: Use the input values and the transition table for the given sequential logic device to determine the value of output Q after the trigger condition. Mark the state of the outputs on the timing diagram, until the next transition of the output can occur. Go back to Step 3.

Note: Q^- = value of Q before transition. Q^+ = value of Q after transition.

This same procedure can be applied to construction of timing diagrams for sequential circuits with more than one sequential device. The procedure is applied by recognizing that the outputs of some sequential devices are connected to the inputs of other sequential devices. Therefore, after the timing diagram for an output of one device has been found, it may be used as an input to construct the timing diagram for the connecting device. This process is similar to the process used to evaluate combinational circuits using truth tables.

Verify the timing diagrams for single sequential logic devices of Figures 4.1 through 4.7. Then examine and verify the timing diagrams for the sequential circuits discussed in Section 4.3.

4.3. Sequential Circuits

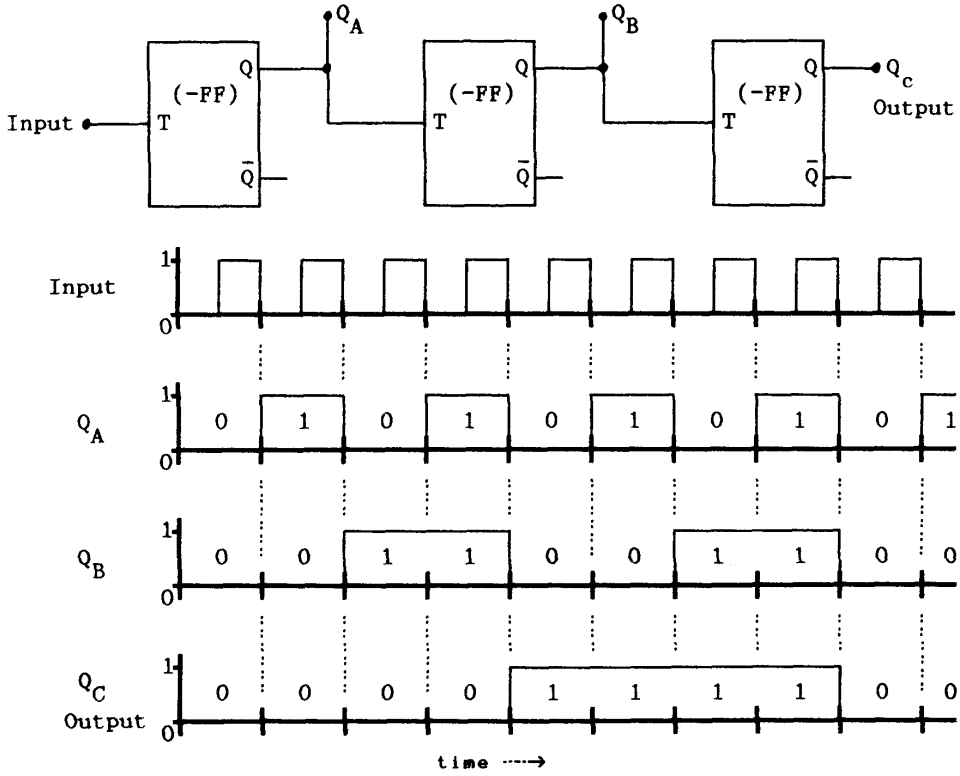
Sequential circuits are constructed by interconnecting flip-flops, latches, and logic gates. Sequential circuits are used to perform many operations required by a computer. Discussed in this section are several simple sequential circuits that are found in computer hardware. These sequential circuits include frequency dividers, counters, data registers, shift registers, and data converters.

4.3.1. Frequency Dividers and Counters

Frequency divider circuits are used to generate an output frequency that is a fraction of the input frequency for a periodic waveform. Either T flip-flops or J-K flip-flops can be used to construct frequency divider circuits. When flip-flops are used to make a frequency divider circuit, the frequency of the input is divided by an integer value to generate the output frequency.

The circuit of Figure 4.9 is a divide-by-eight frequency divider. It is constructed using three T flip-flops configured as shown. Each T flip-flop will generate, on its output, a divide-by-two operation of the input frequency. Therefore, a circuit in the configuration of Figure 4.9, with ***n*** T flip-flops will be a divide-by-2^n frequency divider. This circuit could also be constructed using J-K flip-flops with both the J and K inputs connected to a logic '1' state.

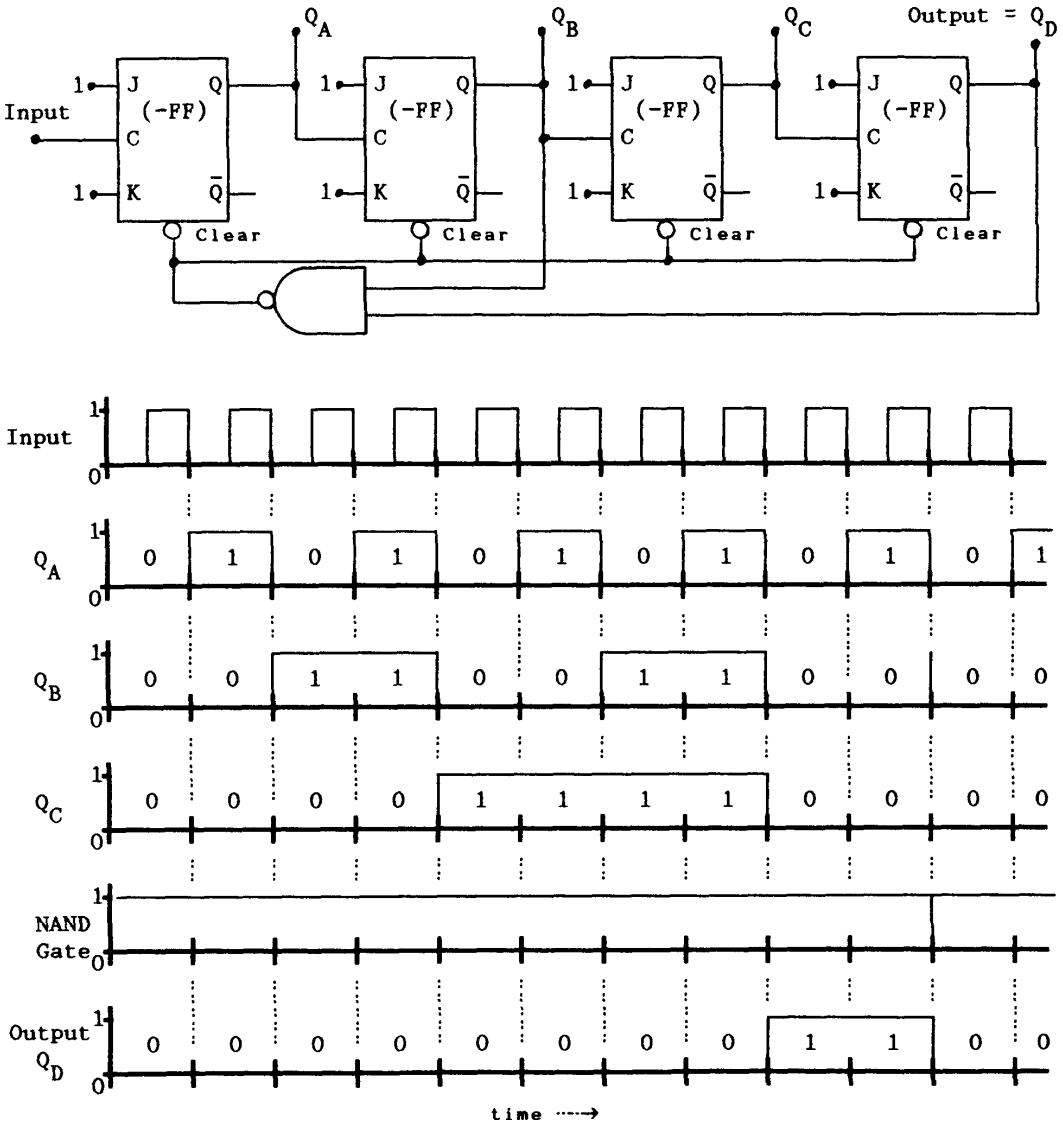
Figure 4.9 Divide-by-Eight Frequency Divider or Three Bit Binary Counter



J-K flip-flops with clear inputs are used to construct the divide-by-ten frequency divider of Figure 4.10. Note that the output Q will complete one full period after ten periods of the input signal. The NAND gate is used to reset the circuit after ten input cycles. The propagation delays for both the logic gate and the flip-flops are relatively small when compared with the input frequency. Therefore, after every ten clock cycles of the input, the output of the NAND gate pulses from the 1 to 0 and back to 1 state.

A frequency divider circuit, that is configured like Figure 4.9, can be used as a *binary counter* by accessing the output of each flip-flop. The divide-by-eight frequency divider circuit of Figure 4.9 is also a 3-bit binary counter. Output Q_A represents the least significant bit, Q_B the next most significant bit, and Q_C the most significant bit of a three bit binary number. Notice on the timing diagram of Figure 4.9, when the outputs are read vertically after each negative edge, the output states (Q_A, Q_B, and Q_C) represent a binary counting sequence (000 through 111). This three bit binary counter has eight distinct states of its three outputs. The circuit of Figure 4.10 counts in binary from zero to nine. The counter has ten distinct states of its outputs and for this reason it is called a *decade counter*.

