

# DIGITAL ELECTRONICS

## Part 1: Combinatorial Logic

### INTRODUCTION:

Up to now, our discussions have focused on analog signals which could take on a continuous set of values limited only by the power supply capabilities. Digital signals, in contrast, are designed to take on a discrete set of values. The difference between analog and digital signals, therefore, is that a continuous set of electrical values are meaningful for analog signals, whereas only a specifically defined set of values are meaningful for the digital case.

Of course, the idea of a truly continuous set of values all being meaningful is unrealistic in reality since this requires precision to a potentially infinite set of significant digits. Our primary interest for now is in signals taking on a binary set of values. With such a scheme, representation of more than two values requires a set of connections and a description of the signal encoding method. Thus, analog and digital signals can be contrasted in the following way: an analog signal requires only one signal connection but some means of practically distinguishing different signal levels while a (binary-based) digital signal requires a set of connections and a defined scheme for encoding / decoding the set of values.

As before, our discussion will focus on voltage-based signals. However, there are several definitions of binary states in terms of voltages so that incompatibilities exist even in the binary world. Following are some examples of defined binary-based voltages:

<b>Transistor-transistor logic (TTL):</b>	$0V \leq V_L \leq 0.8V$	(Logic "low" or "0")
(Widely used; will be the type used in lab)	$3.5V \leq V_H \leq 5.0V$	(Logic "high" or "1")
(Uses the "typical" +5V supply)		
<b>Emitter-coupled logic (ECL):</b>	$-1.8V \leq V_L \leq -1.6V$	(Logic "low" or "0")
(Almost obsolete; used for high-speed systems)	$-1.0V \leq V_H \leq -0.8V$	(Logic "high" or "1")
(Typically uses a -5.2V supply)		
<b>RS-232C (Serial communications):</b>	$3.0 \leq V_L \leq 12V$	(Logic "low" or "0")
(High "noise margin")	$-12V \leq V_H \leq -3.0V$	(Logic "high" or "1")
(Typically uses a $\pm 12V$ supply)		<b>(Go figure!!!)</b>

Even though the definitions are quite different, one common characteristic is clear: the voltages to be interpreted as "0" are distinctly different from those to be interpreted as "1." Thus, there is no possibility of a "maybe" condition.

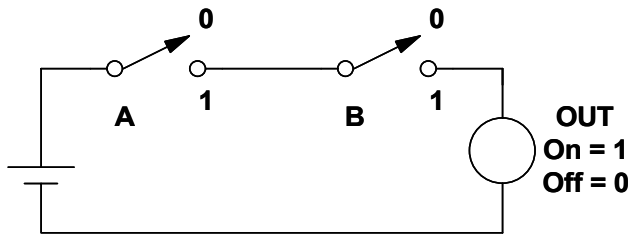
Having illustrated the existence of binary-based voltage systems, and having established the need for specification of the voltages defining the binary states, we will no longer be interested in voltages. From this point onwards, our focus will be on binary sets of values and logical relationships.

### BASIC BINARY LOGICAL OPERATIONS

At this point, we will introduce and illustrate three basic logic operations: **AND**, **OR**, and **NOT**. With these, we can make the following **assertion**: "**all digital electronic systems can be implemented with suitable combinations of the operations AND, OR, and NOT.**" In fact, we will see that OR and AND can be derived from each other plus NOT's, so that only two are really necessary. Moreover, the simplest operation to implement with transistor-based circuitry includes the NOT, so that only one is necessary if we choose the most directly realizable operations as the basis. Nevertheless, it is suitable to take the set of three stated above and they provide a straightforward approach to developing logic-based digital electronics.

### BASIC OPERATIONS

Circuits using mechanical switches and lamps are useful for introducing the basic logic operations. Consider for example Circuit 1 below:



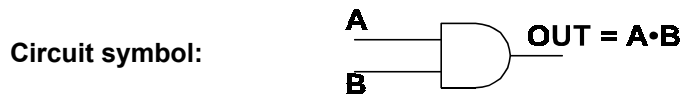
**Circuit 1**

Truth Table

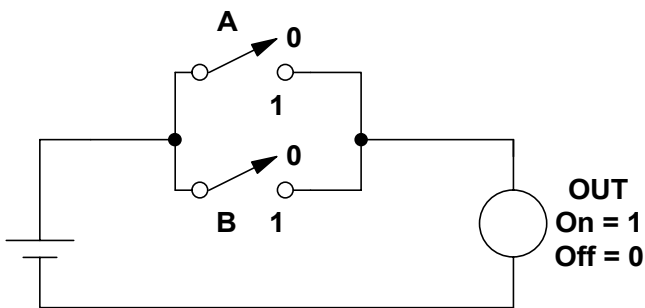
A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

