Fundamentals of Digital Computing

Part-2

3

BOOLEAN ALGEBRA AND LOGIC GATES

Unit Structure

1. OBJECTIVES:

After completing this chapter, you will be able to:

- ❖ Understand the concept of Boolean Logic
- ❖ Learn the concept of Logic gates with the help of Diagrams.
- ❖ Understanding the Universal Gates and their circuit implications.
- ❖ Learn about Exclusive OR & NOR gates.
- ❖ Understand the Boolean algebra and laws of Boolean Algebra for use in implementation.

2. BOOLEAN LOGIC:

Boolean logic is named after a 19th Century English clergyman, George Boole, who was also a keen amateur

mathematician and formalised the mathematics of a logic based on a two-valued (TRUE and FALSE) system.

In Boolean logic any variable can have one of only two possible values, TRUE or FALSE. In some systems these may be called HIGH and LOW or perhaps ONE and ZERO, but whatever the names there are only two possible values. There is no such thing as "in between" or "it has no value". If a variable is not TRUE, then it must be FALSE.

3.2 LOGIC GATES:

Practical devices which obey the laws of Boolean logic can be made in a variety of different forms. In the earlier part of the Century, relays were used extensively for simple logic systems such as the controllers for lifts (elevators) and traffic lights. Victorian signal boxes used mechanical levers and rods for operating points and signals which had logical interlocks, and in some hazardous environments where the use of electricity is avoided, fluid logic circuits are used. In this course we are interested in electronic logic based on the modern, silicon, integrated-circuit technology.

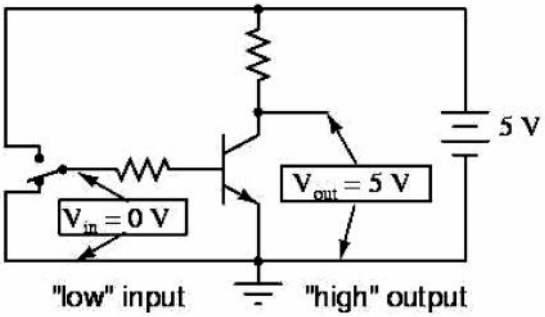
In electronic logic, the value of a logic signal on an input or output is usually defined by the voltage. If the voltage is above a certain value, it is a logic ONE. If it is below that value it is a logic ZERO. Circuits are produced (by the million!) which have a (Boolean) output value, or values, which are directly determined by the (Boolean) values on the inputs. In these notes a logical value of ONE or TRUE will often be written as ’1’ and a value which is ZERO or FALSE written as ’0’.

3.2.1 NOT gate

While the binary numeration system is an interesting mathematical abstraction, we haven’t yet seen its practical application to electronics. This chapter is devoted to just that: practically applying the concept of binary bits to circuits. What makes binary numeration so important to the application of digital electronics is the ease in which bits may be represented in physical terms. Because a binary bit can only have one of two different values, either 0 or 1, any physical medium capable of switching between two saturated states may be used to represent a bit. Consequently, any physical system capable of representing binary bits is able to represent numerical quantities, and potentially has the ability to manipulate those numbers. This is the basic concept underlying digital computing.

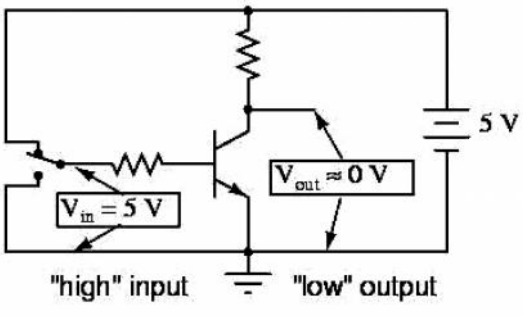
Electronic circuits are physical systems that lend themselves well to the representation of binary numbers. Transistors, when operated at their bias limits, may be in one of two different states: either cutoff (no controlled current) or saturation (maximum controlled current). If a transistor circuit is designed to maximize the probability of falling into either one of these states (and not operating in the linear, or *active*, mode), it can serve as a physical representation of a binary bit. A voltage signal measured at the output of such a circuit may also serve as a representation of a single bit, a low voltage representing a binary "0" and a (relatively) high voltage representing a binary "1." Note the following transistor circuit:

Transistor in cutoff



0 V = "low" logic level (0)
5 V = "high" logic level (1)

Transistor in saturation



0 V = "low" logic level (0)
5 V = "high" logic level (1)

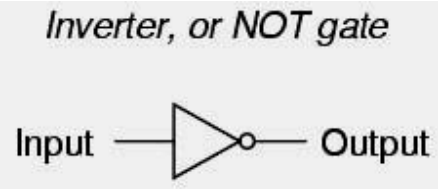
In this circuit, the transistor is in a state of saturation by virtue of the applied input voltage (5 volts) through the two-position switch. Because it's saturated, the transistor drops very little voltage between collector and emitter, resulting in an output voltage of (practically) 0 volts. If we were using this circuit to represent binary bits, we would say that the input signal is a binary "1" and that the

output signal is a binary "0." Any voltage close to full supply voltage (measured in reference to ground, of course) is considered a "1" and a lack of voltage is considered a "0." Alternative terms for these voltage levels are *high* (same as a binary "1") and *low* (same as a binary "0"). A general term for the representation of a binary bit by a circuit voltage is *logic level*.

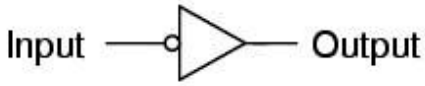
Moving the switch to the other position, we apply a binary "0" to the input and receive a binary "1" at the output:

What we've created here with a single transistor is a circuit generally known as a *logic gate*, or simply *gate*. A gate is a Special type of amplifier circuit designed to accept and generate voltage signals corresponding to binary 1's and 0's. As such, gates are not intended to be used for amplifying analog signals (voltage signals *between* 0 and full voltage). Used together, multiple gates may be applied to the task of binary number storage (memory circuits) or manipulation (computing circuits), each gate's output representing one bit of a multi-bit binary number. Just how this is done is a subject for a later chapter. Right now it is important to focus on the operation of individual gates.