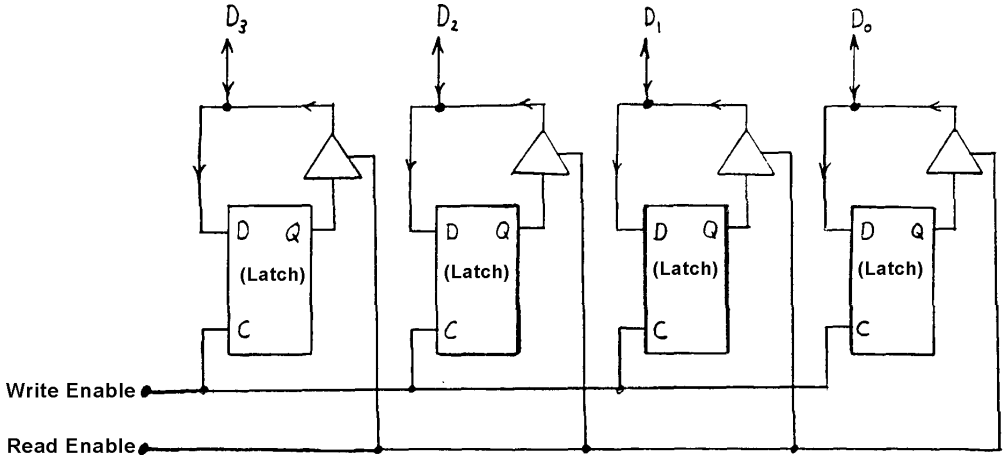
Figure 4.10 Divide-by-Ten Frequency Divider or Decade Counter



4.3.2. Data Registers

Data registers are used to store and retrieve binary data. Data registers are constructed using D-latches, as illustrated in Figure 4.11. A 4-bit data register can be used to store and retrieve four bits of data from a bi-directional data bus. In a store or write operation, data present on the data bus is stored in the D-latches by applying a trigger pulse at the clock inputs. In a retrieve or read operation, data is gated from the outputs of the latches through the tri-state buffers to the data bus. The read operation uses the bi-directional data bus as register outputs, and the write operation uses the data bus as register inputs.

Figure 4.11 Four Bit Data Register



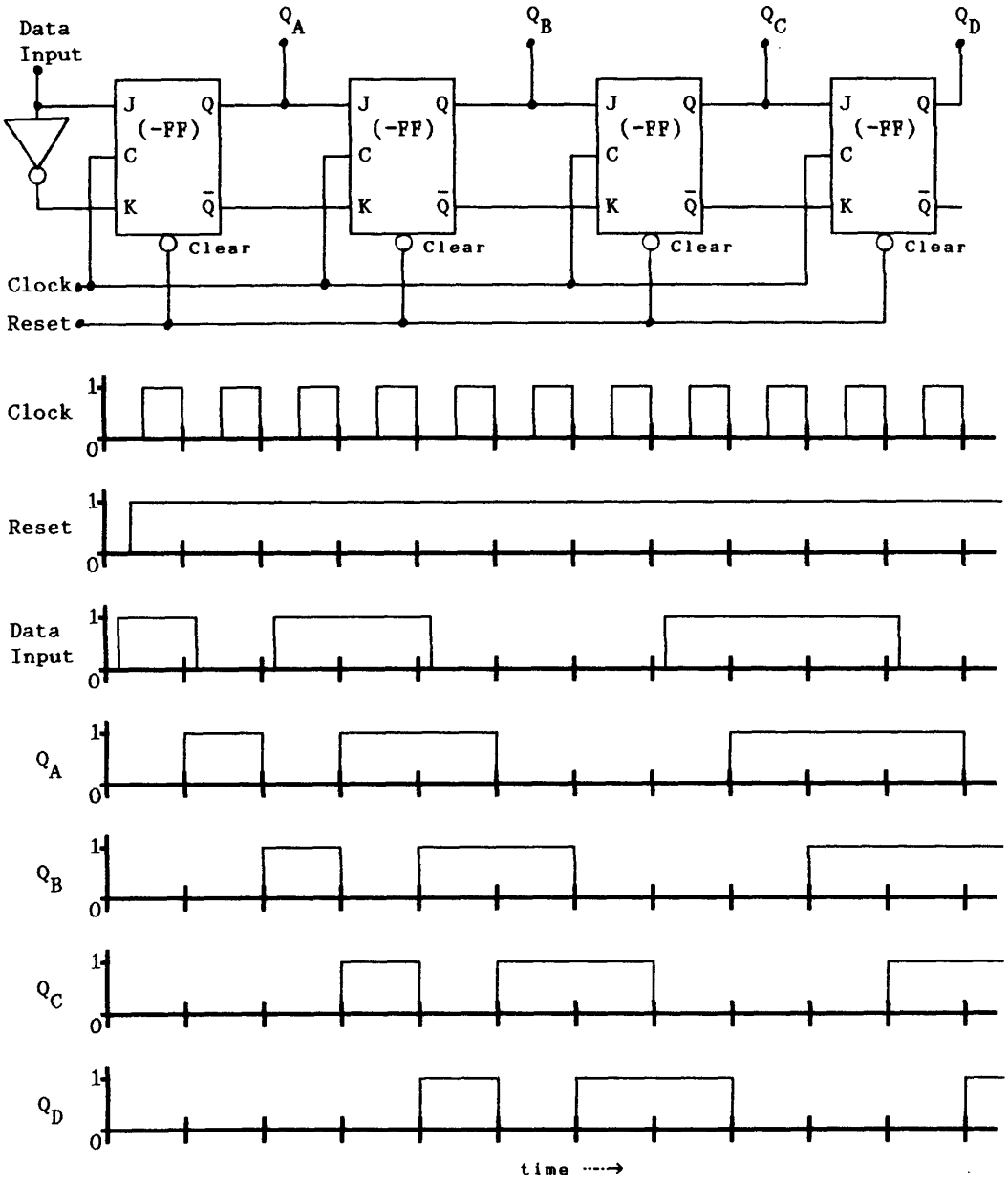
4.3.3. Shift Registers

Shift Registers are a class of data registers that are used to shift a group of bits to the left or right. Shown in Figure 4.12 is a shift register, which will shift a group of four bits to the right. J-K flip-flops are used to construct this shift register such that their J and K inputs will always be opposite in state. Since this is the same configuration used to construct a D flip-flop, this shift register could also be constructed using D flip-flops.

The operation of the shift register can be best understood by examining and verifying the shift register timing diagram of Figure 4.12. The Reset input is used to initially clear all flip-flop outputs. The input value is then stored in the left-most flip-flop while its previous output value is stored in the next flip-flop to the right. The timing diagrams for the outputs illustrate the data shifting properties of the shift register. Notice the data at output Q_D is the same as the data on the input; however, it will be delayed close to four clock cycles. Shift registers can be used as a delay device with the maximum delay time equal to the number of flip-flops multiplied by the clock period.

The clock inputs of every flip-flop of the shift register are tied together so the flip-flops trigger at exactly the same time. Sequential circuits with all flip-flop clock inputs connected to one clock source are called *synchronous* circuits. With synchronous circuits, the clock input is used to synchronize the flip-flops to perform the desired function. Figures 4.11 and 4.12 are considered synchronous circuits. Sequential circuits with the clock signals not coming from a common source are called *asynchronous* circuits. The counter circuits of Figures 4.9 and 4.10 are examples of asynchronous circuits.

Figure 4.12 Four Bit Shift Register



4.3.4. Data Converters

Data can be transmitted one bit at a time or as a group of bits. When data is transmitted one bit at a time it is called serial data. Only one wire is required to transmit serial data. When several bits are transmitted simultaneously through parallel conductive paths, it is called parallel data transmission. A digital device that is used to perform conversions between parallel and serial data formats is called a data converter.

Generally, the number of bits per second of data that can be transmitted on a single conductive path is constrained by its conductive media (e.g. wire, printed circuit path, and fiber optic cable). When the required data transmission rate exceeds the maximum bit rate for the media, then parallel data transmission must be used instead of serial data transmission. Four bit parallel data can transmit four times as many bits per second as serial data for the same speed media. Of course, four parallel conductive paths would be required for parallel data, while only one is required for serial data transmission.

4.3.4.1. Serial to Parallel Data Converter

The shift register of Figure 4.12 can be used as a serial to parallel converter. The single input shown is used as the serial input. The four flip-flop outputs are used as the four parallel outputs of the data converter. To read the parallel data properly, the data converter outputs must be read at one quarter the frequency at which the data is loaded into the serial input. Data should only be read when the outputs Q_A, Q_B, Q_C, and Q_D are stable. This can be accomplished by reading the data on the fifth, ninth, and thirteenth positive edges of the clock.

4.3.4.2. Parallel to Serial Data Converter

A 4-bit parallel to serial data converter can be constructed using four D flip-flops and four NAND gates as shown in Figure 4.13. The serial data is transmitted at four times the frequency at which the parallel data is loaded. This can be verified by comparing the Clock input waveform with the Load input waveform.