This circuit implements the **AND** operation; that is **OUT** is 1 when A **and** B are 1. The “truth table” describes the relationship in the form of a table. The **algebraic** and **circuit-symbol** representations of this functional relation are:

**Algebraic representation:**  $OUT = A \cdot B$



Next, consider Circuit 2 below:



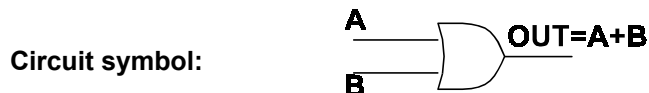
**Circuit 2**

Truth Table

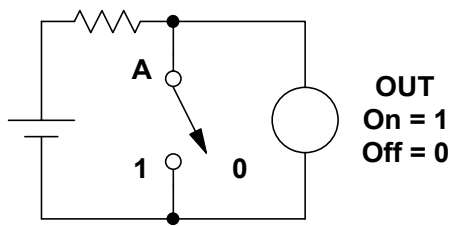
A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

This circuit implements the **OR** operation; that is **OUT** is 1 when either A **or** B are 1. As before, the truth table describes the function in a tabular format. For this function, the algebraic and circuit representations are:

**Algebraic representation:**  $OUT = A + B$



Finally, consider Circuit 3 below:



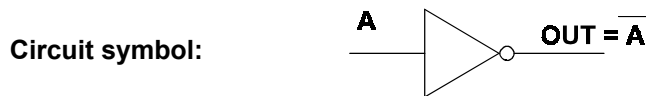
**Circuit 3**

Truth Table

A	OUT
0	1
1	0

This circuit implements the NOT operation; that is, OUT is **NOT** what A is. Logic and circuit representations of this function are:

**Algebraic representation:**  $OUT = \bar{A}$

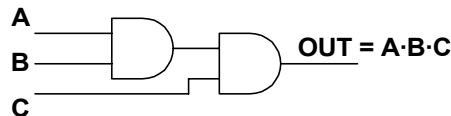


### MATHEMATICAL PROPERTIES OF THE BASIC FUNCTIONS

For future use, we need to be aware of the following properties of the basic functions:

**Commutation:** Both AND and OR are commutative. That is  $A+B = B+A$  and  $A \cdot B = B \cdot A$ ;

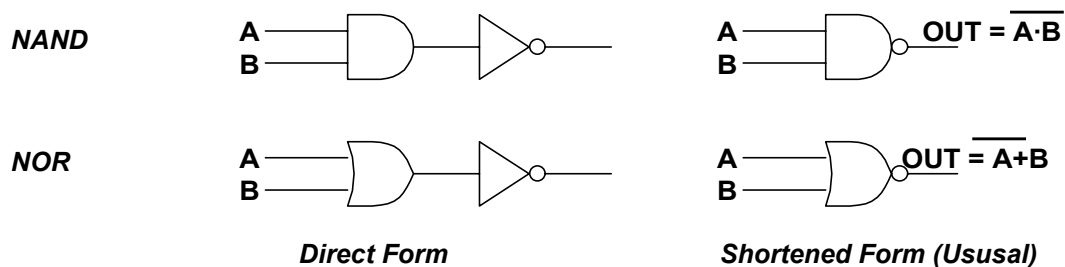
**Associativity:** Both AND and OR terms may be associated two-by-two:  $A+B+C = (A+B)+C = A+(B+C)$ , etc. Associativity is our basis for implementing functions of more than two variables. For example, using the associativity property, we can implement the 3-variable AND,  $A \cdot B \cdot C = (A \cdot B) \cdot C$  as:



**Distribution:** This involves “mixtures” of AND’s and OR’s and introduces the hierarchy of functions and the need for “grouping” by means of parentheses as we do with “ordinary” algebra. For example, the relation  $A \cdot (B+C) = A \cdot B + A \cdot C$ ; however,  $A \cdot (B+C) \neq A \cdot B + C$ .

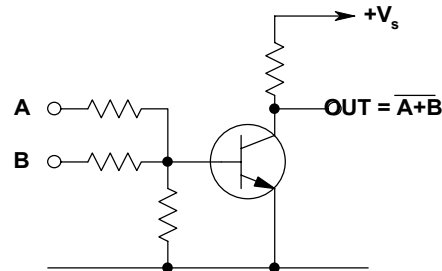
### BASIC FUNCTIONS INCORPORATING NOT

As indicated above, the most directly realizable transistor-based circuits for implementing the logic functions electronically yield the NOT AND, or **NAND**, and NOT OR, or **NOR**, functions as indicated below:



### TRANSISTOR-BASED NOR CIRCUIT

The circuit shown below is a crude version of the NOR function implemented with transistors:



### SOME BASIC RELATIONS

With what we now know, we can recognize a set of basic relations as indicated below. In doing so, it is useful to recall that we will make use of binary variables and binary functions. The results from the functions will be the **dependent** variables, while the “inputs” will be the **independent** variables. Moreover, symbols will indicate variables while specific values will indicate **binary constants**; specifically, “0” and “1.”