The gate shown here with the single transistor is known as an *inverter*, or NOT gate, because it outputs the exact opposite digital signal as what is input. For convenience, gate circuits are generally represented by their own symbols rather than by their constituent transistors and resistors. The following is the symbol for an inverter:

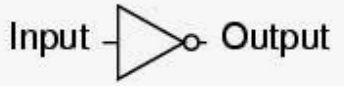


An alternative symbol for an inverter is shown here:



One common way to express the particular function of a gate circuit is called a *truth table*. Truth tables show all combinations of input conditions in terms of logic level states (either "high" or "low," "1" or "0," for each input terminal of the gate), along with the corresponding output logic level, either "high" or "low." For the inverter, or NOT, circuit just illustrated, the truth table is very simple indeed:

NOT gate truth table



Input	Output
0	1
1	0

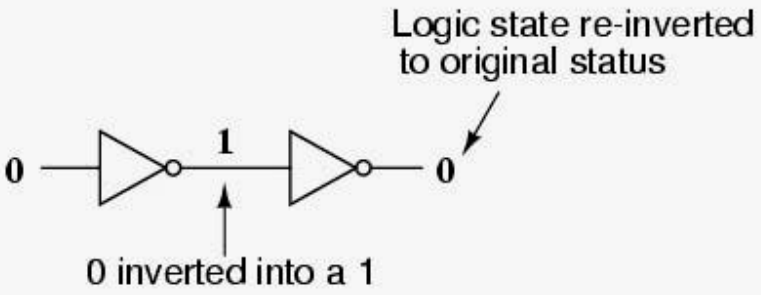
Equation for NOT gate

$$Y = \overline{A}$$

3.2.2 The "buffer" gate

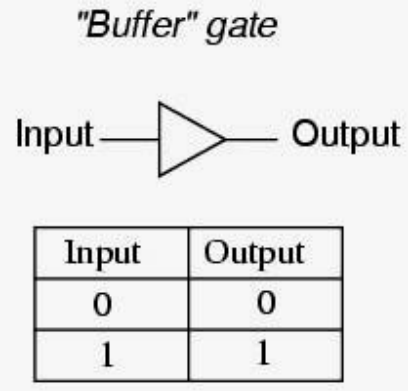
If we were to connect two inverter gates together so that the output of one fed into the input of another, the two inversion functions would "cancel" each other out so that there would be no inversion from input to final output:

Double inversion



While this may seem like a pointless thing to do, it does have practical application. Remember that gate circuits are signal amplifiers, regardless of what logic function they may perform. A weak signal source (one that is not capable of sourcing or sinking very much current to a load) may be boosted by means of two inverters like the pair shown in the previous illustration. The logic level is unchanged, but the full current-sourcing or -sinking capabilities of the final inverter are available to drive a load resistance if needed.

For this purpose, a special logic gate called a buffer is manufactured to perform the same function as two inverters. Its symbol is simply a triangle, with no inverting "bubble" on the output terminal:



Input	Output
0	0
1	1

Equation for Buffer gate

Y = A

Inverters and buffers exhaust the possibilities for single-input gate circuits. What more can be done with a single logic signal but to buffer it or invert it? To explore more logic gate possibilities, we must add more input terminals to the circuit(s).

Adding more input terminals to a logic gate increases the number of input state possibilities. With a single-input gate such as the inverter or buffer, there can only be two possible input states: either the input is "high" (1) or it is "low" (0). As was mentioned previously in this chapter, a two input gate has *four* possibilities (00, 01, 10, and 11). A three-input gate has *eight* possibilities (000, 001, 010, 011, 100, 101, 110, and 111) for input states. The number of possible input states is equal to two to the power of the number of inputs:

$$\text{Number of possible input states} = 2^n$$

Where,

n = Number of inputs

This increase in the number of possible input states obviously allows for more complex gate behavior. Now, instead of merely inverting or amplifying (buffering) a single "high" or "low" logic level, the output of the gate will be determined by whatever *combination* of 1's and 0's is present at the input terminals.

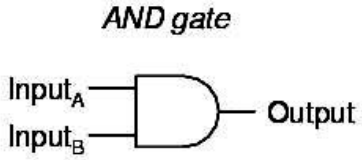
Since so many combinations are possible with just a few input terminals, there are many different types of multiple-input gates, unlike single-input gates which can only be inverters or buffers.

3.2.3 The AND gate

One of the easiest multiple-input gates to understand is the AND gate, so-called because the output of this gate will be "high" (1) if and only if all inputs (first input and the second input and . . .) are "high" (1). If any input(s) are "low" (0), the output is guaranteed to be in a "low" state as well.

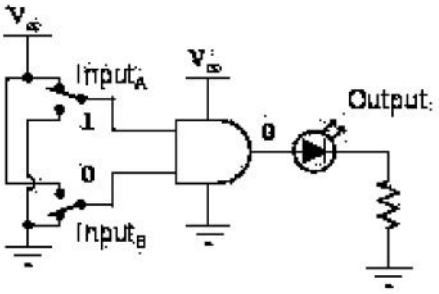
In case you might have been wondering, AND gates are made with more than three inputs, but this is less common than the simple two-input variety.

The logic diagram and the truth table of AND gate are shown below:

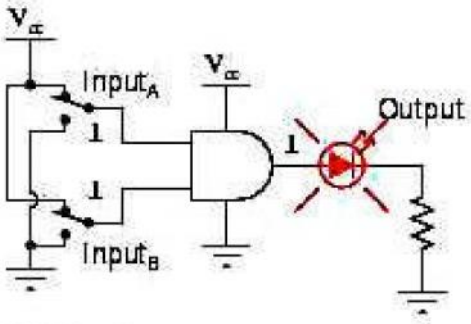


A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

What this truth table means in practical terms is shown in the following sequence of illustrations, with the 2-input AND gate subjected to all possibilities of input logic levels. An LED (Light-Emitting Diode) provides visual indication of the output logic level:



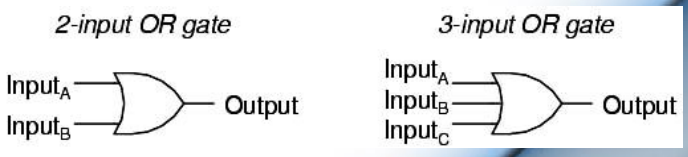
Input$_A$ = **1**
Input$_B$ = **0**
Output = **0** *(no light)*



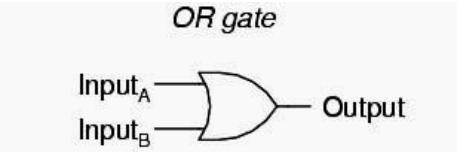
Input$_A$ = **1**
Input$_B$ = **1**
Output = **1** *(light!)*

3.2.4 The OR gate

Our next gate to investigate is the OR gate, so-called because the output of this gate will be "high" (1) if any of the inputs (first input or the second input or . . .) are "high" (1). The output of an OR gate goes "low" (0) if and only if all inputs are "low" (0).