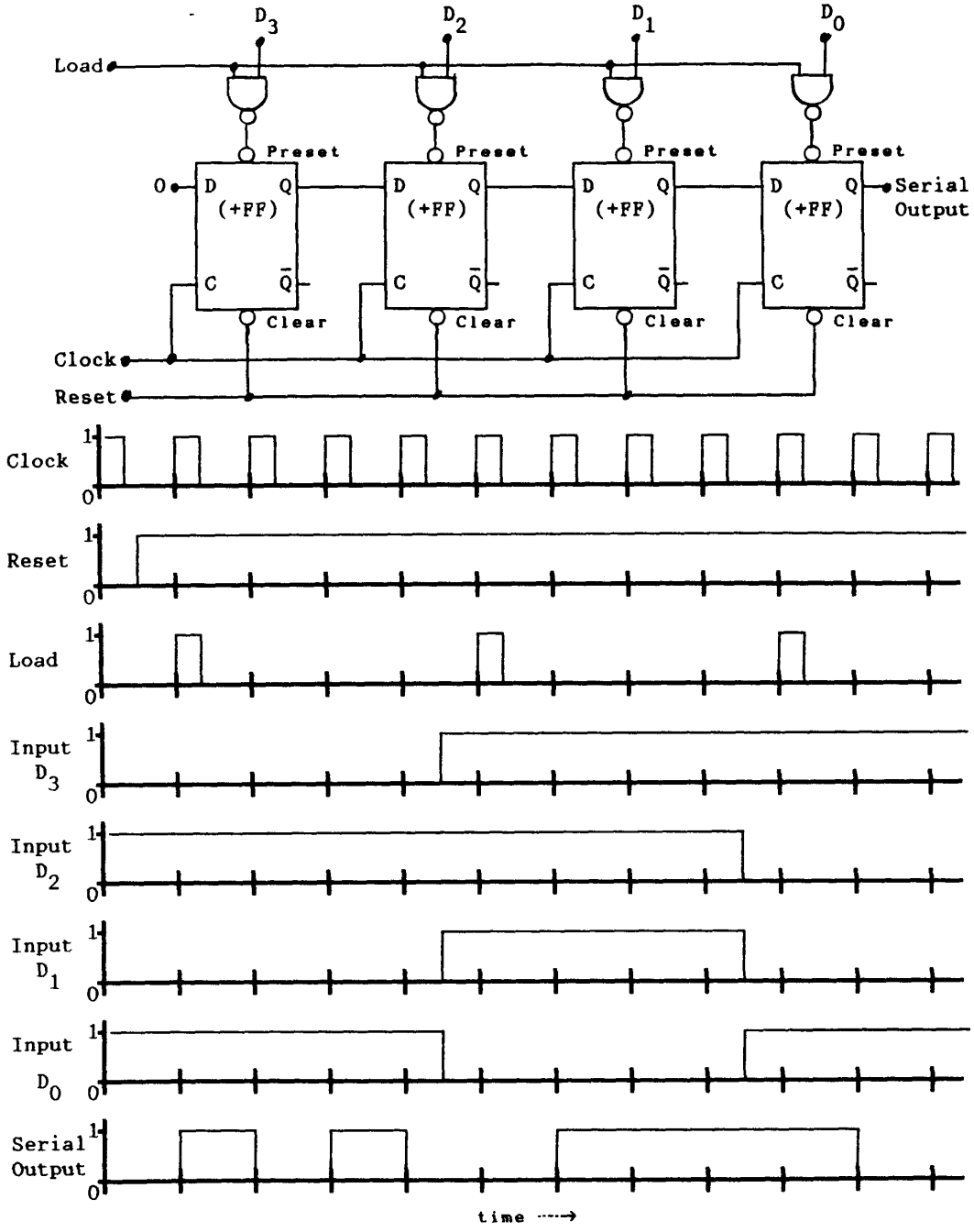
The 4-bit parallel to serial converter is initially reset to generate a '0' state at the outputs of all flip-flops. The Load input is then pulsed high to load the flip-flops with the data from the parallel inputs (D_3 through D_0).

The Clock input to this synchronous circuit is a periodic pulse waveform, which is used to synchronize the parallel to serial data transfer. The flip-flops are positive edge triggered. Serial data should be read at the negative edges of the Clock input because the Serial output is stable at the negative edges.

4.4. Sequential Integrated Circuits

Flip-Flops and latches are manufactured as SSI circuit chips. The sequential circuits discussed in Section 4.4 are also available as MSI circuits. When designing a sequential circuit it is best to first determine which sequential functions can be performed using MSI chips and then use individual flip-flops (SSI chips) to complete the design. Descriptions of these integrated circuits can be found in a manufacturers data book, or downloaded from the Internet as a PDF document (e.g. http://www.ti.com, http://www.national.com)

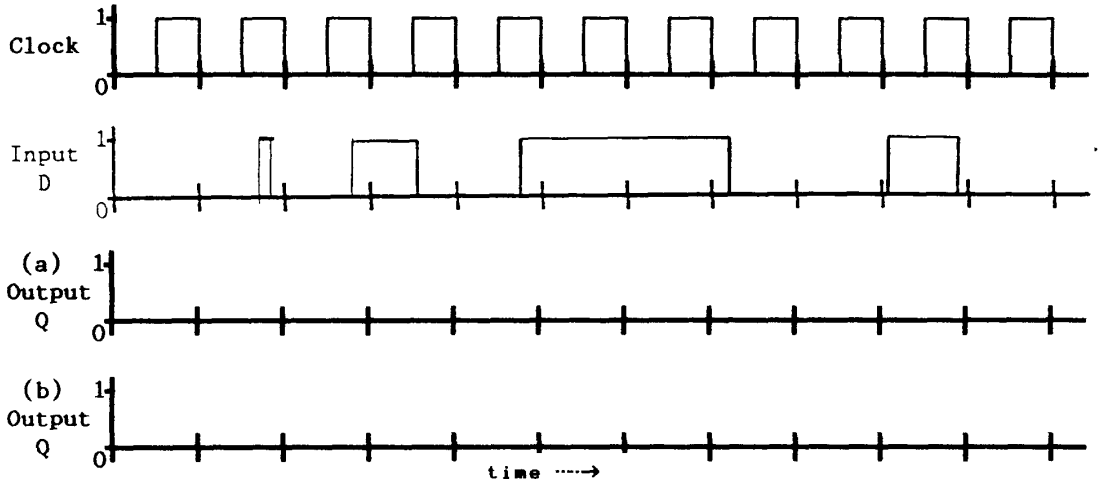
Figure 4.13 Parallel to Serial Data Converter



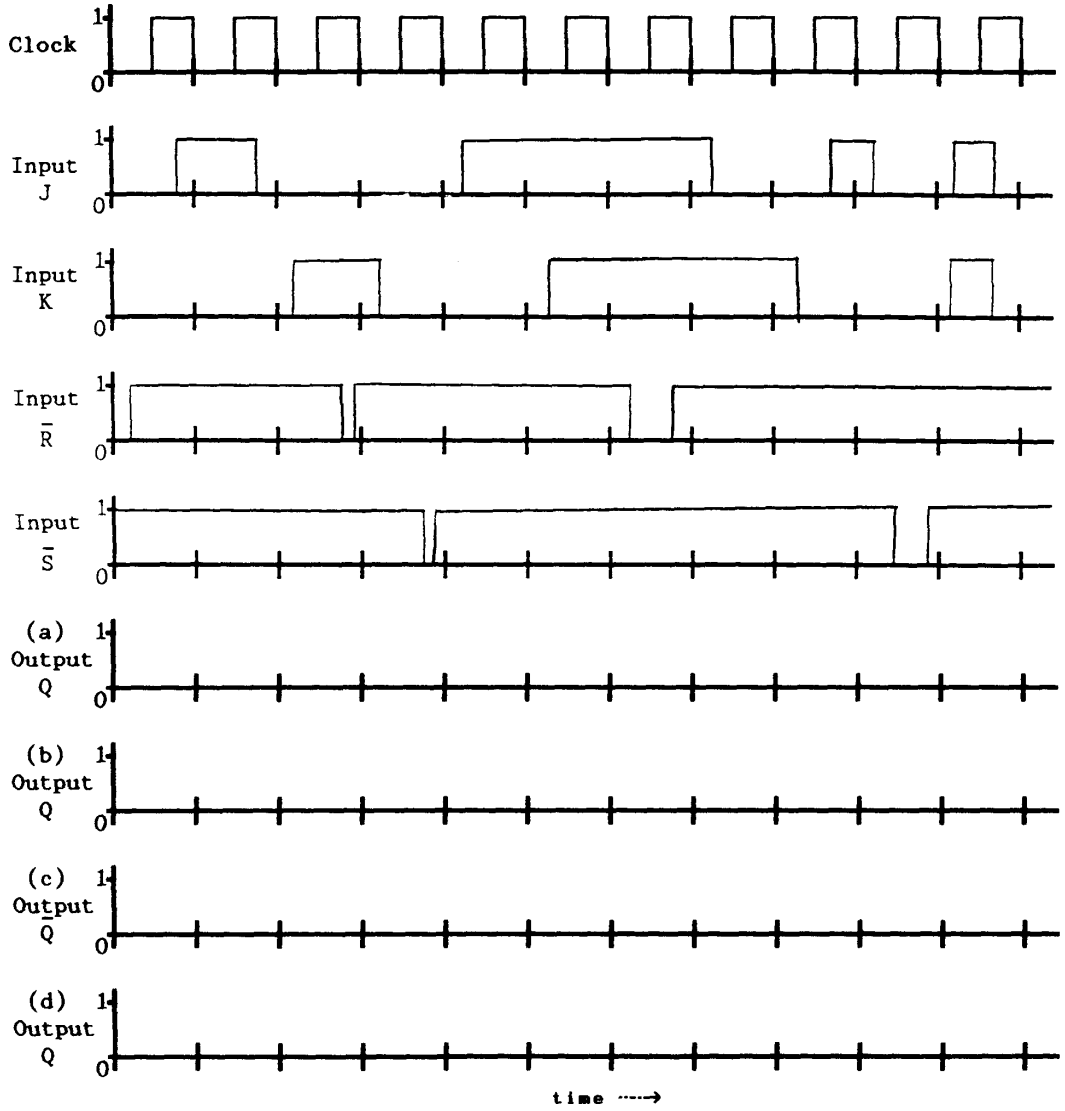
Problem Set

1) Complete the timing diagrams below for the following sequential logic devices. All devices are initially cleared at t=0.
   a) D Latch (1 level triggered)