- |    |                                       |                                   |
|----|---------------------------------------|-----------------------------------|
| 1. | $A \cdot 0 = 0$                       | $A + 0 = A$                       |
| 2. | $A \cdot 1 = A$                       | $A + 1 = 1$                       |
| 3. | $A \cdot A = A$                       | $A + A = A$                       |
| 4. | $\overline{A \cdot 0} = 1$            | $\overline{A + 0} = \overline{A}$ |
| 5. | $\overline{A \cdot 1} = \overline{A}$ | $\overline{A + 1} = 0$            |
| 6. | $\overline{A \cdot A} = \overline{A}$ | $\overline{A + A} = \overline{A}$ |
| 7. | $\overline{(\overline{A})} = A$       |                                   |

**Useful Notes:** Note the particular behavior of **AND**, that **anything**  $\cdot 0 = 0$ ; also note for **OR** that **anything**  $+ 1 = 1$ .

These behaviors lead to the “gate” terminology. That is, for AND, the gate is closed for any input = 0 in the sense that values at the other input have no effect on the output. Similarly with an OR, any input = 1 controls the output regardless of the values at other inputs.

### DeMORGAN'S THEOREMS

As indicated above, it is possible to implement the AND from OR's and NOT's; as well, the OR function can be implemented from AND's and NOT's. Without making a formal derivation of these, we can see the relation by use of the multi-column truth table below:

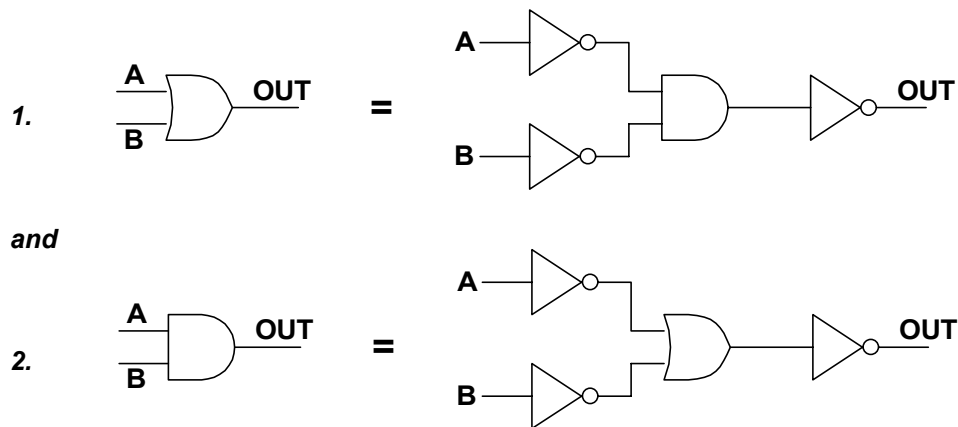
Truth Table

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

From the table, we can identify one of DeMorgan's theorems:

1.  $A + B = \overline{(\overline{A} \cdot \overline{B})}$ , or  $\overline{A + B} = \overline{A} \cdot \overline{B}$  ; the second theorem is similar:
2.  $A \cdot B = \overline{(\overline{A} + \overline{B})}$ , or  $\overline{A \cdot B} = \overline{A} + \overline{B}$

From the standpoint of circuitry, these relations are:



**APPLICATIONS**

We are now in the position to apply the concepts and relations developed above which form the basis for **Combinatorial Logic**. One useful class of problems involve testing a combination of binary values to see if they match a predetermined set. This is the basic problem of "decoding" and has a "lock and key" character. Following are examples of this:

**Example 1:**

Write the logical relation expressed by the truth table given below and sketch a circuit for implementing the function using **NOT's** along with basic two-input **AND's** and **OR's** as necessary.