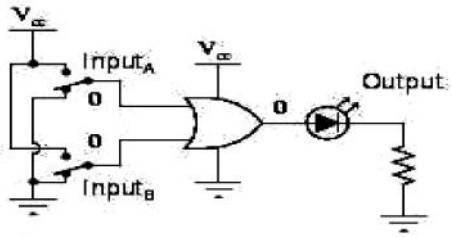


The logic diagram and the truth table of OR gate are shown below:

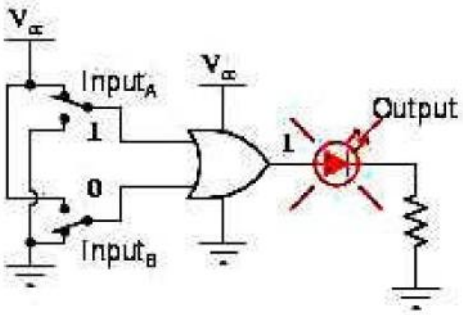


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

The following sequence of illustrations demonstrates the OR gate's function, with the 2-inputs experiencing all possible logic levels. An LED (Light-Emitting Diode) provides visual indication of the gate's output logic level:



Input$_A$ = 0
Input$_B$ = 0
Output = 0 *(no light)*



Input$_A$ = 1
Input$_B$ = 0
Output = 1 *(light)*

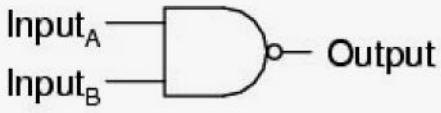
3.3 UNIVERSAL GATES

NOT, AND and OR gates are alled as baisc gates. The NAND and NOR gates are called as Universal gates as any gate can be derived using these gates.

3.3.1 NAND gate:

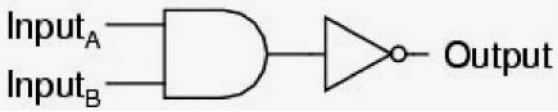
A variation on the idea of the AND gate is called the NAND gate. The word "NAND" is a verbal contraction of the words NOT and AND. Essentially, a NAND gate behaves the same as an AND gate with a NOT (inverter) gate connected to the output terminal. To symbolize this output signal inversion, the NAND gate symbol has a bubble on the output line. The truth table for a NAND gate is as one might expect, exactly opposite as that of an AND gate:

NAND gate

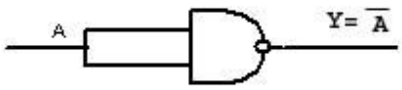


A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

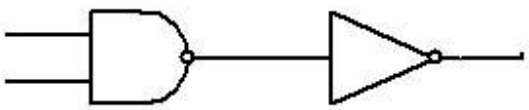
Equivalent gate circuit



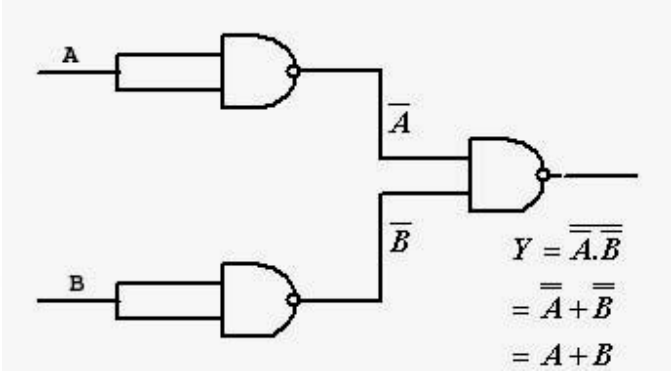
NOT gate from NAND gate:



AND gate from NAND gate:



OR gate from NAND gate:

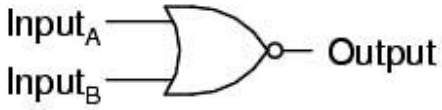


3.3.2 NOR gate:

As you might have guessed, the NOR gate is an OR gate with its output inverted, just like a NAND gate is an AND gate with an inverted output.

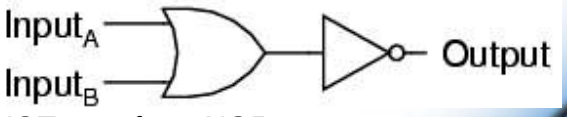
The NOR gate has output high when both of its inputs are low. The symbolic diagram and the truth table of NOR gate is shown below:

NOR gate

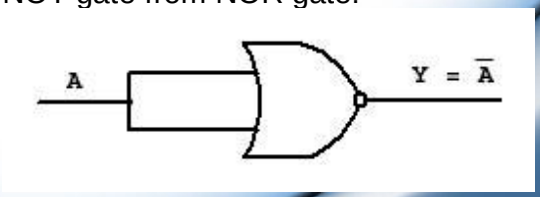


A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

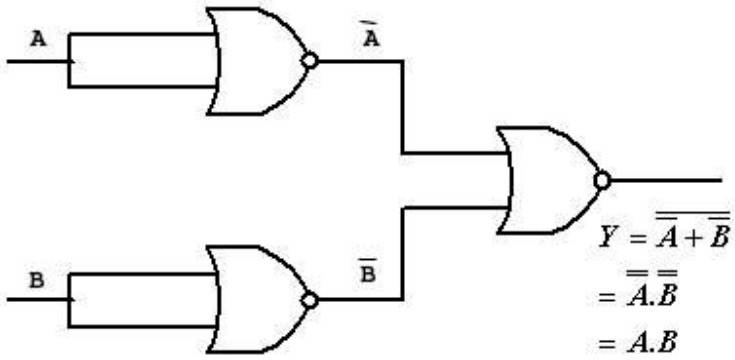
Equivalent gate circuit



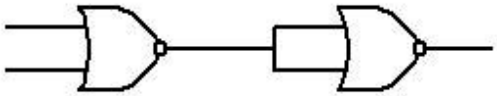
NOT gate from NOR gate:



AND gate from NOR gate:



OR gate from NOR gate:



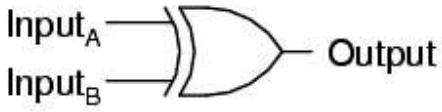
So we have seen that all the basic gates can be constructed using NAND and NOR gates and hence they are called Universal gates.