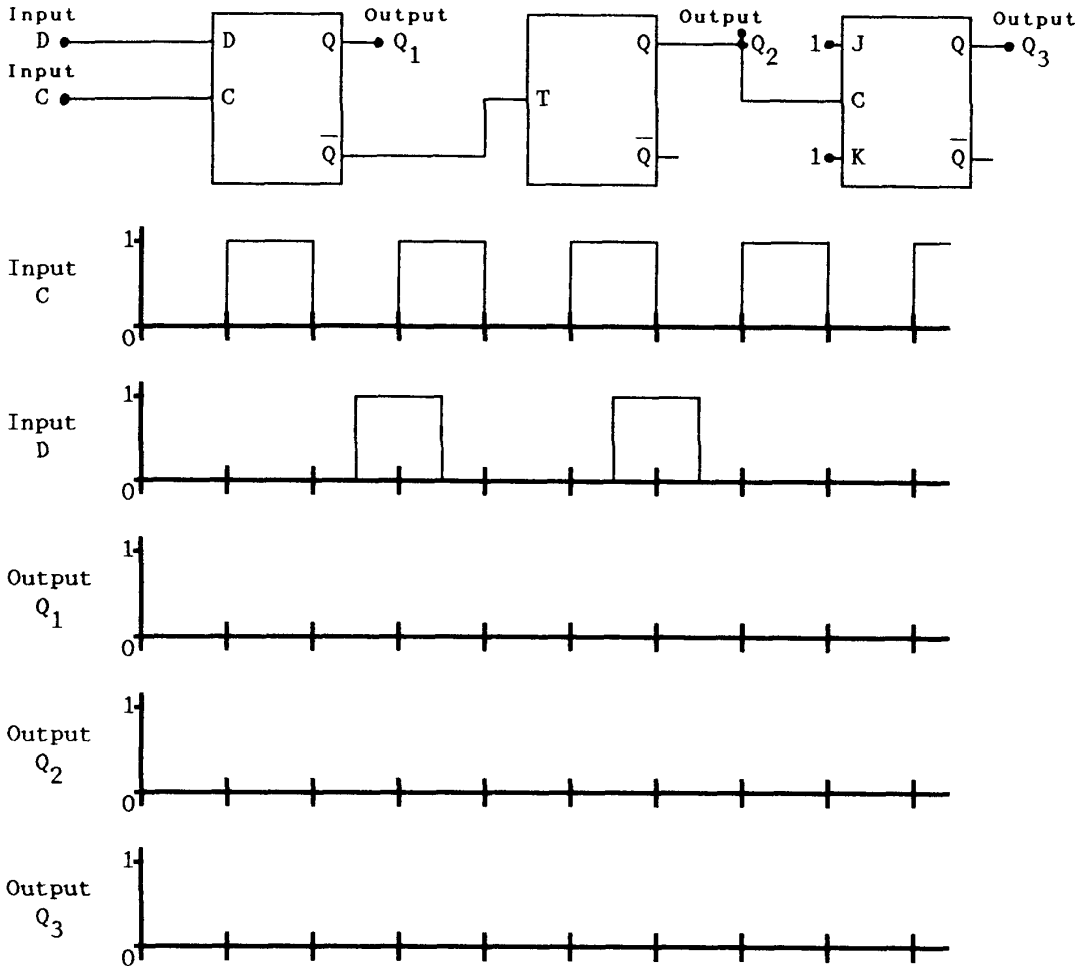
   b) D Flip-Flop (negative edge triggered)



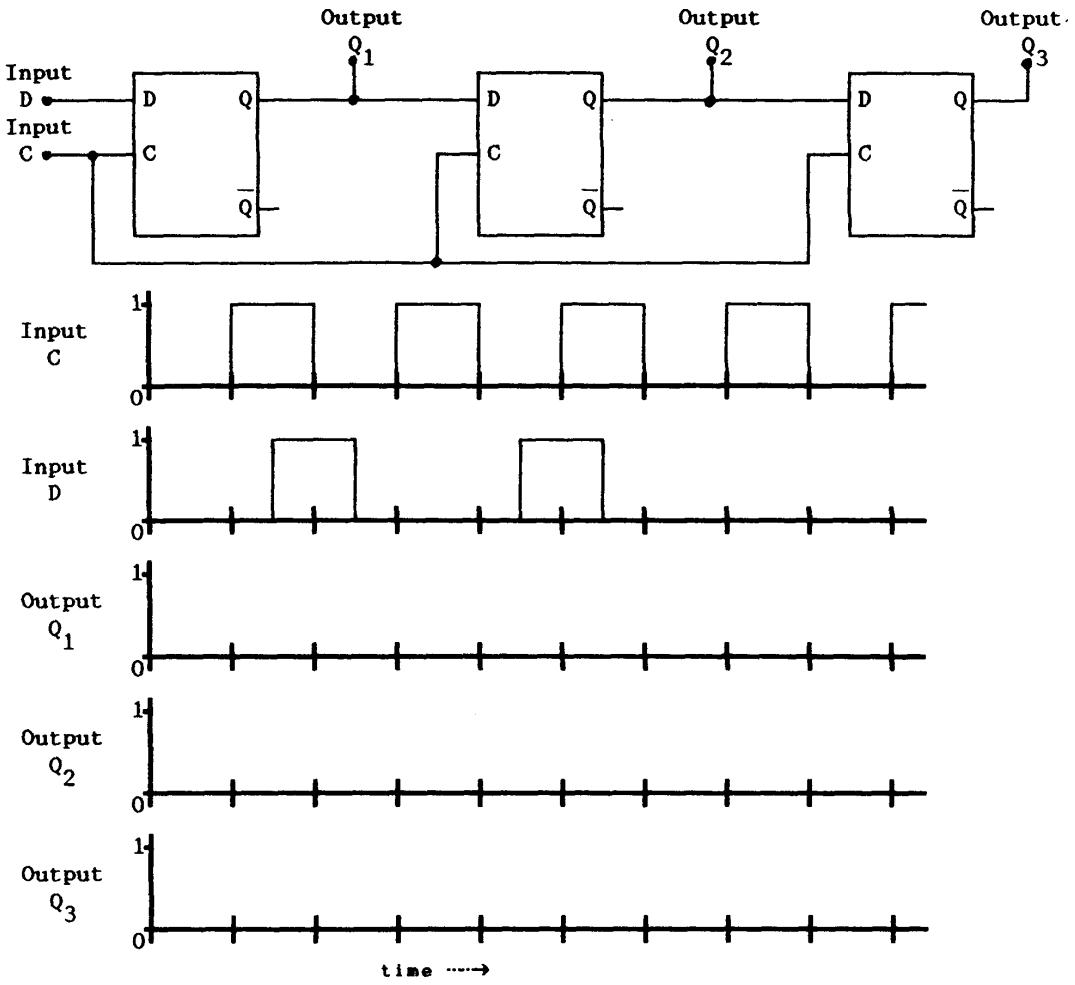
2) Complete the timing diagrams below for the following sequential logic devices. All devices are initially cleared at t=0.
   a) J-K Flip-Flop (negative edge triggered)
   b) J-K Flip-Flop (positive edge triggered)
   c) J-K Flip-Flop for output $\overline{Q}$ (positive edge triggered)
   d) J-K Flip-Flop with Preset and Clear Inputs (negative edge triggered)



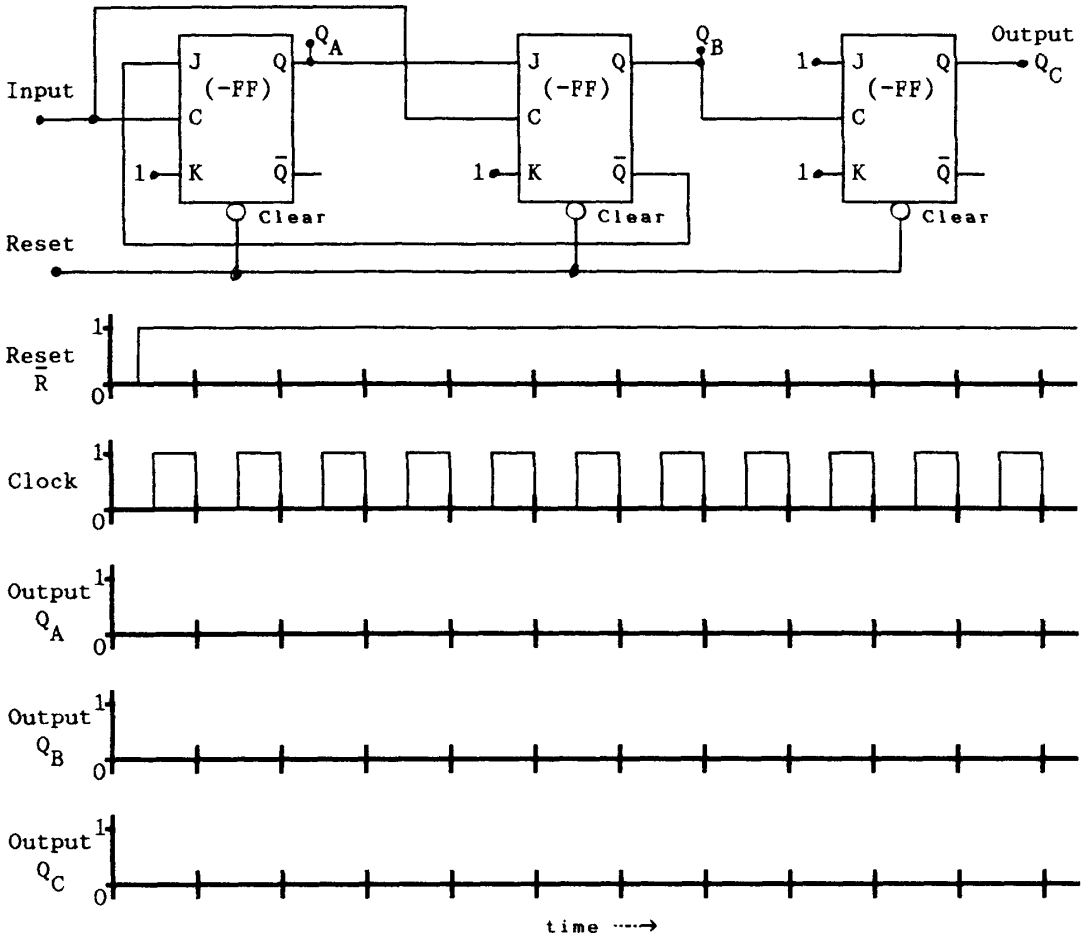
3) Complete the timing diagram below for outputs Q_1, Q_2, and Q_3. The Flip-Flops are negative edge triggered and initially cleared at t=0.



4) Complete the timing diagram below for outputs Q_1, Q_2, and Q_3. The Flip-Flops are negative edge triggered D Flip-Flops and initially cleared at t=0.



5) The J-K Flip-Flops are negative edge triggered and are initially reset at t=0. Draw the waveforms for Q_A, Q_B, and Q_C. Show this is a divide-by-six frequency divider. Is this a synchronous or asynchronous circuit?



6) Use the divide-by-6 frequency divider of Problem 5 to design a divide-by-twelve frequency divider. Draw the circuit diagram for this frequency divider.

Chapter 5. Number Systems And Codes

The decimal number system (base 10) has become the standard number system used by people for counting and mathematical operations. The base ten system is used by most cultures primarily because people have 10 fingers. Each finger is used to represent one of ten possible values that a digit can assume.

