

Fall 2009

Engineering 1040

Mechanisms and Electric Circuits

Digital Logic Module

by

Eric W. Gill, Ph.D., P.Eng

This module deals with the Digital Logic portion of the Engineering 1040 course. We will consider only elementary concepts in this vast area of study which touches essentially every aspect of modern technological advancement. As you progress through your program of study, no matter what facet of modern engineering practice you eventually pursue, you will in some fashion encounter and use products which have appeared as a result of advances in digital electronics. In fact the electronic circuitry associated with everyday 'devices' that we take for granted, whether it be in the computer, the car or the coffee pot, will most certainly involve some aspect of digital logic.

The intent of this module is to give an exposure to a few rudimentary ideas associated with digital logic including

- **digital** versus **analog** signals
- **binary** and **decimal numbers**
- **logic variables, gates, and truth tables**
- **combinational logic** including **Boolean algebra** and **network minimization**
- **sequential logic**

Unit D1: Basic Concepts and Combinational Logic

D1.1 Analog and Digital Signals

In the Electric Circuits Module studied earlier we considered voltages and current that, for the most part, had constant values (i.e. we considered direct current circuits). We did, however, early in the module consider quantities that changed over time. For example, we have seen voltage versus time and current versus graphs that varied linearly with time as seen in Figure D1.1 (a) and (b). A more common illustration might be that of Figure D1.1 (c) showing the *alternating voltage* (sinusoidal) that is used in everyday household circuits. In each of those instances, the signals have *continuous* values over time and are

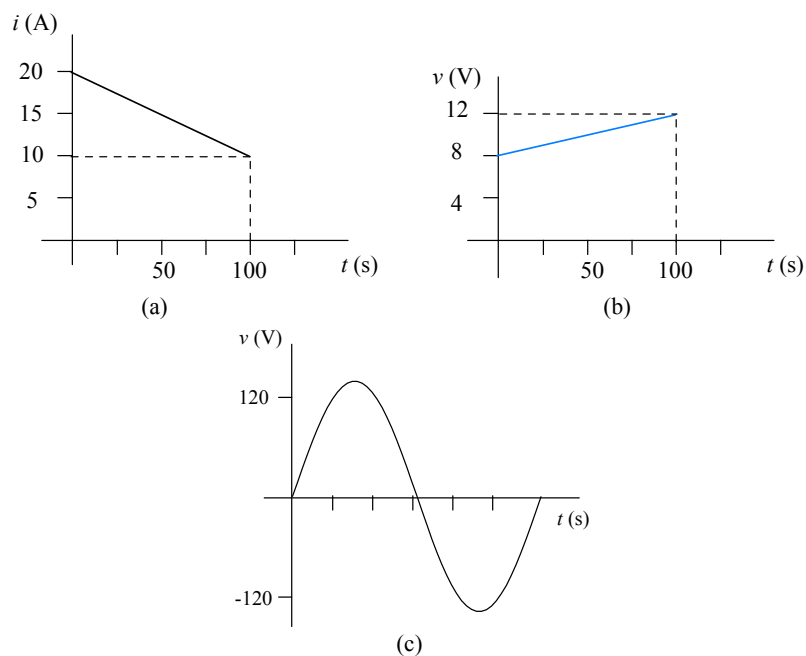


Figure D1.1 Examples of continuous or analog signals.

referred to as **analog** signals. The time varying feature (variable) of the signal may, in fact, be a representation of some other time varying quantity, i.e. *analogous to* another time varying signal. For example, as the resistance of a circuit element changes due to temperature changes, this change may be measured via the change in a voltage across the

resistor. Thus, with the proper scale calibration, the continuous voltage is representative of or *analogous to* the resistance change.

In some cases, discrete pieces of an analog signal may be used to convey useful information. In fact, for some circuits, signals such as those in Figure D1.1 (c) may activate an electrical device only after the signal reaches a predefined level. Likewise, once the signal falls below another predefined level, the device may deactivate. Let's

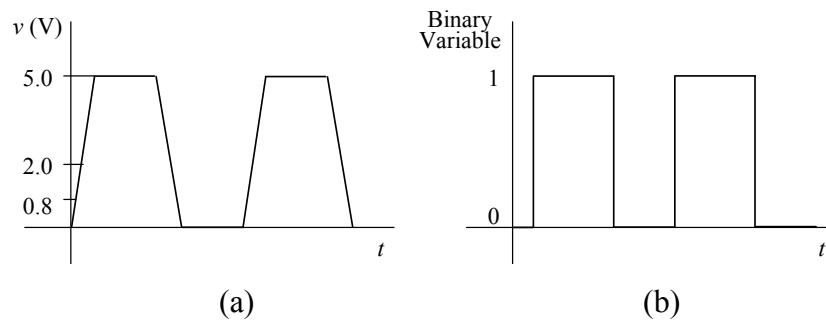


Figure D1.2 An analog signal (a) and its digital counterpart (b).

consider the example in Figure D1.2. Figure D1.2 (a) shows an analog signal that varies continuously from a minimum of 0 volts to a maximum of 5.0 V. Suppose that this signal is used to activate a piece of circuitry, the precise nature of which for our purposes is not important (it might consist, for example, of some combination of devices such as transistors, which will be dealt with in other courses). Let's suppose the circuit does not activate until the signal level reaches 2.0 V. However, when the signal level reaches 2.0 V and as long as the signal is above that level the circuit remains in the ON or HIGH state. For certain types of circuits, the maximum value of the high voltage should be about 5.0 V. To this ON state we assign a value of 1, (or TRUE). Here, the designations of 'ON', 'HIGH', 'TRUE', or '1' are all taken to mean the same thing – namely, that the circuit **is** activated. Next, suppose that when the signal level drops below 0.8 V, the circuit that it is driving deactivates. To this deactivated level, or OFF or LOW state, we assign a value of 0 (or FALSE). Again, the designations 'OFF', 'LOW', 'FALSE' or '0' are all taken to mean that the circuit **is not** activated. Strictly speaking, in this example, we do not assign a meaning to the signal levels lying between 0.8 and 2.0 V (this range, being sometimes referred to as the *logic swing*). Systems which use discrete

(discontinuous) values such as those being discussed here are referred to in general as **digital systems** (thus, the reason for referring to the signal in Figure D1.2 (b) as a **digital signal**). As is discussed in more detail in the following section the term **binary** is used to describe a special digital system which can take only **two** possible states of operation (*on* or *off*) and the ‘Binary Variable’ label used in Figure D1.2 (b) is an indication of this fact.