3.4 OTHER GATES

3.4.1 The Exclusive-OR gate

Exclusive-OR gates output a "high" (1) logic level if the inputs are at different logic levels, either 0 and 1 or 1 and 0. Conversely, they output a "low" (0) logic level if the inputs are at the same logic levels. The Exclusive-OR (sometimes called XOR) gate has both a symbol and a truth table pattern that is unique:

Exclusive-OR gate

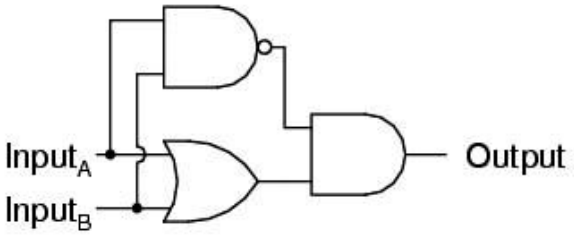


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

There are equivalent circuits for an Exclusive-OR gate made up of AND, OR, and NOT gates, just as there were for NAND, NOR, and the negative-input gates. A rather direct approach to

simulating an Exclusive-OR gate is to start with a regular OR gate, then add additional gates to inhibit the output from going "high" (1) when both inputs are "high" (1):

Exclusive-OR equivalent circuit

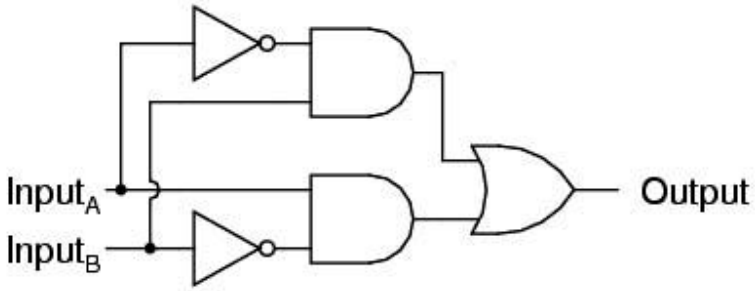


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

In this circuit, the final AND gate acts as a buffer for the output of the OR gate whenever the NAND gate's output is high, which it is for the first three input state combinations (00, 01, and 10). However, when both inputs are "high" (1), the NAND gate outputs a "low" (0) logic level, which forces the final AND gate to produce a "low" (0) output.

Another equivalent circuit for the Exclusive-OR gate uses a strategy of two AND gates with inverters, set up to generate "high" (1) outputs for input conditions 01 and 10. A final OR gate then allows either of the AND gates' "high" outputs to create a final "high" output:

Exclusive-OR equivalent circuit

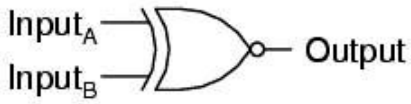


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

3.4.2 The Exclusive-NOR gate

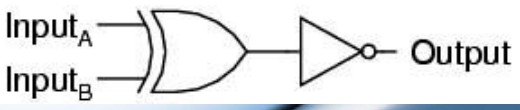
Finally, our last gate for analysis is the Exclusive-NOR gate, otherwise known as the XNOR gate. It is equivalent to an Exclusive-OR gate with an inverted output. The truth table for this gate is exactly opposite as for the Exclusive-OR gate:

Exclusive-NOR gate



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

Equivalent gate circuit



3.5 BOOLEAN ALGEBRA

The most obvious way to simplify Boolean expressions is to manipulate them in the same way as normal algebraic expressions are manipulated. With regards to logic relations in digital forms, a set of rules for symbolic manipulation is needed in order to solve for the unknowns.

A set of rules formulated by the English mathematician *George Boole* describe certain propositions whose outcome would be either *true or false*. With regard to digital logic, these rules are used to describe circuits whose state can be either, *1 (true) or 0 (false)*. In order to fully understand this, the relation between the AND gate, OR gate and NOT gate operations should be appreciated. A number of rules can be derived from these relations as the following Table demonstrates.

- **P1: X = 0 or X = 1**
- **P2: 0 . 0 = 0**
- **P3: 1 + 1 = 1**
- **P4: 0 + 0 = 0**
- **P5: 1 . 1 = 1**
- **P6: 1 . 0 = 0 . 1 = 0**
- **P7: 1 + 0 = 0 + 1 = 1**

3.6 LAWS OF BOOLEAN ALGEBRA

The following table shows the basic Boolean laws. Note that every law has two expressions, (a) and (b). This is known as *duality*. These are obtained by changing every AND(.) to OR(+), every OR(+) to AND(.) and all 1's to 0's and vice-versa. It has become conventional to drop the **.** (AND symbol) i.e. A.B is written as AB.

T1 : Commutative Law

(a) A + B = B + A
(b) A B = B A

T2 : Associate Law

(a) *(A + B) + C = A + (B + C)*
(b) *(A B) C = A (B C)*

T3 : Distributive Law

(a) *A (B + C) = A B + A C*
(b) *A + (B C) = (A + B) (A + C)*

T4 : Identity Law

(a) A + A = A
(b) A A = A

T5 :

(a) $AB + A\bar{B} = A$

(b) $(A+B)(A+\bar{B}) = A$

T6 : Redundance Law

(a) A + A B = A

(b) A (A + B) = A

T7 :

(a) 0 + A = A

(b) 0 A = 0

T8 :

(a) *1 + A = 1*

(b) *1 A = A*

T9 :

(a) $\bar{A} + A = 1$

(b) $\bar{A}\,A = 0$

T10 :

(a) $A + \bar{A}\,B = A + B$

(b) $A(\bar{A} + B) = AB$

T11 : De Morgan's Theorem

(a) $\overline{(A+B)} = \bar{A}\,\bar{B}$

(b) $\overline{(AB)} = \bar{A} + \bar{B}$

Examples:

ab + ab’ + a’b = a(b+b’) + a’b

= a•1 + a’b By P5

= a + a’b By

= a + a’b + 0

= a + a’b + aa’

= a + b(a + a’)

= a + b•1

= a + b

(a’b + a’b’ + b’)‘ = (a’(b+b’) + b’)’

= (a’ + b’)’

= ((ab)’)’