Computers do not have 10 fingers. However, they are made up of electronic switches that represent Boolean variables in either a 1 or 0 state. For this reason, the binary (base 2) number system is used to represent the states of Boolean variables. A single binary digit is called a bit, which is in either a 1 or 0 state. A group of eight bits is called a byte. A half byte, which is a set of four bits, is often called a nibble.

Discussed in this chapter are the data formats used by computers to represent numbers and alphanumeric data. Also, binary addition is presented for both signed and unsigned numbers. The base of a number is denoted in this chapter by the subscript 2 for binary, 10 for decimal, and 16 for hexadecimal (base 16). In this text, commas are used with binary numbers to separate groups of four bits, which make the number more readable.

Hexadecimal (Base 16) = 17_{16}

Decimal (Base 10) = 23_{10}

Binary (Base 2) = $0001,0111_2$

} These numbers are equivalent magnitudes in different bases

5.1. Unsigned Binary Numbers

Binary numbers are base two numbers, which can be used to represent various integer quantities. The base two number system operates almost the same way as the decimal system; However, only two symbols (0 and 1) exist for each bit while ten symbols (0 through 9) exist for each digit of the decimal system.

Figure 5.1 compares the decimal and binary number systems. Both number systems are right justified. That is, the least significant bit (LSB) of a binary number is always the right-most bit, just as the least significant digit of a decimal number is the right-most digit. The next significant bit is always a power of two higher than the previous bit. The most significant bit (MSB) represents the highest power of two required for representing a number and is the left-most bit.

5.1.1. Binary to Decimal Conversion

The four bit binary number of Figure 5.1 is easily converted to decimal notation. Simply sum the powers of two for all bits with a 1. Bits with a 0 are not added to this sum. Therefore, the conversion of 1011_2 to decimal is performed by the sum:

$$(1)2^3 + (0)2^2 + (1)2^1 + (1)2^0 = 8 + 2 + 1 = 11_{10}$$

Generally, the minimum data word length for microcomputers is 8 bits or one byte. To convert any binary number to decimal, determine the powers of two corresponding to each

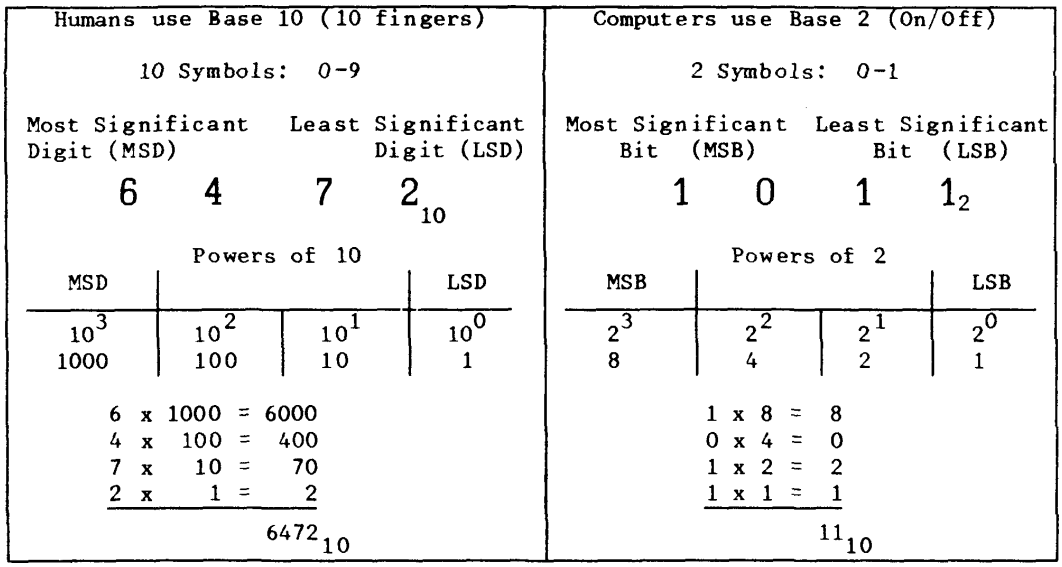
bit with a value of 1, and add up the appropriate magnitudes representing the power of two for each bit. For an eight-bit number represented by the eight bits $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ the least significant bit, b_0, represents the 2^0 or the 1's place and the most significant bit, b_7, represents the 2^7 or the 128's place. Conversion of an eight-bit number is represented by the following equation:

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$

$$= \quad b_7\,(128) + b_6\,(64) + b_5(32) + b_4\,(16) + b_3\,(8) + b_2\,(4) + b_1\,(2) + b_0\,(1)$$

The largest number that can be represented by eight bits would have a 1 for all eight bits b_7 through b_0. This corresponds to $1111,1111_2 = 255_{10}$. The binary number $0000,0000_2$ represent decimal zero. Therefore, eight bits may be used to represent any integer decimal number within the range of 0 to 255 inclusive. Additional bits are required to represent decimal integers greater than 255.

Figure 5.1 Decimal and Binary Number Systems



5.1.2. Decimal to Binary Conversion

To convert a decimal number to a binary number is more tedious than binary to decimal conversion. Tables such as Figure 5.2 are often used to facilitate these conversions. As an alternative to tables, a direct mathematical procedure is shown in Example 5.1. Consider the decimal number 19. Conversion is performed using successive divisions by 2. First 19 is divided by 2 which will generate a quotient of 9 and remainder of 1. Next divide the preceding quotient 9 by 2, which will generate the next quotient of 4 and remainder of 1.

Example 5.1: Convert 19 to Binary

	Q	R	
19 ÷ 2 =	9	1	(LSB Least Significant Bit)
9 ÷ 2 =	4	1	
4 ÷ 2 =	2	0	
2 ÷ 2 =	1	0	
1 ÷ 2 =	0	1	(MSB Most Significant Bit)

Therefore $19_{10} = 10011_2 = 0001,0011_2$

Continue the division by 2 until a quotient of 0 exists. The binary equivalent is determined by examining the remainder column and taking the final remainder as the most significant bit and the first remainder as the least significant bit. Therefore, 19 decimal is represented by the binary number 10011. Binary numbers are often zero filled to eight bits resulting in 0001,0011 to represent decimal 19.

Figure 5.2 Unsigned Decimal to Binary Conversions

Decimal	Binary	Decimal	Binary
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	34	00100010
3	00000011	...	
4	00000100	62	00111110
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	66	01000010
9	00001001	...	
10	00001010	126	01111110
11	00001011	127	01111111
12	00001100	128	10000000
13	00001101	129	10000001
14	00001110	130	10000010
15	00001111	...	
16	00010000	252	11111100
17	00010001	253	11111101
...		254	11111110
31	00011111	255	11111111

5.2. Signed Binary Numbers

Signed binary number representations are used to represent both positive and negative numbers. They also allow signed binary addition and subtraction operations, which may yield negative results.

To utilize the same full adder circuit for both signed and unsigned binary numbers the *2's complement* data format is utilized to represent signed binary numbers. When referring to signed binary numbers, the 2's complement representation will always be used in this text.

An abbreviated table of 8-bit 2's complement numbers is shown in Figure 5.3. For positive numbers within the range of 127_{10} through 0_{10}, the 2's complement representation is identical to the unsigned binary format.

However, for negative numbers a conversion procedure is required. Conversion is performed for 2's complement negative numbers using the following three steps:

Step 1) Determine the unsigned binary number magnitude
Step 2) Complement (invert) the state of each bit
Step 3) Add 1 to the result

Figure 5.3 Some Signed Binary to Decimal Conversions

+	Two's complement code	−	Two's complement code
+ 127	01111111	− 128	10000000
+ 126	01111110	− 127	10000001
+ 125	01111101	− 126	10000010
. . .		− 125	10000011
		. . .	
+ 65	01000001	− 65	10111111
+ 64	01000000	− 64	11000000
+ 63	00111111	− 63	11000001
. . .		. . .	
+ 33	00100001	− 33	11011111
+ 32	00100000	− 32	11100000
+ 31	00011111	− 31	11100001
. . .		. . .	
+ 17	00010001	− 17	11101111
+ 16	00010000	− 16	11110000
+ 15	00001111	− 15	11110001
+ 14	00001110	− 14	11110010
+ 13	00001101	− 13	11110011
+ 12	00001100	− 12	11110100
+ 11	00001011	− 11	11110101
+ 10	00001010	− 10	11110110
+ 9	00001001	− 9	11110111
+ 8	00001000	− 8	11111000
+ 7	00000111	− 7	11111001
+ 6	00000110	− 6	11111010
+ 5	00000101	− 5	11111011
+ 4	00000100	− 4	11111100
+ 3	00000011	− 3	11111101
+ 2	00000010	− 2	11111110
+ 1	00000001	− 1	11111111
+ 0	00000000		

Example 5.2 illustrates this 2's complement conversion procedure using four examples. This procedure is only used for converting negative numbers to 2's complement form. Again, positive numbers will have the same form for both unsigned binary and 2's complement representations.

This same three-step procedure can also be used to change the sign of the number. This is useful when converting a negative 2's complement number to decimal and for the subtraction operation. Subtraction can be performed on 2's complement numbers by first performing the three-step 2's complement procedure described to change the sign of the subtrahend; Then, add the two numbers together. By using this procedure to perform subtraction, no additional subtraction hardware is required to perform the subtraction operation on two 2's complement numbers.

The range of 8-bit 2's complement numbers is +127 to -128 inclusive, as shown in Figure 5.3. The most significant bit is the sign bit. If $b_7 = 0$, the number is positive, and if $b_7 = 1$ the number is negative. Note that only one value exists for zero in 2's complement numbers, and zero is considered a positive number. Therefore, the highest value of positive numbers is 127 instead of 128 as one might expect.