D1.2 Base-2 (Binary) and Base-10 (Decimal) Number Systems

As intimated above, here we are interested in systems which operate in only two possible states – i.e. **binary systems**. These states of ON/OFF, HIGH/LOW, TRUE/FALSE may be expressed in the **base-2** (or **binary**) **numeral system** which represents numeric values using two symbols, usually 0 and 1. Thus, 1 can be used to indicate ON or HIGH or TRUE while 0 represents OFF or LOW or FALSE. Furthermore, because basic ON/OFF circuitry, such as used in modern computers, may be combined not only to replicate logical decision making but to perform arithmetic operations, it is useful to explore how the familiar **base-10** numbers and operations on them may be implemented using the **base-2** numerals (or **binary digits**) 0 and 1. The term **binary digits** is typically shortened to **bits**.

Recall that the **base-10** or **decimal** system represents numbers using the ten digits 0, 1, 2, ... , 9, and the position of each digit signifies a power of 10. For example, the number 230.5 base-10 (which for emphasis may be written 230.5_{10}) really means

$$230.5_{10} = 2 \times 10^2 + 3 \times 10^1 + 0 \times 10^0 + 5 \times 10^{-1}.$$

In an analogous fashion, **binary numbers** have the same structure, but with each digit (or bit) position being associated with a power of 2 instead of a power of ten. Of course, each digit is either a 0 or a 1. For example, 1101.1 base-2 (which for emphasis may be written 1101.1_2) actually means

$$1101.1_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1}.$$

The ‘dot’ here is, strictly speaking, the ‘binary point’ rather than the ‘decimal point’.

Binary-to-Decimal Conversion

From the example immediately above, it is easily observed that

$$1101.1_2 = (8 + 4 + 0 + 1 + .5)_{10} = 13.5$$

It is clear how this idea generalizes to a binary number of any size. For example, if we allow the B 's in the following expression to be 0's or 1's, then the base-10 equivalent, say x_{10} , is as given:

$$[B_n B_{n-1} \dots B_1 B_0 B_{-1} \dots B_{-j}] = [B_n \times 2^n + B_{n-1} \times 2^{n-1} + \dots + B_0 \times 2^0 + B_{-1} \times 2^{-1} + \dots + B_{-j} \times 2^{-j}]_{10}$$

Example: Convert 11101.101_2 to its base-10 equivalent.

Decimal-to-Binary Conversion

Decimal-to-binary conversion can be a little more tedious than binary-to-decimal conversion, especially if there are digits beyond the decimal point (i.e. if the base-10 number is non-integer). Large decimal integers may be converted to their binary equivalents via a process referred to as **double-dabble** (it may be noted that there is a corresponding technique by the same name that may be used to convert from binary to decimal). The double-dabble decimal-to-binary conversion is as follows:

Divide the base-10 number by two and write down the remainder (which will be 0 or 1); take the quotient as a new integer and repeat the process until the quotient is reduced to zero. **Reverse the remainders** (see arrow in the Figure D1.3 below) to obtain the digits of the binary equivalent of the original decimal number.

Illustration: Convert 207 to binary.

	<u>Remainder</u>	
2	207	1
	103	1
	51	1
	25	1
	12	0
	6	0
	3	1
	1	1
0		<div style="display: flex; align-items: center; justify-content: center;"> <div style="border-left: 1px solid black; border-right: 1px solid black; height: 100px; margin-right: 5px;"></div> <div style="font-size: 2em; margin-right: 5px;">↑</div> </div>

Figure D1.3 Decimal-to-binary conversion

Thus, $207_{10} = 11001111_2$.

Converting decimal fractions to binary form, is slightly more complicated because in general an ‘exact’ conversion requires an infinite number of bits – the exceptions to this statement are decimal fractions which are sums of the powers of $\frac{1}{2}$. As an example of such an exception consider the decimal number 0.75:

$$0.75_{10} = 0.5_{10} + 0.25_{10} = 1/2 + (1/2)^2 = 1 \times 2^{-1} + 1 \times 2^{-2} = 0.11_2$$

However, for a decimal number such as 0.7, there is no such simple conversion. In these not-so-simple cases there is available a procedure referred to as **reverse double-dabble**. The result will always have to be *truncated* to some satisfactory approximation. The procedure goes as follows:

Multiply the decimal fraction by two and write down the product and the integer carry (which will be 0 or 1); subtract the carry from the product to get a new decimal fraction and repeat the entire process. Continue this process until a desired precision is reached, at which point the **carries written in forward order** constitute the binary approximation of the original decimal fraction.

Illustration: Convert 0.7 to binary.

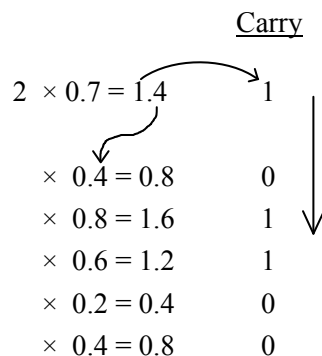


Figure D1.4 Conversion of a decimal fraction to binary.

Noting that the pattern repeats, it is obvious that $0.7_{10} = 0.101100110\dots_2$. Thus, if we truncate after seven bits, the finite-length approximation yields

$$\begin{aligned} 0.7_{10} &\approx 0.1011001_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} + 0 \times 2^{-5} + 0 \times 2^{-6} + 1 \times 2^{-7} \\ &= 0.6953125_{10}. \end{aligned}$$

Note that the truncation error for this case is about 0.67% (i.e. $0.0046875/0.7 \times 100\%$). One could imagine that unless steps are taken to minimize truncation errors, they could become significant as the number of arithmetic operations involving them increase.

Putting the previous two illustrations together, it is obvious that if a decimal number involves both an integer portion and a fractional portion, then both direct and reverse double-dabble will be required to convert the number to its binary equivalent. So to the approximations used above, for example,

$$207.7_{10} = 207_{10} + 0.7_{10} \approx 11001111_2 + 0.1011001_2 = 11001111.1011001_2.$$

Practice Examples: (1) Convert 105.8_{10} to binary. (2) Check that $0.75_{10} = 0.11_2$ using reverse double-dabble. Answer: (1) 1101001.110011 to six-bit precision.

Binary Addition and Subtraction

While we will not in any significant way apply the general concepts associated with the binary operations of addition and multiplication, we consider them briefly as some of the ideas will recur in the sections where we address digital logic gates.