= ab

b(a+c) + ab’ + bc’ + c = ba + bc + ab’ + bc’ + c

= a(b+b’) + b(c + c’) + c

= a•1 + b•1 + c

= a + b + c

7. QUESTIONS:

1. What are basic gates? Explain.
2. What are universal gates? Why are they so called?
3. Explain XOR and XNOR gates.
4. Construct the following gates from universal (both NAND and NOR) gates:
    a. AND GATE
    b. OR GATE
    c. NOT GATE
    d. XOR GATE
    e. XNOR GATE
5. State and prove De' Morgans Laws.
6. State and prove the laws of Boolean algebra.
7. Prove the following using Laws of Boolean Algebra:

i. $(A+\overline{AB})(A+\overline{AB})(CD+\overline{CDA}+C\overline{D}+\overline{C}DA)=A$

ii. $\overline{XYZ}+\overline{X}Y\overline{Z}+X\overline{YZ}+XY\overline{Z}=X\overline{Y}+\overline{Z}$

iii. $\left(\overline{A+\overline{BC}}\right)\left(A\overline{B}+\overline{ABC}\right)=\overline{A}BC$

iv. $(AB+ABC)A\overline{B}C=0$

4

CANONICAL FORMS AND KARNAUGH MAPS

Unit Structure

1. Objectives
2. Canonical Forms
3. Karnaugh Maps
4. Simplifying Boolean Expressions Using Karnaugh Maps
5. Maxterms and Minterms
6. Quine Mcclusky Method
7. Questions
8. Further Reading

1. OBJECTIVES

After completing this chapter, you will be able to:

- Learn about the CANONICAL FORMS.
- Understand the building and working of KARNAUGH MAP.
- Using the KARNAUGH MAP for solving the Boolean expression.
- Understanding the concept of Maxterms and Minterms
- Learn the Quine McClusky Method for finding a minimum-cost sum-of-products.

2. CANONICAL FORMS

Since there are a finite number of boolean functions of *n* input variables, yet an infinite number of possible logic expressions you can construct with those *n* input values, clearly there are an infinite number of logic expressions that are equivalent (i.e., they produce the same result given the same inputs). To help eliminate possible confusion, logic designers generally specify a boolean function using a *canonical*, or standardized, form. For any given boolean function there exists a unique canonical form. This eliminates some confusion when dealing with boolean functions.

Actually, there are several different canonical forms. We will discuss only two here and employ only the first of the two. The first is the so-called *sum of minterms* and the second is the *product of maxterms.* Using the duality principle, it is very easy to convert between these two.

A *term* is a variable or a product (logical AND) of several different literals. For example, if you have two variables, A and B, there are eight possible terms: A, B, A', B', A'B', A'B, AB', and AB. For three variables we have 26 different terms: A, B, C, A', B', C', A'B', A'B, AB', AB, A'C', A'C, AC', AC, B'C', B'C, BC', BC, A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. As you can see, as the number of variables increases, the number of terms increases dramatically. A *minterm* is a product containing exactly *n* literals. For example, the minterms for two variables are A'B', AB', A'B, and AB. Likewise, the minterms for three variables A, B, and C are A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. In general, there are 2^n minterms for *n* variables. The set of possible minterms is very easy to generate since they correspond to the sequence of binary numbers:

Binary Equivalent (CBA)	Minterm
000	A'B'C'
001	AB'C'
010	A'BC'
011	ABC'
100	A'B'C
101	AB'C
110	A'BC
111	ABC

4.2 KARNAUGH MAPS

A **Karnaugh map** comprises a box for every line in the truth table; the binary value for each box is the binary value of the input terms in the corresponding table row. Unlike a truth table, in which the input values typically follow a standard binary sequence (00, 01, 10, 11), the Karnaugh map's input values must be ordered such that the values for adjacent columns vary by only a single bit, for example, 00, 01, 11, and 10. This ordering is known as a *Gray code*.

We use a **Karnaugh map** to obtain the simplest possible Boolean expression that describes a truth table.

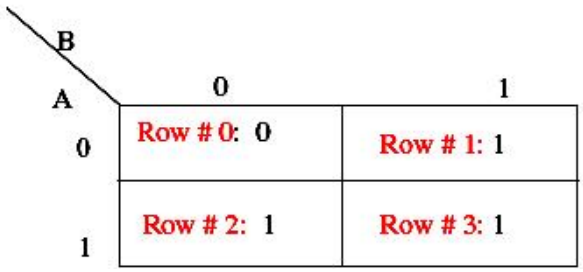
Each row in the table (or minterm) is equivalent to a a cell on the Karnaugh Map.

Example #1:

Here is a two-input truth table for a digital circuit:

Row	Inputs		Output
	A	B	F
Row # 0	0	0	0
Row # 1	0	1	1
Row # 2	1	0	1
Row # 3	1	1	1

The corresponding K-map is



Example #2:

Here is a three-input truth table for a digital

Row	Inputs			Output
	A	B	C	F
Row # 0	0	0	0	0
Row # 1	0	0	1	1
Row # 2	0	1	0	1
Row # 3	0	1	1	1
Row # 4	1	0	0	1
Row # 5	1	0	1	1
Row # 6	1	1	0	0
Row # 7	1	1	1	1

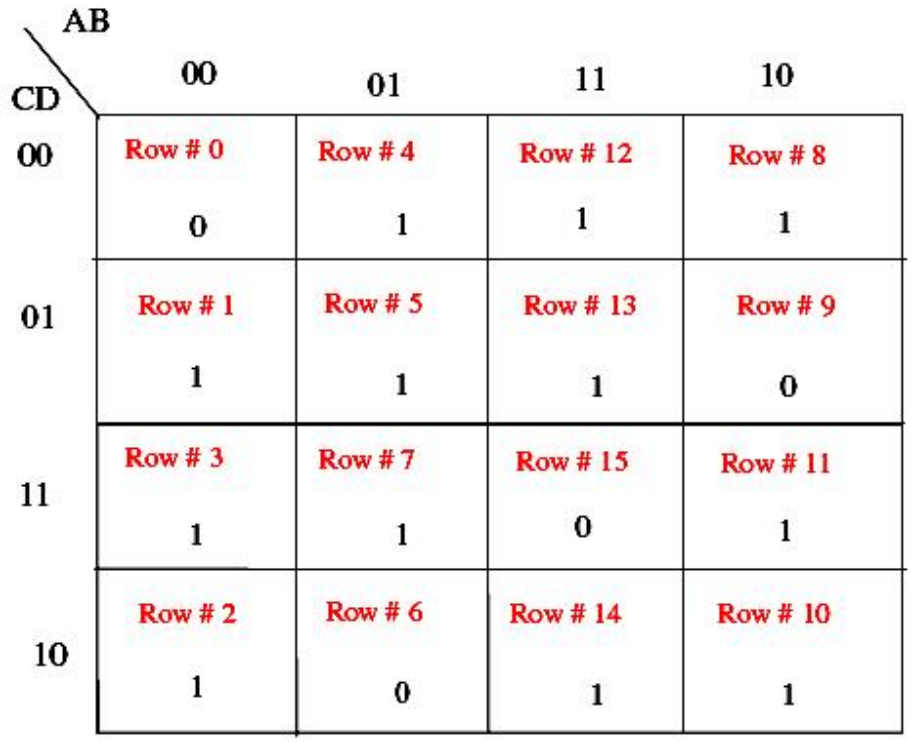
The corresponding K-map is

C \ AB	00	01	11	10
0	Row # 0 0	Row # 2 1	Row # 6 0	Row # 4 1
1	Row # 1 1	Row # 3 1	Row # 7 1	Row # 5 1