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

**Solution:** The output is high ("1") for three cases as expressed by rows 3, 4, and 8. Verbally, we can express the relation as  $OUT = 1$  for case<sub>3</sub>, OR case<sub>4</sub>, OR case<sub>8</sub>. Thus the initial expression is:

$$OUT = case_3 + case_4 + case_8$$

In addition, each case must yield "1" for the specific combinations listed (but only for those!). This can be accomplished with AND's as follows:

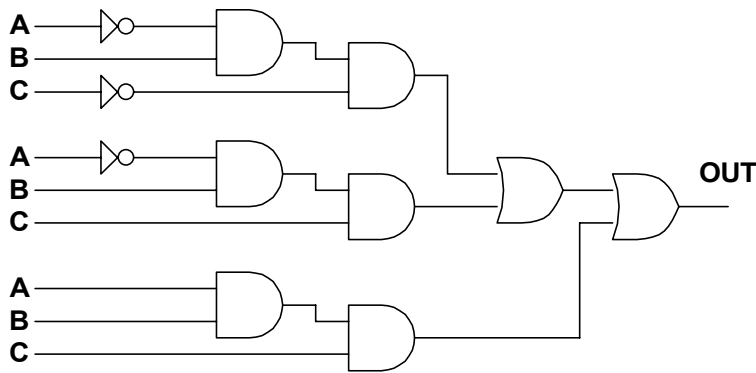
$$case_3 = \overline{A} \cdot B \cdot \overline{C},$$

$$case_4 = \overline{A} \cdot B \cdot C, \text{ and}$$

$$case_8 = A \cdot B \cdot C.$$

Thus the functional relationship between OUT and A, B, & C can be expressed as :  $OUT = \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$  .

The circuit directly derived from this expression is straightforward but requires many components as shown below.

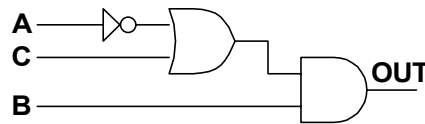


For practical implementation, it is sensible to seek a more efficient expression for accomplishing the objective. This can be done in two ways. One is to use Karnaugh mapping; however, our focus is on **electronic** circuits so we will not discuss that procedure. The second way is to simply use the basic mathematical properties and relations developed above.

Specifically, the expression is:  $OUT = \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot B \cdot C + A \cdot B \cdot C$ . By distribution and use of other relations, we can develop the expression as follows:

$$\begin{aligned}
 OUT &= \bar{A} \cdot B \cdot \bar{C} + (\bar{A} + A) \cdot B \cdot C \\
 OUT &= \bar{A} \cdot B \cdot \bar{C} + B \cdot C, \text{ since } (\bar{A} + A) = 1 \\
 OUT &= \bar{A} \cdot B \cdot \bar{C} + (\bar{A} + 1) \cdot B \cdot C, \text{ since } (\bar{A} + 1) = 1 \\
 OUT &= \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot B \cdot C + B \cdot C \\
 OUT &= \bar{A} \cdot B \cdot (\bar{C} + C) + B \cdot C \\
 OUT &= \bar{A} \cdot B + B \cdot C = (\bar{A} + C) \cdot B
 \end{aligned}$$

Obviously, the last relation is much more compact and efficient than the original, “straightforward” relation given above. In fact the circuit uses very few components:



**Example 2:**

Write the logical relation expressed by the truth table given below and sketch a circuit for implementing the function using **NOT's** and basic two-input **AND's** and **OR's** as necessary.

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

**Solution:** This expression is obviously the same as the previous example with the output **NOT'ed**. However, it will be a useful example of working with “0's” in the output. Actually, the approach shown for **example 1** was only one possibility since the expression could be built from the terms having only “0's” in the output. (In fact, there are three ways to build the expression: consider all the terms with “0's,” all the terms with “1's,” or all terms regardless of “0's” or “1's”. This is reflection of the fact that the binary state tells what it is by telling what it isn't!)

Specifically for this example, the output is low (“0”) for three cases as expressed by rows 3, 4, and 8. Verbally, we can express the relation as  $OUT = 0$  for case<sub>3</sub>, OR case<sub>4</sub>, OR case<sub>8</sub>. Thus the initial expression is:

$$\overline{OUT} = \text{case}_3 + \text{case}_4 + \text{case}_8$$

As before, the functional relationship between OUT and A, B, & C can be

expressed as :

$$\overline{\text{OUT}} = \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$$

Obviously, therefore

$$\text{OUT} = \overline{(\overline{\text{OUT}})} = \overline{(\overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C)} = \overline{(\overline{A} + C)} \cdot B$$

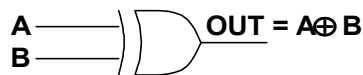
### A "NEW" FUNCTION: THE EXCLUSIVE OR (XOR)

Although the XOR function is built from the basic AND, OR, & NOT functions described above, it is useful enough to deserve its own logic and circuit symbols. The XOR function is defined as:

$$\text{XOR: } A (\text{XOR}) B = A \cdot \overline{B} + \overline{A} \cdot B = A \oplus B$$

**XOR Truth Table**

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0



**XOR Circuit Symbol**

**XOR Applications:** The XOR function plays a central role in the electronic implementation of arithmetic operations. However, it also has many other useful applications and we will examine three of these before introducing arithmetic.