Binary Addition

When adding the bits associated with two or more binary numbers it must be remembered that the base is now 2 so that while $(1+0)=1$, the expression $(1 + 1)$ which is 2_{10} is now 10_2 . That is, $(1+1)$ is 0 carry 1 in the binary system. This is completely analogous to $(4_{10} + 6_{10} = 0 \text{ carry } 1 \text{ or } 10_{10})$. The basic results for the *binary* addition of two digits are therefore

$$0+0 = 0 \text{ or '0 carry 0' or } 00$$

$$1+0 = 1 \text{ or '1 carry 0' or } 01$$

$$0+1 = 1 \text{ or '1 carry 0' or } 01$$

$$1+1 = 10 \text{ or '0 carry 1'}$$

Notice that 0 is the *additive identity*. Using the ideas above we note the following examples:

1	0	0	0
1	1	0	0
<u>+1</u>	<u>+1</u>	<u>+1</u>	<u>+0</u>
11	10	01	00

Adding binary numbers which contain multiple bits in each number follows the basic results given above. A few examples will suffice to get the idea:

111	111	111	111
<u>+000</u>	<u>+001</u>	<u>+011</u>	<u>+111</u>
111	1000	1010	1110

Subtraction

We are going to consider binary subtraction very briefly in a manner exactly analogous to decimal subtraction. There are other more powerful and useful techniques that also account for negative numbers (for example, a technique referred to as *twos complement*)

that may be encountered in future courses. Here, we simply do subtraction in the usual manner, borrowing as necessary. The basic results are

$$0 - 0 = 0$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

Using these results we can complete any subtractions for which the **subtrahend** is less than the **minuend** – i.e. cases for which the **difference** is positive. Consider the following examples:

$$\begin{array}{r}
 11101 \\
 - \quad 101 \\
 \hline
 \end{array}
 \qquad
 \begin{array}{r}
 \overset{1}{11101} \\
 - \quad 10011 \\
 \hline
 \end{array}
 \qquad
 \begin{array}{r}
 \overset{1111}{10000} \\
 - \quad 11 \\
 \hline
 \end{array}$$

Borrows

Binary Multiplication

Since we have not considered concepts associated with signed numbers and binary subtraction, we also restrict our consideration of multiplication to the case of unsigned binary numbers. The basic rules for the multiplication of single-bit binary numbers are as follows:

$$0 \cdot 0 = 0$$

$$1 \cdot 0 = 0$$

$$0 \cdot 1 = 0$$

$$1 \cdot 1 = 1$$

We have used the dot (\cdot) to indicate multiplication. We will find this symbolization useful in our introduction of the AND gate (see Section D1.3). The extension of this idea to multiple-bit numbers is completely analogous to decimal multiplication. That is, in its most basic form (but not necessarily in its implementation is an actual hardware circuit), the multiplication of unsigned binary numbers involves: (1) the multiplication of one binary number (the multiplicand) by the digits of another (the multiplier), one at a time, (2) position shifting between the results for each digit of the multiplier and (3) an addition of the results.

Example: Convert 15_{10} and 11_{10} to binary and multiply the results.

$$15_{10} = 1111_2 \text{ and } 11_{10} = 1011_2$$

$$\begin{array}{r} 1111 \text{ (multiplicand)} \\ \underline{1011 \text{ (multiplier)}} \\ 1111 \\ 1111 \quad \text{Partial} \\ 0000 \quad \text{products} \\ \underline{1111} \\ 10100101 \quad \text{Product} \end{array}$$

Notice that the product of two 4-bit binary numbers is an 8-bit binary number.

Practice Example: Convert 13_{10} and 11_{10} to binary and multiply the results.

Answer: 10001111

D1.3 Logic Variables, Gates and Truth Tables

Digital Logic and Logic Variables

A common dictionary definition of **logic** goes something like this: **logic** is the study of the principles of valid demonstration and inference or the study of the principles of reasoning. **Digital logic** involves the **testing of conditions by digital circuitry**: that is, the use of digital circuitry to determine if a condition is true or false. The circuit inputs and outputs will be so called **logic variables** having only two possible states. Thus, the whole of digital logic is based on the binary number system considered in the first two sections of these notes. As we shall see, by appropriately interpreting the inputs to a digital circuit that the circuit is ‘capable’ of making a logical decision. Thus, in general, a logical network operates on logic variables to produce some desired logical output. Again, it must be emphasized that in the present context, all inputs and outputs will take one of two states. These states can be represented by 0s (FALSE) and 1s (TRUE). Circuitry based on this simple two-state system of variables can be built up to express very complex relationships and interactions among any desired number of individual conditions. One fundamental reason for basing logical operations on the binary number system is that it is quite simple to design stable electronic circuits that can unambiguously switch back and forth between two clearly-defined states. Circuits may also be constructed so that they can remain indefinitely in one specified state (i.e. **on** or **off**) unless they are deliberately switched to the second state. This property makes it possible to use such circuits to construct computers which can remember sequences of events and adjust their behaviors according to specified changes in input.

Logic Operations and Gates

No matter how complicated the input-output relationship of a logic network may be it may always be broken down into *three basic logic operations*: AND, OR and NOT (or COMPLEMENT). Relatively simple circuits that may be used to implement these logical operations are referred to as **logic gates**. In the today’s world of digital circuitry, the components of these gates consist largely of combinations of some type of transistors – it is not the purpose of this module to explore the workings of the actual integrated

circuit (IC) technology, but rather to consider the high level logical operations which they perform. In fact, in a few cases we will consider how these gates may be implemented using simple switches.

Input and Output States

Consider first a simple two-state circuit as shown in Figure D1.5 in which the **input** is the position (i.e. state) of a switch which can be CLOSED (represented by a 1) or OPEN (represented by a 0). We'll use A as the *logic variable* representing the two possible states of the switch. Meanwhile, the input state of the switch determines the **output** state of the light. In this example, the light is either ON or OFF also and we will use F as the *logic variable* to represent the state of the output. Thus, A and F can take values of 1 or 0. As shown in the figure, when $A = 1$, $F = 1$ (i.e. when the switch is CLOSED, the light is ON), and when $A = 0$, $F = 0$ (i.e. when the switch is OPEN, the light is OFF). A table of all the possible input-output conditions is referred to as a **truth table**.