Example #3:
Here is a four-input truth table for a digital circuit:

Row	Inputs				Output
	A	B	C	D	F
Row # 0	0	0	0	0	0
Row # 1	0	0	0	1	1
Row # 2	0	0	1	0	1
Row # 3	0	0	1	1	1
Row # 4	0	1	0	0	1
Row # 5	0	1	0	1	1
Row # 6	0	1	1	0	0
Row # 7	0	1	1	1	1
Row # 8	1	0	0	0	1
Row # 9	1	0	0	1	0
Row # 10	1	0	1	0	1
Row # 11	1	0	1	1	1
Row # 12	1	1	0	0	1
Row # 13	1	1	0	1	1
Row # 14	1	1	1	0	1
Row # 15	1	1	1	1	0

The corresponding K-map is



4.3 SIMPLIFYING BOOLEAN EXPRESSIONS USING KARNAUGH MAP

To simplify the resulting Boolean expression using a Karnaugh map adjacent cells containing one are looped together. This step eliminated any terms of the form *AA* .

Adjacent cells means:

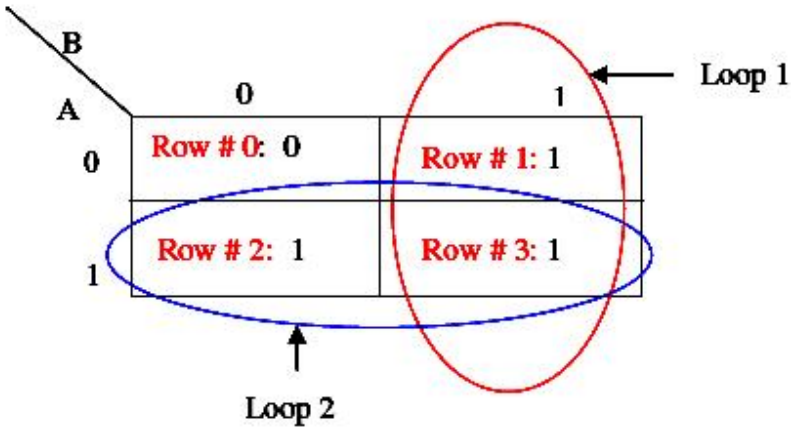
1. Cells that are side by side in the horizontal and vertical directions (but not diagonal).
2. For a map row: the leftmost cell and the rightmost cell.
3. For a map column: the topmost cell and the bottom most cell.
4. For a 4 variable map: cells occupying the four corners of the map.

Cells may only be looped together in twos, fours, or eights. As few groups as possible must be formed. Groups may overlap one another and may contain only one cell.

The larger the number of 1s looped together in a group the simpler is the product term that the group represents.

Example #1:

Simplifying the corresponding K-map of a two-input truth table for a digital circuit:



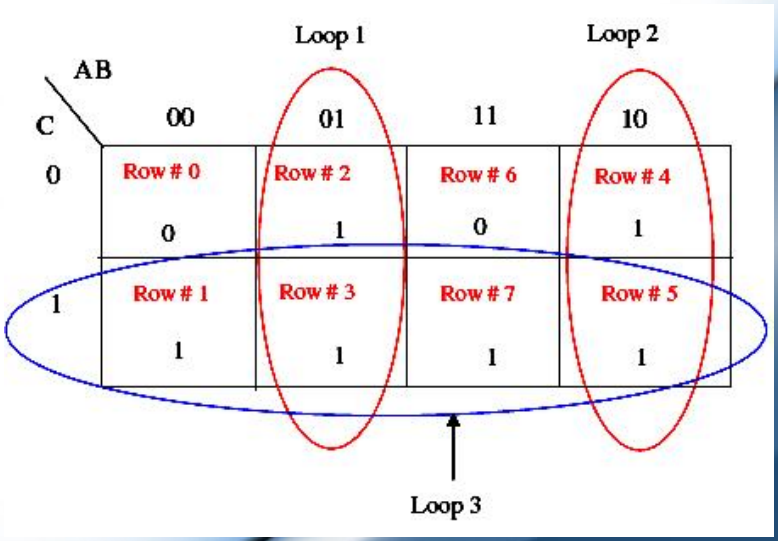
In Loop 1 the variable A has both logic 0 and logic 1 values in the same loop. B has a value of 1. Hence minterm equation is: $F = B$.

In Loop 2 Variable B has both logic 0 and 1 values in the same loop. A = 1, hence minterm equation is: $F = A$.

The overall Boolean expression for F is therefore: $F = A + B$

Example #2:

Simplifying the corresponding K-map of a three-input truth table for a digital circuit:



In Loop 1 the variable C has both logic 0 and logic 1 values in the same loop. A has a value of 0 and B has a logic value of 1. Hence minterm equation is: *F*

In Loop 2 the variable C has both logic 0 and 1 values in the same loop. A = 1 and B = 0, hence minterm equation is: *F*