**1. Inequality / Equality Detector:** Inspection of the XOR truth table shows its fundamental behavior, that the result is "1" when A and B are NOT equal. Thus its fundamental nature is to indicate inequality. Since equality / inequality are a binary set, an equality detector is simply **NOT XOR**. This function has the shortened name **XNOR**. Following this realization, we can create the following example:

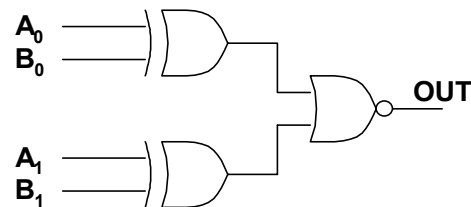
**Example 3:** Develop the logic and circuit to indicate with a "1" when two 2-bit binary numbers (or codes) are equal. The numbers (codes) are A and B and have individual bits identified by  $A_0, A_1, B_0,$  and  $B_1$ .

**Solution:** The two are equal (the same) if, and only if,  $A_0 = B_0$  &  $A_1 = B_1$ . So we can write the relation as follows:

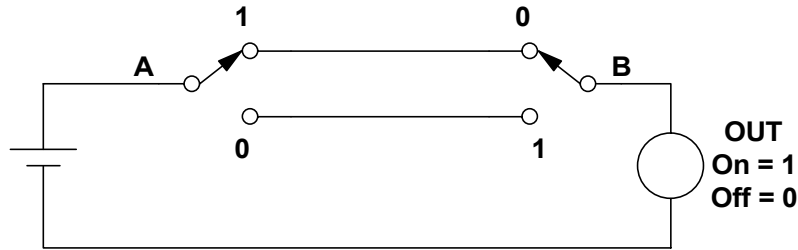
$$\text{OUT} = \overline{(A_0 \oplus B_0)} \cdot \overline{(A_1 \oplus B_1)} = \overline{(A_0 \oplus B_0) + (A_1 \oplus B_1)}$$

Inspection of the logic, and reflection on the objective, reminds us that the goal is to have the output "1" when **neither** pair of bits is **unequal**.

This task can be implemented with the circuit:



**2. Multi-way light switch:** As many of you know, light switches can be placed at multiple doorways to a room with the behavior that lights for the room can be turned on or off at each location. This operation can be connected to the XOR function as shown below with the simple "3-way" switching circuit. ("3-way" is the standard electrician's terminology for the specific switching arrangement. The more general terminology for the switches is "single-pole, double throw" or SPDT.)



A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Note the “weird” labelling of the switch positions. This is suitable since there is no absolute meaning of “0” or “1” insofar as the switch positions are concerned. Thus we are free to choose the labels as we wish as long as they have a binary relationship.)

A	B	OUT
0	0	0
0	1	1
1	1	0
1	0	1

The “relationship table” shows the connection between the switching circuit and the XOR function. To see the more general problem, it is useful to rearrange rows of the table so that adjacent rows are different by change in the state of only one variable as shown (The last two rows are interchanged.)

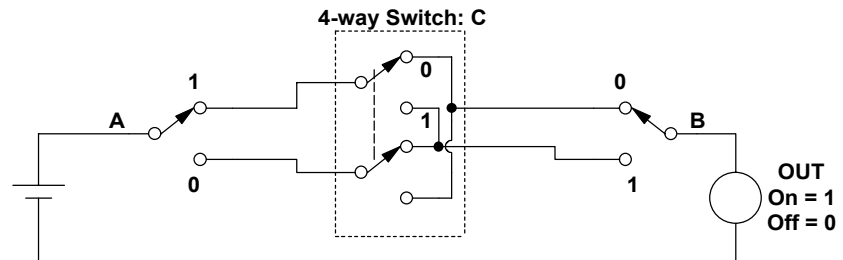
Evident from the rearranged table is that OUT (dependent variable) changes state for each (single) change of an input (independent variable). Also, it shows that a double change leads to no change in the output. Thus the operation of the multi-way light-switching system is to generate a change in the “state” of OUT (on to off, or *vice versa*) when the state of a **single input** is changed (0 to 1, or *vice versa*).

This leads to the following example:

**Example 4:** Extend the 3-way switching example to the case of 3 (or more) inputs.

**Solution:** Actually, this can be done inductively by introducing the third independent variable **C**, and recalling that the result of  $A \oplus B$  is either 0 or 1 and changes with each single change of A or B. Thus, the composite function  $A \oplus B \oplus C$  has the desired property. (Without proof or demonstration, we note that the XOR operation is commutative and associative.)