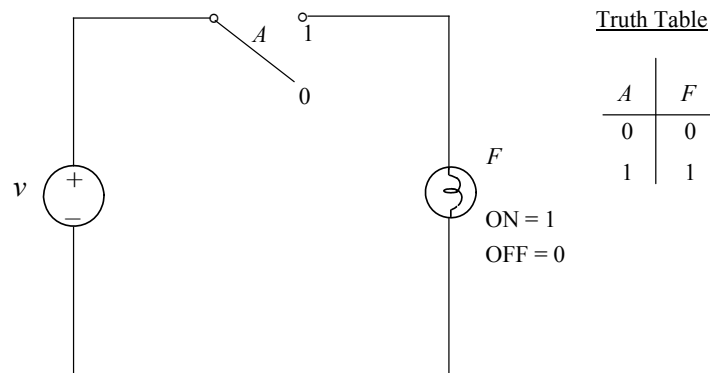


Figure D1.5 A simple binary (two-state) system consisting of a switch and a light.

It is trivially obvious that, in this example, the input-output relationship is given by

$$F = A.$$

The AND Gate

Consider next a simple two-state circuit as shown in Figure D1.5 (a) in which the **inputs** are the states of the two series-connected switches and the **output** is again the state of the light. We use A and B as the input logic variables and F as the output logic variable.

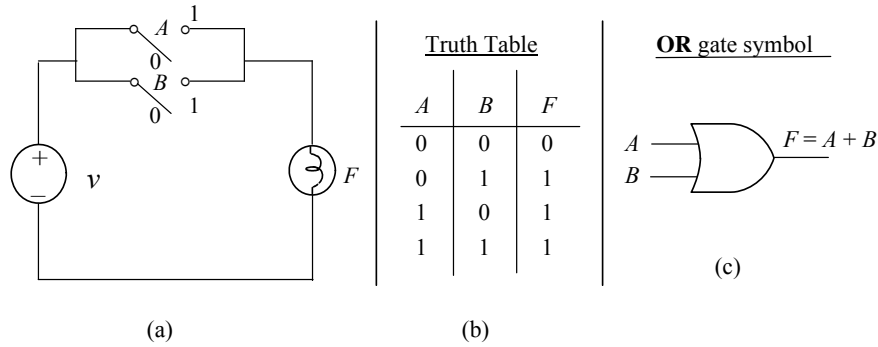


Figure D1.7 (a) A circuit in which the output (light) is controlled by two switches connected in parallel; (b) the truth table for the circuit; (c) the **OR** gate symbol with two inputs which corresponds to the operation of this circuit.

As for the **AND** gate, we may make four logical statements regarding the **OR** gate:

- (1) when both A and B are 0 (i.e. both switches OFF) F is also 0 (i.e. the light is OFF);
- (2) when A is 1 and B is 0, F is 1;
- (3) when A is 0 and B is 1, F is 1;
- (4) when A is 1 and B is 1, F is 1.

The logical **OR** is written symbolically as

$$F = A + B \quad \dots\dots (D1.3)$$

where $A + B$ represents “ A **OR** B ”. It is immediately obvious that, except for the case of $1 + 1 = 1$, the **OR** behaves as ordinary addition of the bits 0 and 1 because

$$0 + 0 = 0 \quad 0 + 1 = 1 \quad 1 + 0 = 1,$$

and we have succeeded in building such a gate by connecting two switches in parallel. Of course, $1 + 1 = 1$ because, the light remains lit (i.e. the output undergoes no change) if the $1 + 0$ or $0 + 1$ conditions (the cases where only one switch is closed) are changed to the case where the other switch also closes.

The idea may be naturally extended to **OR** gates with more than two inputs (say n) and such gates may be realized by connecting n switches in parallel. The **OR** operation is also *commutative* and *associative* and the output (F) of a three-input **OR** gate may be written as

$$F = A + B + C = (A + B) + C = A + (B + C) = B + A + C \text{ etc.} \quad \dots\dots (D1.4)$$

To summarize, the output F of an **OR** gate will be 1 if *any* of the inputs is 1.

The NOT Gate

As a third example in our preliminary consideration of logic gates, we address the idea of *negating* an input. This operation is referred to as the **logical NOT** and it is performed by a **NOT** gate or **inverter** as symbolized in Figure D1.8 (a). This is a single-input gate and its output always takes the opposite state of the input.

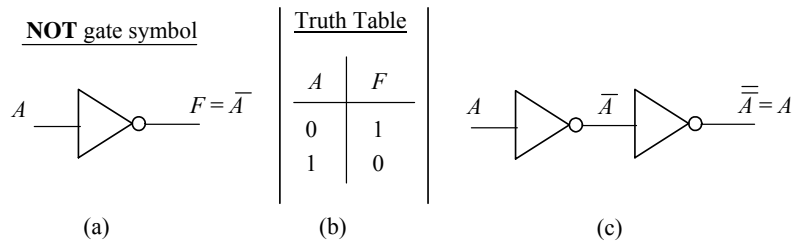


Figure D1.8 (a) The **NOT** gate; (b) the **NOT** truth table; (c) a cascade connection.

We write this logical inversion with F as the output as

$$F = \bar{A} \dots\dots (D1.5)$$

The bar over the A indicates “ A **NOT**”. The **NOT** operator is also referred to as the **negation** or **complement**. The truth table in Figure D1.8 changes a 0 to 1 and vice versa. If two **NOT** gates are cascaded together as shown in Figure D1.8 (c), the second gate negates the negation of the first gate and the final output is the same as the initial input. Symbolically,

$$\bar{\bar{A}} = \overline{(\bar{A})} = A \dots\dots (D1.6)$$

A simple switch implementation of the **NOT** gate involves the concept of a **normally-closed switch**. The “normal” position of a switch defines its position when it is not being pressed or perturbed in some other manner (i.e. not being “thrown”). A normally closed switch is one that is typically spring loaded and conducts electricity when it is not being pressed. Pressing the switch stops the flow of electricity. The switches in the circuits we have looked at so far in this section have been **normally-open** – that is, a **normally-open** switch is one that normally prevents current flow and which allows current to flow only when it is perturbed. We have assumed that a switch in the ‘unperturbed’ position corresponds to the logical 0. Notice that a logical 0 in Figures D1.5-D1.7 means that the switch is open. Now consider Figure D1.9. In this figure, the

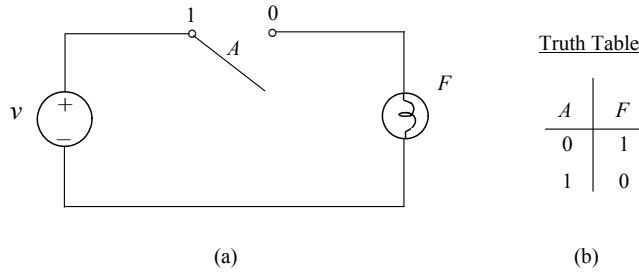


Figure D1.9 (a) A normally-closed (nc) switch; (b) the truth table for the nc switch.

unperturbed position (i.e. position 0) is the closed position. Thus, when the logical input is 0, the light is ON and when the switch is perturbed (i.e. in position 1, as it is shown in the figure), the light is OFF. Notice that the truth table for the normally closed switch is exactly that of the **NOT** gate of Figure D1.8.