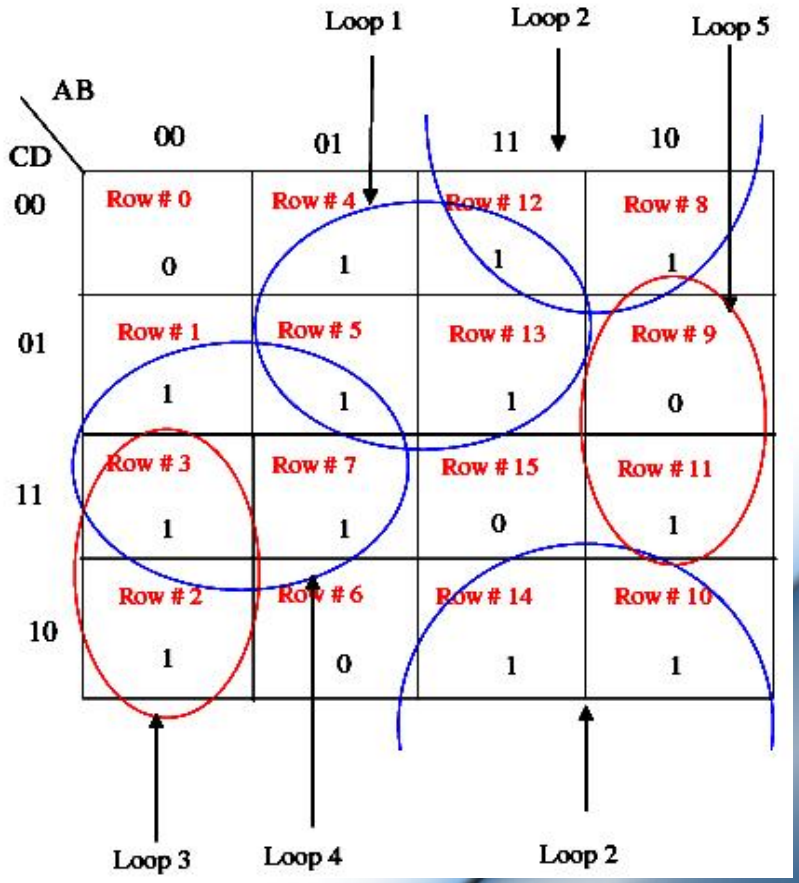
.

In Loop 3 the two variables A and B both have logic 0 and logic 1 values in the same loop. C has a value of 1. Hence minterm equation is: *F* *C*.

The overall Boolean expression for F is therefore: *F* $+A\bar{B}+C$

Example #3:

Simplifying the corresponding K-map of a four-input truth table for a digital circuit:



In Loop 1 the two variables A and D both have logic 0 and logic 1 values in the same loop. C has a value of 0 and B has a value of 1. Hence minterm equation is: *F*

In Loop 2 the two variables B and C both have logic 0 and logic 1 values in the same loop. A has a value of 1 and D has a value of 0. Hence minterm equation is: *F* .

In Loop 3 the variable D has logic 0 and logic 1 values in the same loop. A and B both have a value of 0 and C has a value of 1. Hence minterm equation is: *F*
.

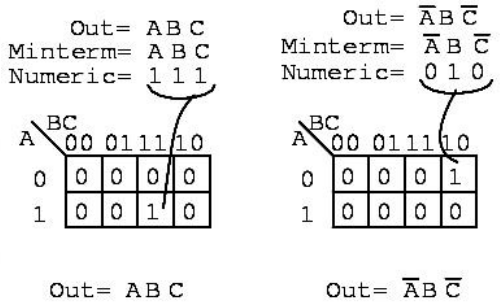
In Loop 4 the two variables B and C both have logic 0 and logic 1 values in the same loop. A has a value of 0 and D has a value of 1. Hence minterm equation is: *F* .

In Loop 5 the variable C has logic 0 and logic 1 values in the same loop. A and D both have s a value of 1 and B has a value of 0. Hence minterm equation is: *F* .

The overall Boolean expression for F is therefore: *F* $+\bar{A}BC$ *D*

4.4 MAXTERMS AND MINTERMS

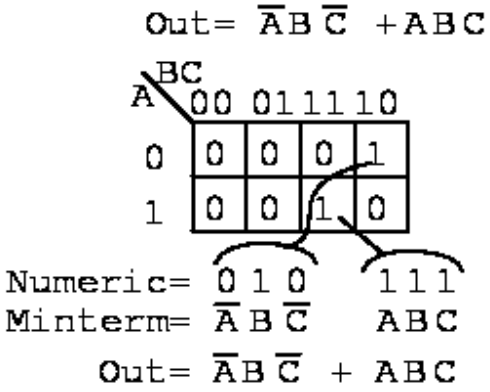
Sum-Of-Product (SOP) and Product-Of-Sums solution (POS):



A *minterm* is a Boolean expression resulting in **1** for the output of a single cell, and **0**s for all other cells in a Karnaugh map, or truth table. If a minterm has a single **1** and the remaining cells as **0**s, it would appear to cover a minimum area of **1**s. The illustration above left shows the minterm **ABC**, a single product term, as a single **1** in a map that is otherwise **0**s. We have not shown the **0**s in our Karnaugh maps up to this point, as it is customary to omit them unless specifically needed. Another minterm **A'BC'** is shown above right. The point to review is that the address of the cell corresponds directly to the minterm being mapped. That is, the cell **111** corresponds to the minterm **ABC** above left. Above right we see that the minterm **A'BC'** corresponds directly to the cell **010**. A Boolean expression or map may have multiple minterms.

Referring to the above figure, Let's summarize the procedure for placing a minterm in a K-map:

- Identify the minterm (product term) term to be mapped.
- Write the corresponding binary numeric value.
- Use binary value as an address to place a **1** in the K-map
- Repeat steps for other minterms (P-terms within a Sum-Of-Products).



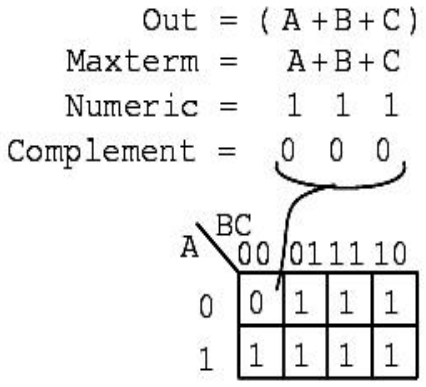
A Boolean expression will more often than not consist of multiple minterms corresponding to multiple cells in a Karnaugh map as shown above. The multiple minterms in this map are the individual minterms which we examined in the previous figure above. The point we review for reference is that the **1**s come out of the K-map as a binary cell address which converts directly to one or more product terms. By directly we mean that a **0** corresponds to a complemented variable, and a **1** corresponds to a true variable. Example: **010** converts directly to **A'BC'**. There was no reduction in this example. Though, we do have a Sum-Of-Products result from the minterms.

Referring to the above figure, Let's summarize the procedure for writing the Sum-Of-Products reduced Boolean equation from a K-map:

- Form largest groups of **1**s possible covering all minterms. Groups must be a power of 2.
- Write binary numeric value for groups.
- Convert binary value to a product term.
- Repeat steps for other groups. Each group yields a p-terms within a Sum-Of-Products.

Nothing new so far, a formal procedure has been written down for dealing with minterms. This serves as a pattern for dealing with maxterms.

Next we attack the Boolean function which is **0** for a single cell and **1**s for all others.