Example 5.2: The 2's Complement Representation

Convert the following decimal Numbers:

	-5_{10}	-65_{10}	-126_{10}	$+63_{10}$
Step 1) Determine the unsigned binary number which represents the magnitude;	0000,0101	0100,0001	0111,1110	Positive Number
Step 2) Complement (invert) the state of each bit;	1111,1010	1011,1110	1000,0001	↓
Step 3) Add 1 to the result.	1111,1011	1011,1111	1000,0010	0011,1111

5.3. Binary Addition

Binary addition can be easily performed by hand just as decimal addition. First, line up the bits in columns as shown in Example 5.3. Then add up the bits of each column. When the sum for each bit exceeds 1, a carry is generated into the next significant bit. The carry out of the most significant bit represents the carry flag. If this carry flag is in the 1 state and the numbers are unsigned binary, then the sum exceeds what can be shown with 8 bits. When a carry out exists from bit 6 with no carry out from bit 7, or vice versa, an improper sign change has occurred for the sum in 2's complement form and an overflow condition exists. When this overflow condition is true the resulting 2's complement sum is invalid.

Addition is performed using the same method for both unsigned and signed 2's complement numbers. The only differences are the conditions dictating the validity of the sums. Therefore, it is up to the computer programmer to keep track of whether the data is in signed or unsigned formats and to check the appropriate carry or overflow conditions to determine if the sums are valid.

Binary addition of both unsigned and 2's complement signed numbers can be performed using the full adder hardware of Figure 5.4. This figure is similar to Figure 2.18 with the exception that eight full adder circuits are cascaded together so that addition can be performed on 8-bit data and both a Carry flag (C) and Overflow flag (V) are available for examination. The C and V flags are often referred to as the carry and overflow condition codes in computer literature.

The carry flag is actually the carry out from the full adder of the most significant bit. The carry flag is considered only when performing unsigned binary addition. When the carry flag is a '0' it specifies that the resulting unsigned binary sum is within the range of valid binary numbers 0 through 255. A carry flag of one specifies that the resulting sum is greater than 255 and the result is not valid. The overflow flag is meaningless when performing unsigned binary addition.

Addition of signed 2's complement numbers is also performed using the circuitry of Figure 5.4. When performing 2's complement arithmetic the carry flag is meaningless and only the overflow flag specifies whether the resulting sum is within the valid range from -128 through +127. The overflow flag is generated by exclusive-OR'ing the carry out from bit 7 with the carry out of bit 6. If one, but not both, of these carry outs are a '1' it means that an improper sign change has occurred and the resulting sum is not valid. Therefore, if the overflow flag is 0, the 2's complement sum is valid; An overflow flag of 1 signifies that the sum is out of range (-128 through +127).

Example 5.3: Binary Addition

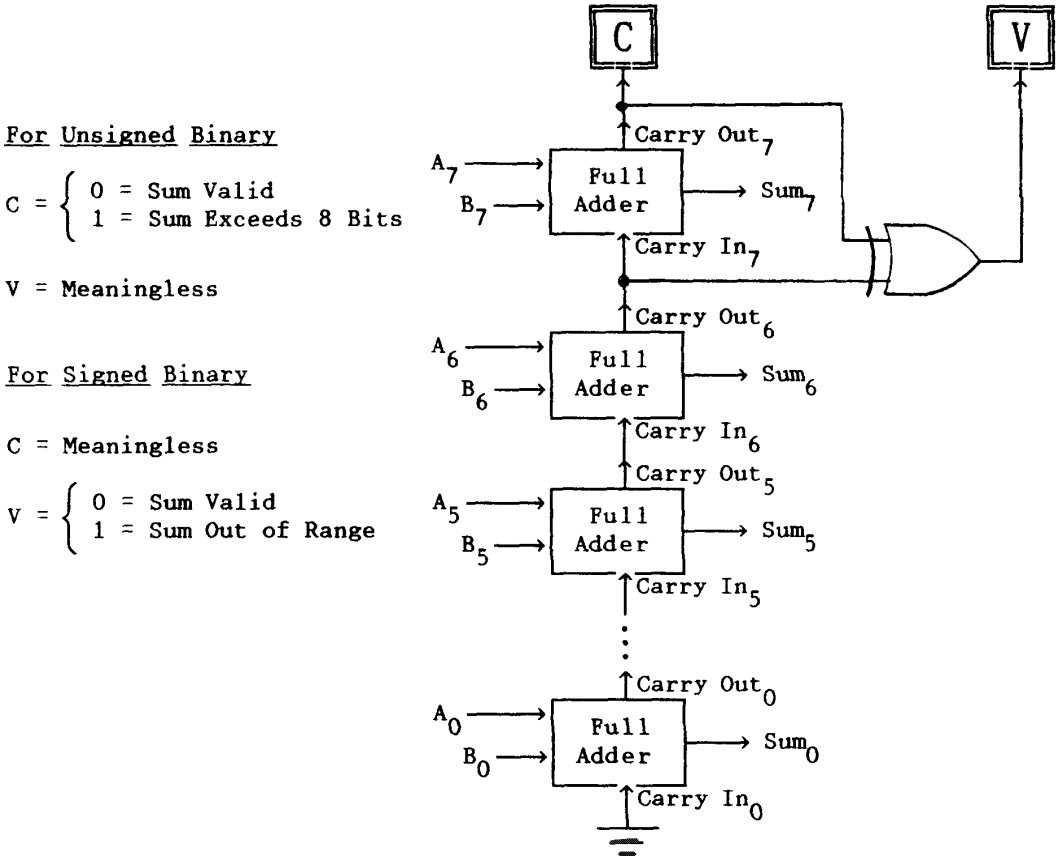
		Unsigned	Signed
Carry			
A =	0001,0010	18	+18
B =	0000,0101	5	+5
Sum =	0001,0111	23	+23
		Valid	Valid
		C=0	V=0

		Unsigned	Signed
Carry	11		
A =	0100,0011	67	+67
B =	1000,0011	131	-125
Sum =	1100,0110	198	-58
		Valid	Valid
		C=0	V=0

		Unsigned	Signed
Carry	1111 1 1		
A =	0111,1101	125	+125
B =	0010,0101	37	+37
Sum =	1010,0010	162	+162
		Valid	>+127
		C=0	V=1

		Unsigned	Signed
Carry	11111 111		
A =	0000,0111	7	+7
B =	1111,1101	253	-3
Sum =	0000,0100	260	+4
		>255	Valid
		C=1	V=0

Figure 5.4 8-Bit Adder Circuit With Overflow and Carry Flags



5.4. Binary Number Magnitude

The number of bits used for the data word restricts the magnitude of decimal numbers that can be represented by binary numbers. Binary numbers with eight bits can represent unsigned numbers in the range from 255 through 0, or signed numbers in the range from +127 through -128. Numbers outside of these ranges cannot be represented unless additional bits are used to increase the data word size.

Binary numbers can represent a maximum of 2^n decimal numbers, where n is the number of bits of the data word. For unsigned numbers the range begins at zero. For signed binary numbers the center of the range is zero.

Consider the case when the data word size is increased from one byte to two bytes (16 bits). A two-byte binary number can be used to represent 65,536 different decimal numbers. For unsigned binary numbers, 16 bits would represent decimal numbers in the range of 65,535 to 0. The signed 2's complement range is from +32,767 to -32,768. Once again, zero is considered a positive number. Therefore, the range of negative numbers appears to be one more than positive numbers.

Computers have a standard word size in which all data is represented as some multiple of eight bits (i.e. 8, 16, 32, 64).

5.5. Binary Coded Decimal Representation

People prefer to work with decimal numbers for both data entry and display applications. A direct binary representation of decimal numbers is preferable to the conversion methods of Section 5.1. A binary data format used for direct representation of decimal digits is called *binary coded decimal*, usually abbreviated *BCD*. The standard format used to represent BCD numbers is shown in Figure 5.5. This BCD format uses a group of four bits to represent one decimal digit. The four bits are weighted such that each bit corresponds to the values 2^3, 2^2, 2^1, 2^0 or 8, 4, 2, 1. Only binary numbers 0000 through 1001 are valid BCD codes. Any four bits with a value greater than 1001 are considered invalid. Therefore, only ten out of the sixteen possible combinations of four bits are considered valid BCD codes. Since each decimal digit is encoded into four bits, two decimal digits can be represented in one byte. This is called often called packed BCD and will be the standard format used to represent BCD numbers in this text.

Although BCD formats are good for human interfaces they are very inefficient for data storage and processing. A two-byte BCD format can only represent decimal numbers in the range 9999 to 0. A two byte unsigned binary format will represent numbers in the range 65,535 to 0. The adder circuit of Figure 5.4 will not give direct results when adding BCD formatted numbers. Additional logic circuits are required to perform BCD addition.

The *Grey Code* of Figure 5.5 is an example of a non-weighted binary format used to represent decimal numbers. Grey code is often used for sensors with digital outputs, such as shaft encoders. A binary code exists for each decimal digit; however, the bits do not represent powers of two. A direct conversion algorithm does not exist for Grey code as with the standard BCD data. Therefore, a look-up table, such as the one in Figure 5.5, must be used.

Figure 5.5 Binary Coded Decimal Codes

Decimal	Standard BCD	Grey Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1000

5.6. Floating Point Representations

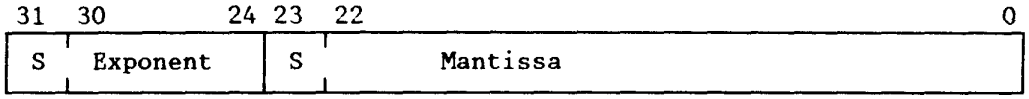
Signed integer numbers are represented using the 2's complement format of Section 5.2. Integer numbers are whole numbers. An alternate approach is required to represent real numbers.

The floating-point representation is used to represent real numbers in much the same way as scientific notation. Consider the number -0.0004772, which can be represented in scientific notation as -0.4772×10^{-3}. The mantissa is normalized to be a number between ±0.9999 and ±0.1000. The exponent -3 is a signed integer quantity representing the power of ten that the mantissa is multiplied. The actual bit format used to represent floating point numbers varies dependent on the data word size of the computer.

Figure 5.6 shows a typical 32-bit format used to represent floating point numbers. For example both the mantissa and exponent can be represented using 2's complement numbers. In both cases a sign bit of 0 represents a positive number and a sign bit of 1 represents a negative number. The number of bits in the exponent determines the range of powers of ten. An 8 bit exponent can represent powers of ten from 10^{+127} to 10^{-128}. A 24-bit mantissa can represent signed decimal numbers of six significant digits for the range +999,999 to !999,999.