The **NOT** may also be implemented using switches and relays as seen in Figure D1.10. A relay may be thought of as an electromagnetically-activated switch. When A (the input) is OFF, which it is normally (and the relay is deactivated), RY1 is CLOSED the light (the output) is ON. However, when A is ON the relay is activated, RY1 OPENS and the light goes OFF. Obviously, the truth table for this situation is the same as in Figure D1.9.

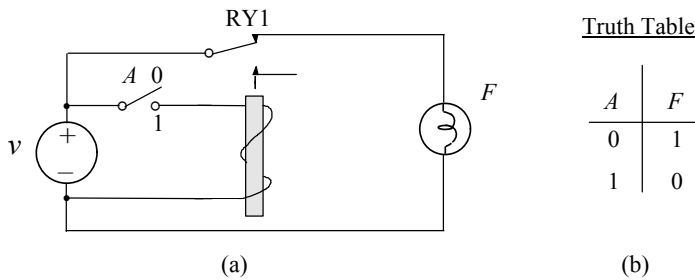
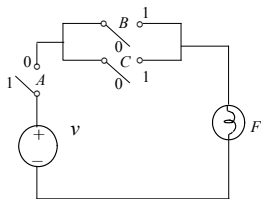


Figure D1.10 (a) A relay implementation of a **NOT** operation and (b) its truth table.

Example: Write the expression for the output logic function F of the circuit shown in the figure below, and implement the function using the gates examined in this section. Construct the truth table and verify that the fifth line of the table checks against the expression for F derived here.



The NAND and NOR Gates

While it is beyond the scope of this course to investigate the reasons, it is interesting to note that digital electronic circuits may be more naturally constructed to give the **negations** of the **AND** and **OR** gates rather than the gates themselves. Thus, it is very worthwhile to consider the **NOT-AND** (or **NAND**) and **NOT-OR** (or **NOR**) gates. As intimated, these gates simply provide the inversions of their **AND** and **OR** counterparts.

The output F of a two-input **NAND** gate may be written as

$$F = \overline{A \cdot B} \dots\dots (D1.7)$$

while that for a two-input **NOR** gate may be written as

$$F = \overline{A + B} \dots\dots (D1.8)$$

These gates and their corresponding truth tables are found in Figure D1.11. In the section which follows we will see that, amongst other important features, the **NAND** and **NOR** gates can be used to construct the other gates we have studied.

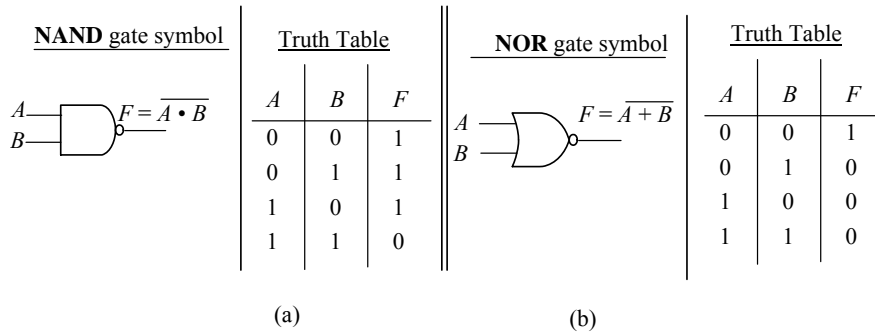


Figure D1.11 (a) The **NAND** gate and truth table; (b) the **NOR** gate and truth table.

Example: (a) Develop the truth table for the expression $F = \overline{A + B}$ and compare it to that for the **NAND** gate. (b) Develop the truth table for the expression $F = \overline{A \cdot B}$ and compare it to that for the **NOR** gate.

The Exclusive-OR (XOR) Gate

As a final preliminary example, we consider the **Exclusive-OR** (abbreviated **XOR**) operation. Again, in the next section we'll consider its implementation using gates considered previously. The **XOR** gate is a two-input module whose output equals 1 when one, but **not both**, of its inputs is 1. We write this operation as

$$A \oplus B = F \dots\dots\dots (D1.9)$$

where “ $A \oplus B$ ” means “ A or B but not both.” The symbol and truth table for the **XOR** gate are found in Figure D1.12.

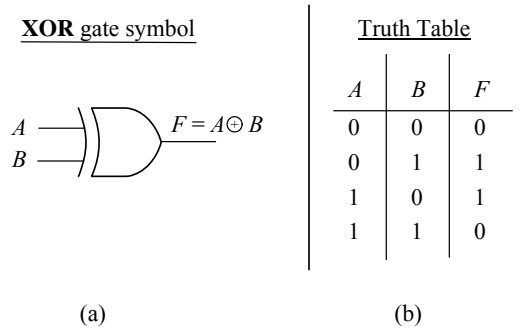


Figure D1.12 (a) The **XOR** gate and (b) its truth table.

From the **XOR** truth table, we see that this operation gives

$$0 \oplus 0 = 0 \quad 0 \oplus 1 = 1 \quad 1 \oplus 0 = 1 \quad 1 \oplus 1 = 0,$$

and we see that it differs from the **OR** operation in that $1 \oplus 1 = 0$.

Example: Determine the truth table for an exclusive-**NOR** (**X-NOR**) – i.e the **negation** of the **XOR**.

D1.4 Combinational Logic

Combinational logic consists in essence of an interconnected group of gates which accepts a combination of input digital signals and produces an output (in effect, a *decision*) based on these inputs. The outputs at any instant are completely determined by the input values. It is necessary that we develop a set of mathematical tools for manipulating the, sometimes complex, expressions which relate the inputs and outputs of the digital circuitry.

D1.4.1 Boolean Algebra

Boolean algebra is a powerful tool for addressing our need to deal with a mathematical analysis of combinational logic expressions. Two distinguishing features of this algebra are:

- All variables are logic variables restricted to the binary values 0 and 1.
- All operations are defined in terms of the logical **AND**, **OR**, and **NOT**.