The inductive approach allows extension of the “multi-way” function to any number of inputs (independent variables). It should also be noted that the appropriate behavior can be implemented with mechanical switches; however, while the “end” two can be the 3-way / SPDT versions, all intermediate switches must be “4-way” / DPDT versions as shown:



The following truth table, and its rearrangement, demonstrate that  $A \oplus B \oplus C$  indeed serves the purpose for the case of 3 independent variables:

A	B	$A \oplus B$	C	$A \oplus B \oplus C$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0
0	0	0	1	1
0	1	1	1	0
1	0	1	1	0
1	1	0	1	1

A	B	C	$A \oplus B \oplus C$
0	0	0	0
0	1	0	1
1	1	0	0
1	0	0	1
1	0	1	0
1	1	1	1
0	1	1	0
0	0	1	1

**3. Programmable NOT generator:** This application will also be helpful in implementation of arithmetic. However, for now it provides us with the first example of “programmability.” Actually, the point is that processors implement their command set by an array of organized logic designed to respond to “operation codes” by performing the appropriate task. This application of the XOR function provides insight into how that is done from the standpoint of logic: it is basically a labeling of the “inputs” descriptive of their functional relation to the results.

Consider the XOR truth table below and the re-labeled version beside it:

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	0

CMD	IN	OUT
0	0	0
0	1	1
1	0	1
1	1	0

From the re-labeling, it is clear that  $OUT = IN$  when  $CMD = 0$  and that  $OUT = \overline{IN}$  when  $CMD = 1$ . Thus the “command” set consists of two operation codes: “0” means “true” and “1” means NOT.

## BINARY ARITHMETIC

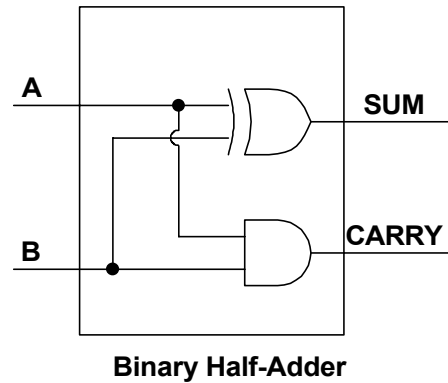
We are now at the point where it is appropriate to introduce binary arithmetic and discuss the electronic circuitry for implementing the basic arithmetic operations. We will consider the four basic operations (add, subtract, multiply, and divide), but will mainly focus on addition and subtraction. The strategy will be to figure out how to do addition and then to convert the other operations to addition.

**Addition:** In the general problem of addition, it is necessary not only to obtain the sum but also to obtain a carry. Also, for addition of columns to the left of the **least significant** position (or column), it is necessary to properly figure in an “incoming” carry from the column to the right. Thus the approach we will take is to develop addition for the least significant position and then to extend it to handle **more significant** positions.

Shown below is the set of relationships appropriate for binary addition:

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

From the table, we can identify the functional relationships necessary to implement the table:  $Sum = A \oplus B$ , and  $Carry = A \cdot B$ . Thus, addition at the least significant column can be implemented with the circuit shown below:



Extending the operations to meet the requirements of the more significant positions (or columns) is fairly straightforward insofar as the sum is concerned:  $A+B+C$  (arithmetic sum) can be calculated by the associativity property as  $(A+B)+C$ . (In this case, C represents the "incoming" carry from the adjacent less significant position.)

However, the main problem is to manage the carries, of which there will be two, in a manner suitable for generating the appropriate carry result. We can examine this problem in a tabular fashion to seek the appropriate functional relationship between the available carry results and the needed overall result.

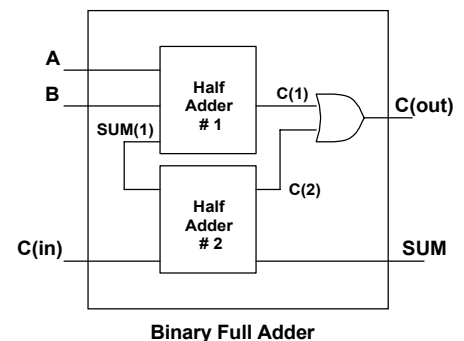
A	B	Sum	Carry(1)	C	Carry(2)	Needed
0	0	0	0	0	0	0
0	1	1	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	1
0	0	0	0	1	0	0
0	1	1	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	1	0	1

Carry(1)	Carry(2)	Needed
0	0	0
0	1	1
1	0	1
1	1	x

x = doesn't matter

Thus, the appropriate relation between the two "intermediate" carries and the necessary final result can be implemented in two ways: with XOR and with OR. It is customary to use the OR function, which yields the final functional circuit for the **Full Adder**.

**Subtraction:** As suggested above, subtraction can be converted to addition as follows:  $A - B = A + (-B)$ . To use this approach, it is necessary to create the negative of a number. The method in general use is that of **complements** and the resulting arithmetic is **complement arithmetic**.