If more significant digits are required for an application, allocating more bits to the mantissa can increase the mantissa size. Most computer programming languages allow a higher precision (more significant digits) floating point number representation that is often called double precision floating point numbers.

Figure 5.6 Floating Point Number Representation



5.7. Hexadecimal Numbers

Representing eight bits of data as a string of 1's and 0's can be tedious.

The hexadecimal number system is used to simplify data representation by encoding 4 bits as one symbol. Hexadecimal numbers comprise the base 16 number system. The hexadecimal number system has sixteen different symbols, which represent the value of each hexadecimal digit. As shown in the conversion table of Figure 5.7a, 0 through 9 are used to represent the first ten hexadecimal symbols, and letters A through F are used to represent the last six symbols. Each hexadecimal symbol represents one of the sixteen possible combinations of four bits. Therefore, eight bits of data is more easily represented as two hexadecimal digits.

5.7.1. Binary to Hexadecimal Conversion

To convert a binary number to hexadecimal is quite easy. Starting from the least significant bit (b_0), group the binary bits into groups of four. Use the table of Figure 5.7a to determine the hexadecimal symbol that represents each group of four bits.

For example:

$1000,1100,0111,1010_2 =$

$8 \quad C \quad 7 \quad A_{16}$

Figure 5.7 Hexadecimal Numbers

a)

Base 16	Base 2	Base 10
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

b)

Base 16 can represent 4 bits

16 Symbols: 0,1,2,3,4,5,6,7, 8,9,A,B,C,D,E,F

$A\ 5\ 2\ F_{16}$

Most Significant Digit (MSD) — Least Significant Digit (LSD)

Powers of 16

MSD			LSD
16^3	16^2	16^1	16^0
4096	256	16	1

(A=) 10 x 4096 = 40,960
(5=) 5 x 256 = 1,280
(2=) 2 x 16 = 32
(F=) 15 x 1 = 15

$42,287_{10}$

5.7.2. Hexadecimal to Binary Conversion

Converting from hexadecimal to binary is also quite simple. Use the table of Figure 5.7a to determine the group of four bits, which represents each digit of the hexadecimal number.

For example:

A 5 2 F_{16} =
$1010,0101,0010,1111_2$

5.7.3. Hexadecimal to Decimal Conversion

An example of hexadecimal to decimal conversion is shown in Figure 5.7b, for hexadecimal number A52F. To convert any hexadecimal number to decimal, determine the powers of sixteen representing each digit and multiply the number in each digit by the power of sixteen associated with that digit. Then add up the products of each digit to determine the equivalent decimal number.

Conversion of a four digit hexadecimal number, $h_3\ h_2\ h_1\ h_0$, to decimal is represented by the following equation:

$h_3 16^3 + h_2 16^2 + h_1 16^1 + h_0 16^0 = h_3\ (4096) + h_2\ (256) + h_1\ (16) + h_0\ (1)$

5.7.4. Decimal to Hexadecimal Conversion

To convert from a decimal number to a hexadecimal number is more difficult.

One method is similar to the decimal to binary conversion procedure discussed in Example 2.1. The procedure for converting from a decimal number to a hexadecimal number is shown in Example 5.5. Consider the decimal number 42,287. Conversion is performed

using successive divisions by 16. First divide 42,287 by 16 which will generate a quotient of 2642 and remainder of 15.

Next divide the preceding quotient 2642 by 16, which will generate the next quotient of 165 and remainder of 2. Continue the division by 16 until a quotient of 0 exists. The hexadecimal equivalent is determined by examining the remainder column and taking the final remainder as the most significant hexadecimal digit and the first remainder as the least significant digit.

Remainders in the range of 10 to 15 would of have to be converted to their hexadecimal symbols described in the table of Figure 5.7a. Therefore, 42,287 decimal is equivalent to the hexadecimal number A52F.

Example 5.5: Convert 42,287 Decimal to Hexadecimal

```
                 Quotient     Remainder
42,287 ÷ 16 = 2642          15 = F  ← LSD = Least Significant Digit
 2,642 ÷ 16 =  165           2 = 2
   165 ÷ 16 =   10           5 = 5
    10 ÷ 16 =    0          10 = A  ← MSD = Most Significan Digit

          42,287₁₀    =     A52F₁₆
```

5.7.5. Hexadecimal Addition

Hexadecimal addition is similar to decimal addition except digits are incremented up to F (15_{10}) before generating a carry into the next significant digit. This is demonstrated with the following examples:

$$\begin{array}{r} 5A_{16} \\ +\ 36_{16} \\ \hline 90_{16} \end{array} \qquad \begin{array}{r} 52_{16} \\ +\ AC_{16} \\ \hline FE_{16} \end{array} \qquad \begin{array}{r} 29_{16} \\ +\ 28_{16} \\ \hline 51_{16} \end{array}$$

5.8. Alphanumeric Data Representation

People use written language to communicate among themselves and to give instructions to a computer in the form user keyboard input. Alphanumeric data is represented by a binary code for each letter, number, and symbol commonly associated with a typewriter keyboard. The ASCII (American Standard Code for Information Interchange) Code is the most commonly used representation for alphanumeric data. Figure 5.8 presents a table used to convert from either hexadecimal or binary codes to the ASCII characters represented by these codes. All upper and lower case letters, numbers, and symbols used in the English language are in column 2 through 7. Special computer control characters are found in columns 0 and 1. A definition of each of these control characters is located below the conversion table.

An ASCII character is represented by a byte, with bit 7 (MSB) being the parity bit and bits 6 through 0 determined by the conversion table. The parity bit is used for error detection when transmitting and receiving data. Both the transmitter and receiver must be setup to transfer data with the same parity. The parity bit value is determined by the type of parity

selected and the state of bits 6 through 0. Four types of parity exist as described in the table below. However, in most applications the parity bit is simply set to '0'.

0 Parity	Bit 7 set to 0 (Default)
1 Parity	Bit 7 set to 1
Even Parity	Bit 7 set to generate an even number of 1's for the byte
Odd Parity	Bit 7 set to generate an odd number of 1's for the byte

5.8.1. Binary String to ASCII Character Conversion

Given a string of binary bits or hexadecimal numbers, conversion to the ASCII characters is performed by separating the string into individual bytes. Again, it is assumed that the parity bit is equal to 0.

To convert bits 6 through 0 to ASCII, use the ASCII conversion table of Figure 5.8. The least significant nibble, bits 3 through 0, represents the rows of the conversion table. Entries are listed in both binary bits and hexadecimal digits. Bits 6 through 4 represent the columns of the table. Once the specific column and row are located, the ASCII character defined by the byte is found at the specified row and column. Continue this conversion process for each byte of the string. This procedure is illustrated in the example below.

As an example, convert the following hexadecimal representation of a binary string with '0' parity to ASCII characters.

Hexadecimal Representation = 576861743F

57	68	61	74	3F
0101,0111	0110,1000	0110,0001	0111,0100	0011,1111

Convert using ASCII Conversion Table gives you the following answer:

W	h	a	t	?

= What?

5.8.2. ASCII Character to Binary String Conversion

The conversion of a string of characters to a string of bits or hexadecimal digits is performed using the reverse process of the above. Find the ASCII character in the ASCII Conversion table. Determine the byte representing this ASCII character by first determining the row of the character in the table. This specifies the least significant nibble, bits 3 through 0. Then determine the column of the character, which specifies bits 6 through 4. Bit 7 is the parity bit and assumed to be 0. Once all eight bits of the ASCII character byte are determined, they can be converted to hexadecimal, if desired. This procedure is illustrated below.

ASCII Character String = What?

	W	h	a	t	?
Conversion	101,0111	110,1000	110,0001	111,0100	011,1111
Parity Added	0101,0111	0110,1000	0110,0001	0111,0100	0011,1111
Hexadecimal	57	68	61	74	3F

Figure 5.8 ASCII Conversion Table

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BINARY	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	‘	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

Control Characters:

NUL	Null	VT	Vertical	SYN	Svnchronous Idle
SOH	Start of Heading	FF	Form Feed	ETB	End Transmission Block
STX	Start of Text	CR	Carriage Return	CAN	Cancel
ETX	End of Text	SO	Shift Out	EM	End of Medium
EOT	End of Transmission	SI	Shift In	SUB	Substitute
ENQ	Enquiry	DLE	Data Link	ESC	Escape
ACK	Acknowledge	DC1	Device Control	FS	File Separator
BEL	Bell	DC2	Device Control	GS	Group Separator
BS	Backspace	DC3	Device Control	RS	Record Separator
HT	Horizontal	DC4	Device Control	US	Unit Separator
LF	Line Feed	NAK	Negative	DEL	Delete

Problem Set

1. Perform the following decimal to binary conversions. Verify your answers by performing binary to decimal conversions.
   a) 25 b) 31 c) 173 d) 250
   e) 320 f) -19 g) -102 h) -185

2. Perform the following binary additions on one byte numbers. Determine the state of the carry flag [C] and the overflow flag [V] after the addition operation.

```
a) 0010,0110 b) 0110,0010  c) 1010,0111  d) 1010,1010
  +0001,1011   +0101,0001    +0110,0011  +1100,0011
```

3. Determine the decimal equivalents for the numbers of problem 2 for both binary signed and unsigned number representations. Verify that the resulting sum is correct by also determining the decimal equivalents.

4. When performing binary addition what is the significance of the overflow flag and carry flags?

5. Convert the following decimal numbers to both 8421 BCD and Grey Code BCD formats. Verify your answers by performing the reverse conversion. Assume four bits represent one decimal digit.

   a) 25 b)83 c)94 d)122

6. Convert the following hexadecimal numbers to binary. Verify your answers by converting the binary result back to hexadecimal.

   a) 87 b) 23 c) CF d)73B

7. Convert the hexadecimal numbers of Problem 8 to decimal. Verify your answers by converting the decimal result back to binary.