It may be pointed out that certain so-called Boolean expressions take on familiar forms. For example, recalling that ‘dot’ means **AND**, while ‘plus’ means **OR** we have

$$A + 0 = A \quad A \cdot 1 = A \quad A \cdot (B + C) = A \cdot B + A \cdot C$$

However, in other cases, it may at first seem a little strange to write what is the nonetheless correct Boolean expression $1 + 1 = 1$ (recall, this simply says that ‘1 **OR** 1 gives an output of 1).

To immediately facilitate the implementation of combinational logic, we tabulate below some of the important Boolean algebra theorems. We will validate some of these theorems as we progress, but we are mainly interested in seeing how they allow us to simplify various logical expressions. Notice that we have used the variables X and Y , although we could use any variable name we wish. Remember that these variables can take only values of 0 or 1. Also, from now on, except for emphasis, we will not use the ‘dot’ to indicate the **AND** operation; rather, XY will be understood to mean $X \cdot Y$ since the logical **AND** acts like multiplication.

Table of Boolean Theorems

Table of Boolean Theorems

AND Theorems

1. $X \cdot X = X$

2. $X \cdot \bar{X} = 0$

OR Theorems

3. $X + 1 = 1$

4. $X + X = X$

5. $X + \bar{X} = 1$

Absorption Theorems

6. $X + XY = X$

7. $X(X + Y) = X$

8. $XY + X\bar{Y} = X$

9. $(X + Y)(X + \bar{Y}) = X$

DeMorgan's Theorems

10. $\overline{X \cdot Y} = \bar{X} + \bar{Y}$

11. $\overline{X + Y} = \bar{X} \cdot \bar{Y}$

Just as ‘multiplication comes before addition’ in the absence of parentheses, so ‘**AND** comes before **OR**’. For example, $X + XY = X + (X \cdot Y)$. Also, it is important to note that any operation appearing under a negation bar is to be done *before* the negation. For example, $\overline{X \cdot Y} = \overline{(X \cdot Y)}$. Of course, parentheses always take precedence in the order of operation.

The Boolean theorems can be proven by constructing truth tables. For example see Figure D1.13 in which it is verified that $XY + X\bar{Y} = X$. This is one of the **absorption theorems**, so named because the expression equals X regardless of the value of Y , so that Y is seemingly *absorbed* and does not affect the result. One can imagine that the idea of absorption is important because it may lead to a simplification of the gate network required to implement a piece of binary logic.

X	Y	XY	\bar{Y}	$X\bar{Y}$	$X\bar{Y} + XY$
0	0	0	1	0	0
0	1	0	0	0	0
1	0	0	1	1	1
1	1	1	0	0	1

Figure D1.13 Truth table proving the Boolean theorem $XY + X\bar{Y} = X$.

Also, given the **AND** and **OR** theorems along with the commutative, associative and distributive laws, any other Boolean relations may be derived.

Example: Show that $(X + Y)(X + \bar{Y}) = X$.

Finally, we note that entries 9 and 10 of the theorems are particular cases of the following important statement referred to as **DeMorgan's Rule**:

The complement of any Boolean expression may be obtained by negating each variable individually, changing each **AND** operation to an **OR**, and changing each **OR** operation to an **AND**.

This often proves to be a very powerful procedure for reducing the number of gates required to implement a logical function as combinational network.

Example: Suppose a logic function F is given by

$$F = \overline{A\bar{B} + C} \cdot (\bar{A}C + B).$$

It would appear initially that we require a total of eight gates (3 **AND**, 2 **OR** and 3 **NOT**) and several interconnections in order to implement F . However, consider what happens when we first apply DeMorgan's rule:

Thus, we have absorbed the A variable entirely and reduced the implementation of F to just two gates (an **AND** and a **NOT**).

D1.4.2 Two Techniques for Logic Network Minimization

As we have sought to emphasize, it is generally important to be able to implement a logical network using as few gates as possible. There are several techniques available and of those we consider two in this section: (1) the **standard sum of products** (SSOP) and (2) the **standard product of sums**. We will illustrate these methods using examples.

The Standard Sum of Products

Suppose we need to design a logic network with three Boolean input variables A , B , and C and an output variable F which is determined as follows: $F = B$ when $A = 0$ and $F = C$ when $A = 1$. We want to minimize the gates involved in the final expression for F .

Step 1: First we develop the truth table as given below in Figure D1.14:

Line No.	A	B	C	F	Minterms (m)
0	0	0	0	0	
1	0	0	1	0	
2	0	1	0	1	$m_2 = \overline{A}B\overline{C}$
3	0	1	1	1	$m_3 = \overline{A}BC$
4	1	0	0	0	
5	1	0	1	1	$m_5 = A\overline{B}C$
6	1	1	0	0	
7	1	1	1	1	$m_7 = ABC$

Figure D1.14 Truth table for standard sum of products example (SSOP).

For discussion purposes we have numbered the eight lines of the truth table from 0 to 7, which happens to also correspond to the decimal equivalents of the increasingly large **input word** $[ABC]$.

Step 2: Next, consider only those lines for which $F = 1$. For example, on line 2, $F = 1$ when $[ABC] = 010$; on line 3, $F = 1$ when $[ABC] = 011$, and so on. We note that the function

$$m_2 = \overline{A} \cdot B \cdot \overline{C} = \overline{A}B\overline{C}$$

has a value of 1 *if and only if*

$$A = 0, \quad B = 1, \quad \text{and} \quad C = 0.$$

Such a product in which each of the *input* variables appears **once** (either complemented, or uncomplemented) is called a **minterm**. Thus, as an extra example, we note that the minterm for line 3 is

$$m_3 = \overline{A} \cdot B \cdot C = \overline{A}BC$$

which clearly has a value of 1 if and only if

$$A = 0, \quad B = 1, \quad \text{and } C = 1.$$

In this step, then, **we write the minterms, associated with each line where $F = 1$** by

“writing the logical **AND** ‘product’ of all input variables and complementing each variable having a value of 0 on the line in question.”

The remaining minterms associated with this example are found in lines 5 and 7 of the table. In all cases, they identify input words that produce $F = 1$.