The concept of complements is illustrated with the **number line** based on the decimal system shown below:

Decimal Number Line												
<b>Unsigned</b>	8	9	0	1	2	3	4	5	6	7	8	9
<b>Signed</b>	-2	-1	0	1	2	3	4	5	-4	-3	-2	-1

The idea is based on the point that the numbers recycle to zero after reaching their upper limit (9 in the example) in the same way that the odometer in a car “rolls over” to zero. From this basis, a set of the upper numbers (usually the upper half) are defined as negative since they occur “before zero.” Thus the resulting **signed** number system is created from the initially **unsigned** system by redefining a portion of the numbers.

With this, we have two tasks: one is to demonstrate that the resulting arithmetic actually works for subtraction, and the second is to find a suitable way to create the negative, or complement, from the original number. Demonstration of the arithmetic is fairly straightforward as indicated by the examples:

Positive Result		Negative Result	
5	5	2	2
<u>-2</u>	<u>8</u>	<u>-4</u>	<u>6</u>
3	13	-2	8

In the positive result, the complement-based arithmetic yields “3 with a carry of 1,” meaning that the result is “3” and the carry confirms that the result is positive. In the negative case, the result is “8 with no carry.” From the number line, 8 represents -2 which is the correct result. Also, the absence of a carry (or overflow) confirms that the result is negative.

**Examples of Complement Arithmetic**

In summary, therefore, the complement arithmetic method yields the correct results.

In one sense, creation of a number’s complement is straightforward. For example, from the number line, we see that (the 10’s) **complement of  $N = 10 - N$**  (for  $0 \leq N \leq 4$ ; -5 is undefined). For our use, this approach creates a “Catch-22” in that our objective is to avoid the need for subtraction. Consequently, we must seek other ways to implement the creation of a number’s complement. The solution comes from recollection that our numbers will be binary-based and inspection of a binary-based sequence (serving as the equivalent of the decimal number line). Consider the 3-bit binary (octal) sequence shown in the table:

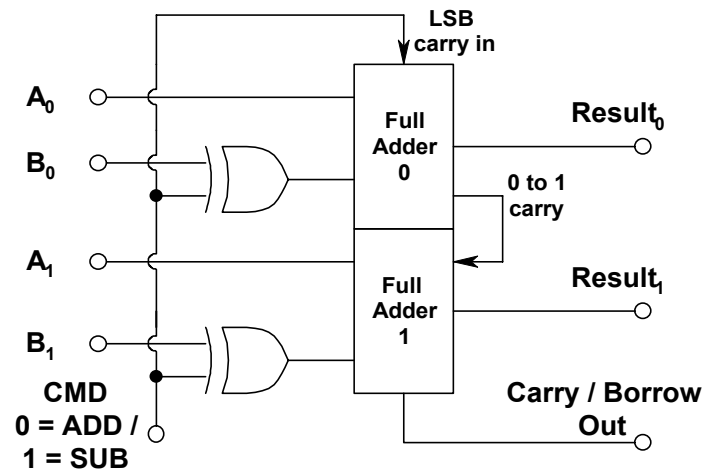
3-bit Binary Sequence (Octal)				
$B_2$	$B_1$	$B_0$	$N(u)$	$N(s)$
0	0	0	0	0
0	0	1	1	1
0	1	0	2	2
0	1	1	3	3
1	0	0	4	-4
1	0	1	5	-3
1	1	0	6	-2
1	1	1	7	-1
0	0	0	0	0
1	0	1	1	1

Based on the systematics of the bit pattern, it is convenient to choose the set of codes with  $B_2$  set to 1 as the negative values. (In this way  $B_2$  serves conveniently as the **sign bit**. In the table,  $N(u)$  indicates the unsigned value and  $N(s)$  indicates the signed value.) From the sequence of codes, we see that  **$-N = N(\text{bitwise NOT}) + 1$** . For example, for  $N = 2$  (010) the bitwise NOT is 101; adding 001 gives 110 which is unsigned 6 and signed -2. (This even works for the negative of a negative!)

This method of **2’s complement** generation is important since it can be implemented with circuitry we already have available: NOT generation and addition (of 1).

**Example 5:** Combine the above developments to design a circuit to ADD or SUBTRACT two 2-bit binary numbers. Individual bits of the numbers are  $A_0, A_1, B_0,$  and  $B_1$  for A and B. The “0” and “1” subscripts indicate respectively the less significant bits (LSB) and more significant bits (MSB) of A and B. The resulting design should have the command set **1 = SUB** and **0 = ADD**.

**Solution:** We will use two full adders—one for each bit—and connect the output carry from the LSB to the input carry of the MSB. In addition, to implement subtraction via the complement method, we will use XOR's as “programmable” NOT generators for both bits of B. Finally, the “1” needed for forming the 2's complement of B will be obtained by making the incoming carry of the LSB = 1 on SUB (and 0 on ADD).