8. Write the hexadecimal string for the following ASCII character strings. Assume the parity bit is ‘0’.

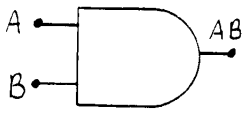
   a) What If?
   b) PO Box 135
   c) Houghton, MI
   d) 49931

9. Convert the following hexadecimal string to ASCII characters.
   45 45 33 30 36 20 49 73 20 46 75 6E 21

Problem Set Solutions for Select Problems

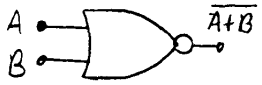
Chapter 2 Problem Solutions

2a)



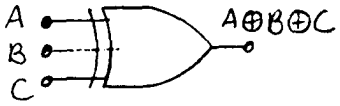
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

2c)



A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

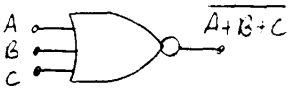
2e)



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

3a)

3c)



4a) $0A = 0$

A	0	0A
0	0	0
1	0	0

↑ Answer

4c) $AA = A$

A	A	AA
0	0	0
1	1	1

↑ ↑ Verifies

4e) $(A+B)+C = A+(B+C)$

A	B	C	A+B	(A+B)+C	B+C	A+(B+C)
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

↑ ↑ Verifies

5a) $(A+B)(A+C)$

$= AA + AC + AB + BC$ Expand Out (Distrib OR)

$= A + AC + AB + BC$ Idempotent AND Law

$= 1A + AC + AB + BC$ Identity AND Law

$= A(1+C) + AB + BC$ Distributive OR Law

$= A(1) + AB + BC$ Null OR Law

$= A(1+B) + BC$ Distributive OR Law

$= A(1) + BC$ Null OR Law

$= A + BC$ Identity AND Law

Therefore $A + BC = (A+B)(A+C)$

Chapter 2 Problem Solutions

5c) $(A + \overline{B+C})\,\overline{A}$

$= A\overline{A} + \overline{B+C}\;\overline{A}$ — Distributive OR Law

$= 0 + \overline{B+C}\;\overline{A}$ — Inverse AND Law

$= \overline{A}\;\overline{B+C}$ — Commutative AND Law

$= \overline{A}\;\overline{B}\;\overline{C}$ — De Morgan's OR Law

$= \overline{A+B+C}$ — De Morgan's OR Law

5e) $\overline{A\;\overline{B(B+C)}}$

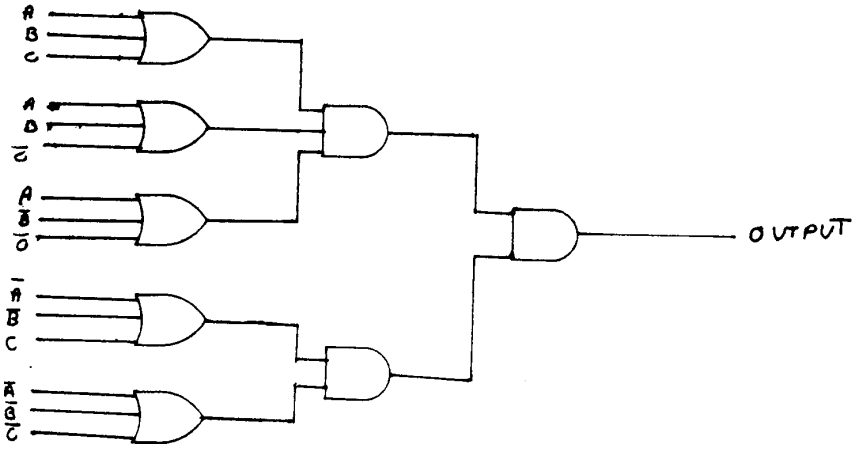
$= \overline{A} + \overline{\overline{B(B+C)}}$ — DeMorgan's AND Law

$= \overline{A} + B(B+C)$ — Double Inversion

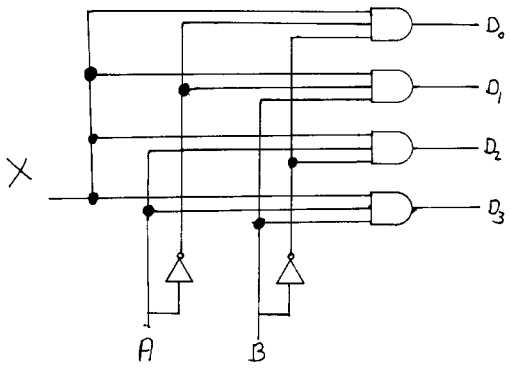
$= \overline{A} + B$ — Absorption AND Law

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

7 POS)



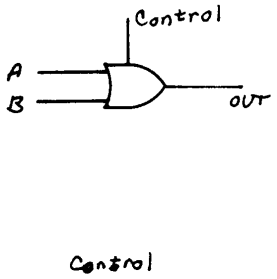
11d)



X	A	B	D_0	D_1	D_2	D_3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Chapter 3 Problem Solutions

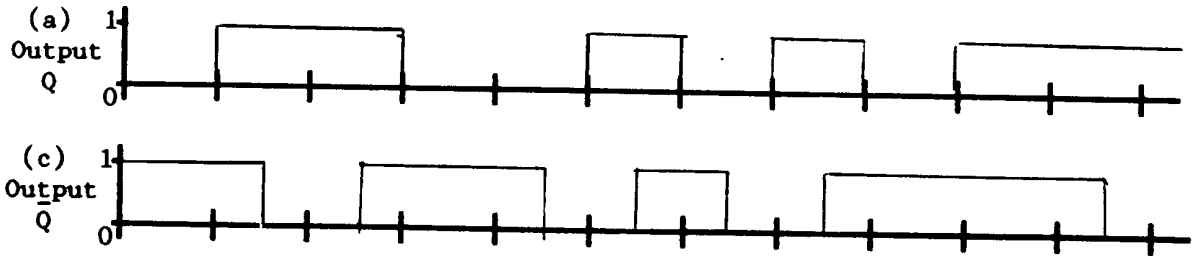
9d)



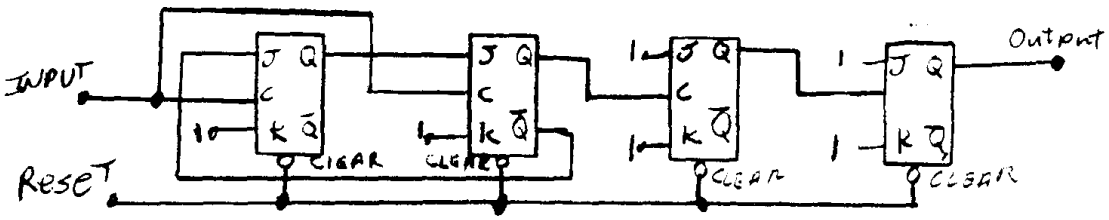
Control	A	B	OUT
0	0	0	Disc
0	0	1	"
0	1	0	"
0	1	1	"
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Chapter 4 Problem Solutions

2a&c)



6)

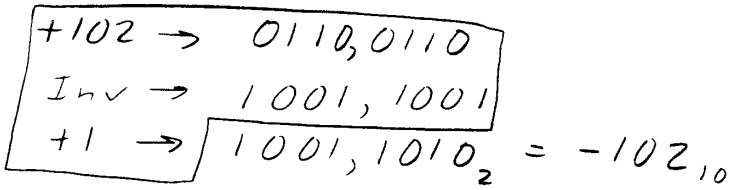


Since the clocks are not all connected together in this circuit it is an asynchronous circuit

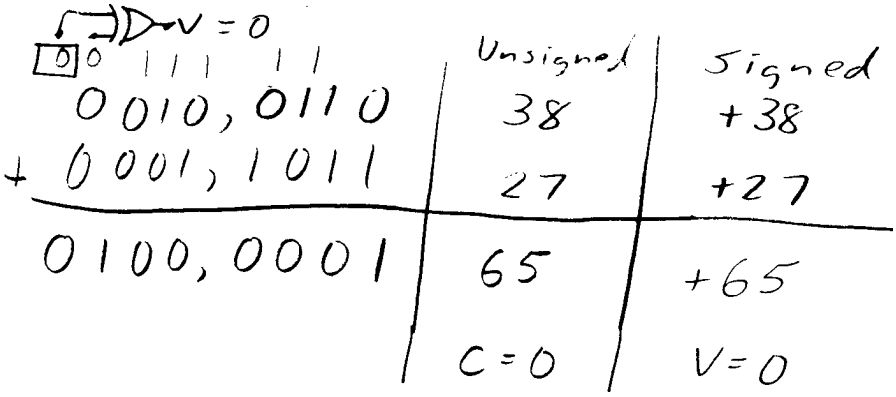
Chapter 5 Problem Solutions

1a) 25=0001,1001 1c) 173=1010,11011 e) 320=1,0100,0000 ←Exceeds 8 bits

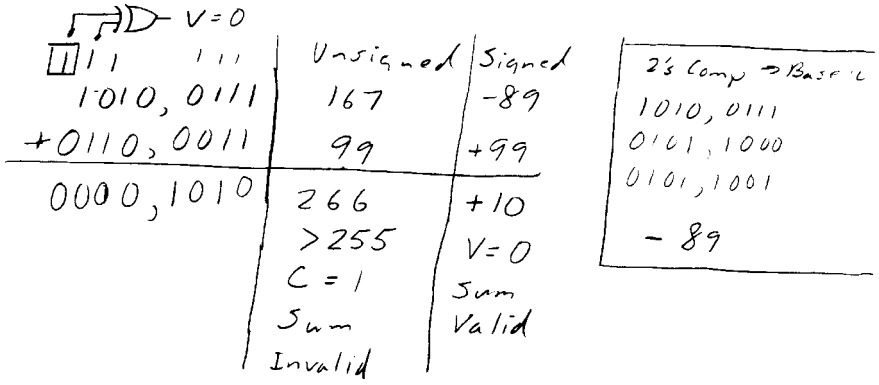
9)



2a)



2c)



6 a) 87_{16} =1000,0111$_2$ 10c) CF_{16}=1100,1111$_2$

7 a) 87_{16} =135_{10} b) 23_{16} =35_{10} c) CF_{16}=207_{10}

8 a) What If? = 57 68 61 74 20 49 66 3F