Step 3: Because $F = 1$ if and only if any of the minterms has a value of 1 (i.e. it is 0 for any other input combinations), the original logical function described in Step 1 is simply the logical **OR** ‘sum’ of these minterms; i.e.

$$F = m_2 + m_3 + m_5 + m_7 = \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}C + ABC \quad \text{SSOP ... (D1.9)}$$

The function F written in this form is an example of the general form referred to as the **standard sum of products (SSOP)**. It may be implemented as an **AND-OR network** as shown in Figure D1.15. The connecting lines between the inputs and the **AND** gates have been omitted so that the one-to-one relationship between the minterms and these gates may be clearly seen.

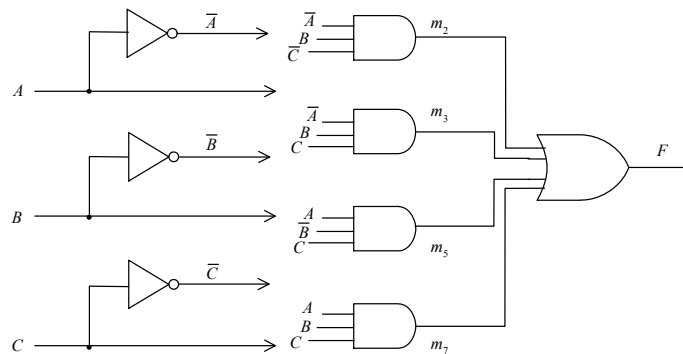


Figure D1.15 AND-OR network for the SSOP example discussed above.

We note that in the actual hardware implementation of such a network the input words containing both the direct and complemented inputs are often stored in electronic entities referred to as *registers*.

While we have accomplished an implementation of the function via the SSOP technique, it is important to note that the network may often be further simplified by applying the Boolean theorems found on page 20 of this module. Consider the following:

$$m_2 + m_3 = (\bar{A}B)\bar{C} + (\bar{A}B)C = \bar{A}B \text{ from Absorption Theorem 8}$$

$$m_5 + m_7 = A\bar{B}C + ABC = (AC)\bar{B} + (AC)B = AC \text{ from Absorption Theorem 8}$$

Therefore, a minimized sum of products is given by

$$F = (m_2 + m_3) + (m_5 + m_7) = \bar{A}B + AC \dots \text{(D1.10)}$$

This expression represents a *minimized* SOP and it requires only the four gates found in Figure D1.16 instead of the eight in Figure D1.15. Thus, we see that simplifications may be effected by applying Boolean absorption whenever the SSOP function contains pairs of minterms that differ from each other only by the negation of one variable.

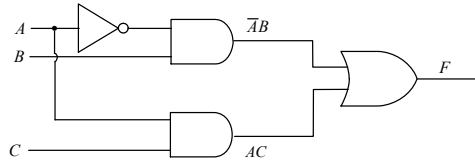


Figure D1.16 AND-OR network for the minimized SSOP example discussed above.

The Standard Product of Sums

A second technique exists for developing an expression for F using only the $F = 0$ lines of the truth table. Each $F = 0$ line of the table (see Figure D1.17) is now associated with a *negated minterm* called a **maxterm**. A maxterm is obtained for each such line by “writing the logical **OR** ‘sum’ of all input variables and negating each variable whose value is a 1.”

Line No.	A	B	C	F	Maxterms (M)
0	0	0	0	0	$M_0 = A + B + C$
1	0	0	1	0	$M_1 = A + B + \bar{C}$
2	0	1	0	1	
3	0	1	1	1	
4	1	0	0	0	$M_4 = \bar{A} + B + C$
5	1	0	1	1	
6	1	1	0	0	$M_6 = \bar{A} + \bar{B} + C$
7	1	1	1	1	

Figure D1.17 Truth table for standard product of sums example (SPOS).

Since we want $F = 0$ when any one of the of the listed maxterms is 0, we may write as another logic function for F

$$F = M_0 \cdot M_1 \cdot M_4 \cdot M_6 = (A + B + C)(A + B + \bar{C})(\bar{A} + B + C)(\bar{A} + \bar{B} + C) \text{..SPOS(D1.11)}$$

A function in this general form is referred to as a **standard product of sums** (SPOS). The function may be diagrammed as an **OR-AND** network as in Figure D1.18 where the one-to-one relationship between the maxterms and the **OR** gates is obvious.

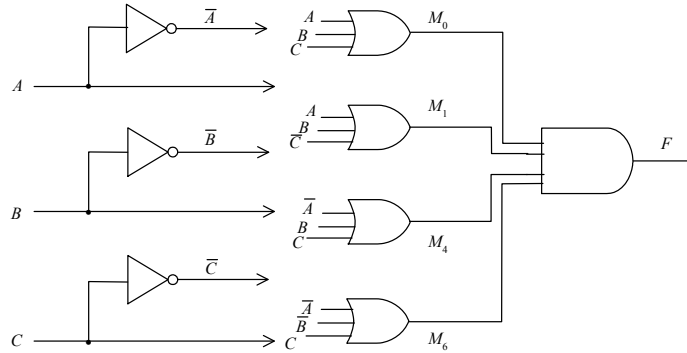


Figure D1.18 OR-AND network for the SPOS example discussed above.

From Theorem 9 (i.e. $(X + Y)(X + \bar{Y}) = X$) on page 20, we see that whenever two maxterms differ only by the negation of one variable, then they may be simplified by the process of absorption. Note that in equation (D1.10) we have two such products:

$$M_0 \cdot M_1 = [(A + B) + C][(A + B) + \bar{C}] = A + B$$

and

$$M_4 \cdot M_6 = [(\bar{A} + C) + B][(\bar{A} + C) + \bar{B}] = \bar{A} + C$$

Hence a *minimized* form of the POS for this example is

$$F = (M_0 M_1)(M_4 \cdot M_6) = (A + B)(\bar{A} + C) \dots \text{(D1.12)},$$

the implementation of which is found in Figure D1.19. It should be noted that (D1.12) and (D1.10) represents the same function F . (We'll validate this with a truth table in the practice problems at the end of this subsection).

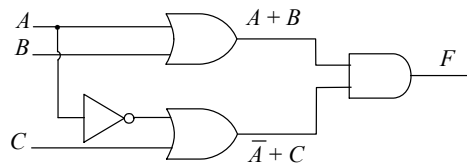


Figure D1.19 OR-AND network for the minimized POS example discussed above.

(4) (a) It is required that the seatbelt alarm in a vehicle should sound whenever the driver is in place and the driver seatbelt is not activated or whenever the passenger is in place and the passenger seatbelt is not activated. Determine and symbolize the four input variables and the output variable in this situation and implement the logic network which corresponds to the requirement. (b) Suppose next that a lamp is illuminated when a safe condition exists. Use part (a) and DeMorgan's theorem to determine and implement this condition. Read 'safe' or 'light on' as being the complement of 'unsafe' or 'alarm'.