**Multiplication:** The most direct way to multiply by adding is to repeatedly add. For example, to multiply 8 times 7, 8 can be added 7 times. This method requires addition and also counting; that is, the number of additions need to be counted. As well, the method of repeated addition requires holding the intermediate results.

The more efficient method of “long multiplication” taught in schools can be described as “shift and add.” That is, multiplication by each digit of the multiplier yields a result to be shifted to the left the same number of positions as the multiplier digit. Left-shifting creates a multiplication by the base of the number system (usually 10). Instead of counting, this method requires the shifting operation; also, it is necessary to hold intermediate results.

Since we have not yet developed methods for counting, shifting, or holding intermediate results, we will not consider multiplication further. However, note that the only *arithmetic* operation used in multiplication is addition.

**Division:** Division can be implemented by repeated subtraction combined with counting and a test of whether or not the remainder is less or greater than the divisor. Even the “long division” method learned in grammar school uses the operations of compare, shift, and subtract. Because of the additional comparison operation, division requires the most steps of all arithmetic operations and is thereby the slowest. Note also that the only arithmetic operation is that of subtraction; thus, all the methods preserve our basic arithmetic strategy of converting each to addition.

## NUMBERS, NUMBER SYSTEMS, THE STRUCTURE OF A NUMBER, AND INTER-BASE CONVERSION

In many places above, we assumed conscious understanding of the structure of numbers. Nevertheless, it makes sense now to review basic concepts behind the structure of numbers. The first concept is simply that of the “number of things.” From that notion, which can be visualized in terms of a number of dots such as: ., .., ..., ....., etc., we next imagine assigning a symbol to represent specific “numbers.”

This introduces the question of how many symbols to select as the basic set, with the answer being basically a matter of convenience. Our culture is built on a set of 10 symbols (the digits 0 through 9) representing a

“number of dots” so basic to our understanding that it is difficult to establish precise definitions. (Notice that there is nothing unique about 0-9 giving them fundamental value as the set of symbols; the point is that we need symbols, and these are as good as any although we could have chosen any set of 10 distinct symbols.) However, the advent of computers, and their fundamental binary-based nature, led to broadening the use of number systems and symbol sets. Three binary-based sets are binary, octal, and hexadecimal. Of these, hexadecimal is useful for illustration as it requires extension of our common symbol set from 10 to 16. This is customarily done by using the first 6 letters of the alphabet, A - F, to indicate the “numbers” 10 - 15, respectively. Notice, however, that we also have common experience with other number bases: 7 days per week; 60 seconds per minute, 24 hours per day, etc.

The need for a multi-digit number arises when we need to symbolically represent a “number of things” too large for a single symbol. Thus, we need a systematic method for constructing the multi-symbol representation. Note that all systems are not equal: for example the Roman Numeral system is not conducive to arithmetic. The system in common use assigns a “weighting” factor to a symbol according to its right-to-left position in the number. For example, in the decimal number 234, the “2” is more important than the “4” because of its position, despite being “half as big” in a literal sense.

From this background, we can now understand the **structure of a number** and, from that understanding, we can develop methods for converting a number from one “base” to another. Specifically, the problem is that 234 has a different meaning depending on whether it is **octal**, **decimal**, or **hexadecimal**. The first step is to recognize that the “base” of a number system specifies the number of distinct symbols, or **the amounts which can be represented by a single symbol**. The next step is to recognize that the positions are weighted on a right-to-left basis with each position being more important by a factor of the base than the one immediately to its right. This means that the weighting factor is  $B^I$ , where  $I$  is the position from the right (starting from 0).

**In summary**, the decimal number  $234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 = 200 + 30 + 4$ . In contrast, the octal number  $234 = 2 \times 8^2 + 3 \times 8^1 + 4 \times 8^0 (= 128 + 24 + 4 = 156 \text{ decimal})$ , and the hexadecimal number  $234 = 2 \times 16^2 + 3 \times 16^1 + 4 \times 16^0 (= 512 + 48 + 4 = 564 \text{ decimal})$ .

## SUMMARY

This introduction to digital electronics provides the basic set of operations necessary for all functions of **combinatorial logic**. These include encoding / decoding and the basic arithmetic operations of addition and subtraction. However, in the case of arithmetic, and in many other tasks easy to envision, it is necessary to hold intermediate values—that is, it is necessary to **remember** intermediate results. Simple combinatorial logic is incapable of implementing memory. Nevertheless, the basic operations of AND, OR, and NOT still provide the means of implementing **memory** as well as other functions such as counting and shifting.

These are topics in **sequential logic** on which the next section of material will focus.