(5) Use a truth table to validate the equivalency of the minimized SOP and POS expressions for the example given in equations (D1.10) and (D1.12).

Unit D2: Sequential Logic – A Brief Introduction

D2.1 Definition

Up to this point in our discussion of digital logic, we have considered only those situations in which the *output* is *completely determined* by the *input* signals – i.e. we have examined only **combinational logic** circuits. As the name implies, however, **sequential logic** refers to digital logic circuits in which the output depends not only on *the current combination* of input signals but also the *past history* of the system. We might use the terminology that the system has a **memory** which contains information on the system's history. A simplistic example of this is a lamp controlled by a *toggle* switch (an ordinary switch as in a house circuit). Suppose the switch is **set** – that is to say an input (a push) closes the switch and turns on the lamp. In our previous terminology, this amounts to an input of 1 causing an output of 1. When the push is released, this kind of switch remains closed and the output remains high – i.e. maintains its value of 1. Thus, after a **set** (a push or an **input** of 1) an input of 0 (no push) does not change the state of the output since the lamp **remains on**. Now to **reset** the state of the lamp, the toggle switch must be **reset** to its other position – that is, it must be pushed to its other position. In this case, the lamp will **remain off** until the switch is again **set**. We may summarize the example:

set the switch with a push (**set** = 1, **reset** = 0) and the lamp goes on (output of 1);
release the switch (**set** = 0, **reset** = 0) and the lamp remains on (output of 1);
reset the switch with a push (**set** = 0, **reset** = 1) and the lamp goes off (output of 0);
release the switch (**set** = 0, **reset** = 0) and the lamp remains off (output of 0).

A device that maintains the value of the output, even though the input may change is referred to as a **latch**. Thus, the basic toggle switch is a simple example of such a device.

These ideas may be better explored using the push buttons and relay circuitry encountered in **Laboratory Exercise D1**. See Figure D2.1. In this figure, PB1 is a *normally open* push-button switch and PB2 is a *normally closed* push-button switch. Thus, a logic level of 0 is associated with PB1 in the open position and PB2 in the closed

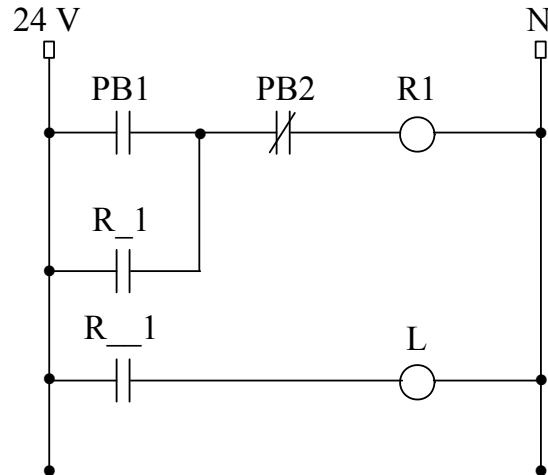


Figure D2.1 A push-button/relay implementation of a latch.

position while a logic level of 1 is associated with PB1 in the closed position and PB2 in the open position. R1 is a relay which activates the *normally open* R_1 and R__1 switches. That is, when R1 is energized, R_1 and R__1 will close and the lamp L will light. Now let's discuss this ladder circuit as follows:

Supposed PB1 is initially open. When PB1 (rename it to S) is pushed ($S=1$) it **sets** the lamp to the ON state because it activates the relay which closes R__1 (of course, R1 activation also closes R_1). Call this output state Q . Suppose PB1 is released and returns to the open position (i.e. $S = 0$). Because the previous $S = 1$ closes R_1, even when $S = 0$ the relay R1, along with its contact R__1, remains activated and the light remains ON ($Q=1$). That is, the output is **latched**.

Suppose, next, that PB2 (rename it to R) is pushed ($R = 1$) while still $S = 0$. Then, the normally-closed PB2 opens, the relay R1 deactivates and R_1 and R__1 open. The fact that R__1 opens means that the lamp goes OFF. That is, $R = 1$ **resets** the system. The lamp will remain in the OFF state when $R = 0$ (i.e. when PB2 is released and closes again) unless PB1 is **set** to $S=1$.

This **set-reset latch (SR latch)**, or memory cell, holds only one logic variable Q capable of taking two *stable* states (0 or 1). Because of this two-state possibility, the circuit is generally known as a **bistable**. The *SR* latch may be written in a generalized way as shown in Figure D2.2(a). The n subscripts in the truth table of Figure D2.2(b) represent the n th state of the latch while $n+1$ subscripts represent the next state beyond the n th.

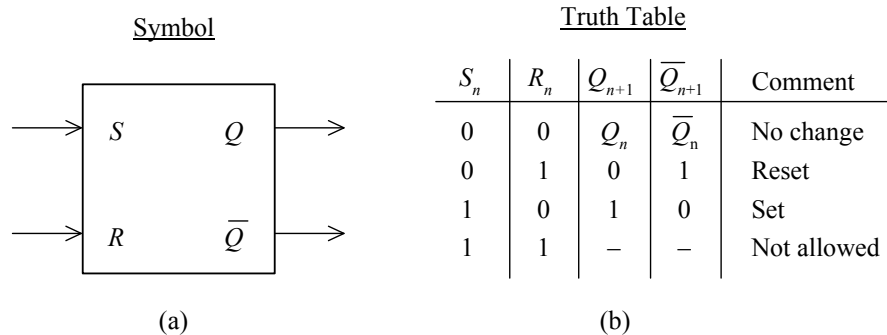


Figure D2.2 The symbol (a) and truth table (b) for a *SR* latch.

In digital circuitry, the **SR latch** may be implemented using logic gates. If the set and reset are executed with a timing waveform containing digital pulses such as in Figure D1.2(b) and referred to as a **clock signal**, the system is called a clocked **SR flip-flop**. The digital design is such that both the logic variable Q as well as its complement \overline{Q} may be available as outputs and this is the reason for both being included in the truth table. Notice that the latch cannot be in two states (i.e. set and reset) at the same time – this explains why $S = R = 1$ is not allowed in the truth table.

This brief introduction to sequential logic forms the basis for further discussion of any digital circuitry which requires memory in the form of storage units that temporarily hold digital information for subsequent processing.