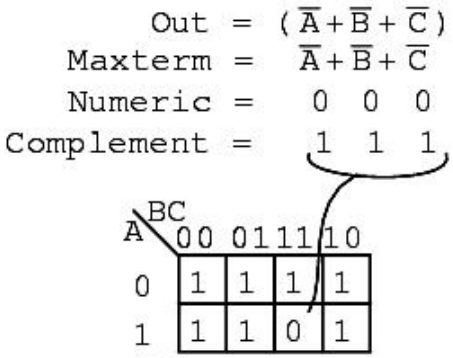
A *maxterm* is a Boolean expression resulting in a **0** for the output of a single cell expression, and **1**s for all other cells in the Karnaugh map, or truth table. The illustration above left shows the maxterm **(A+B+C)**, a single sum term, as a single **0** in a map that is otherwise **1**s. If a maxterm has a single **0** and the remaining cells as **1**s, it would appear to cover a maximum area of **1**s.

There are some differences now that we are dealing with something new, maxterms. The maxterm is a **0**, not a **1** in the Karnaugh map. A maxterm is a sum term, **(A+B+C)** in our example, not a product term.

It also looks strange that **(A+B+C)** is mapped into the cell **000**. For the equation **Out=(A+B+C)=0**, all three variables **(A, B, C)** must individually be equal to **0**. Only **(0+0+0)=0** will equal **0**. Thus we place our sole **0** for minterm **(A+B+C)** in cell **A,B,C=000** in the K-map, where the inputs are all**0** . This is the only case which will give us a **0** for our maxterm. All other cells contain **1**s because any input values other than (**(0,0,0)** for **(A+B+C)** yields **1**s upon evaluation.

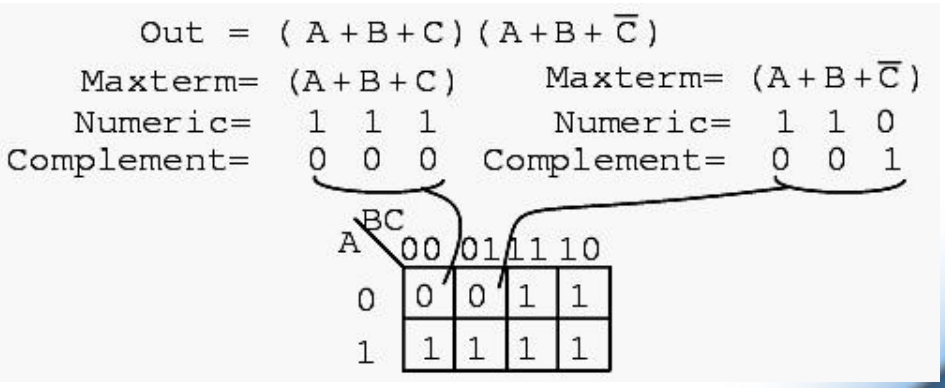
Referring to the above figure, the procedure for placing a maxterm in the K-map is:

- Identify the Sum term to be mapped.
- Write corresponding binary numeric value.
- Form the complement
- Use the complement as an address to place a **0** in the K-map
- Repeat for other maxterms (Sum terms within Product-of-Sums expression).



Another maxterm **A'+B'+C'** is shown above. Numeric **000** corresponds to **A'+B'+C'**. The complement is **111**. Place a **0** for maxterm **(A'+B'+C')** in this cell **(1,1,1)** of the K-map as shown above.

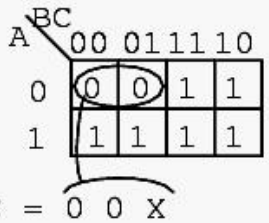
Why should **(A'+B'+C')** cause a **0** to be in cell **111**? When **A'+B'+C'** is **(1'+1'+1')**, all **1**s in, which is **(0+0+0)** after taking complements, we have the only condition that will give us a **0**. All the **1**s are complemented to all **0**s, which is **0** when **OR**ed.



A Boolean Product-Of-Sums expression or map may have multiple maxterms as shown above. Maxterm **(A+B+C)** yields numeric **111** which complements to **000**, placing a **0** in cell **(0,0,0)**. Maxterm **(A+B+C')** yields numeric **110** which complements to **001**, placing a **0** in cell **(0,0,1)**.

Now that we have the k-map setup, what we are really interested in is showing how to write a Product-Of-Sums reduction. Form the **0**s into groups. That would be a group of two below. Write the binary value corresponding to the sum-term which is **(0,0,X)**. Both A and B are **0** for the group. But, **C** is both **0** and **1** so we write an **X** as a place holder for **C**. Form the complement **(1,1,X)**. Write the Sum-term **(A+B)** discarding the **C** and the **X** which held its' place. In general, expect to have more sum-terms multiplied together in the Product-Of-Sums result. Though, we have a simple example here.

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



A B C = 0 0 X

Complement = 1 1 X

Sum-term = (A + B)

Out = (A + B)

Let's summarize the procedure for writing the Product-Of-Sums Boolean reduction for a K-map:

- Form largest groups of 0s possible, covering all maxterms. Groups must be a power of 2.
- Write binary numeric value for group.
- Complement binary numeric value for group.
- Convert complement value to a sum-term.
- Repeat steps for other groups. Each group yields a sum-term within a Product-Of-Sums result.

Example:

Simplify the Product-Of-Sums Boolean expression below, providing a result in POS form.

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

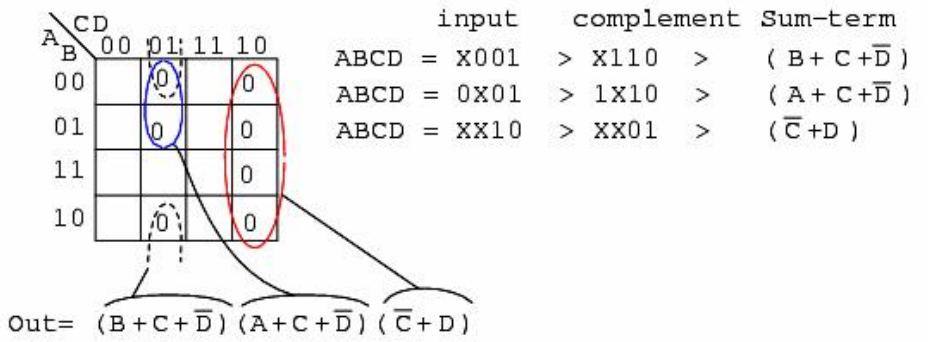
Solution:

Transfer the seven maxterms to the map below as **0**s. Be sure to complement the input variables in finding the proper cell location.

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

We map the **0**s as they appear left to right top to bottom on the map above. We locate the last three maxterms with leader lines..

Once the cells are in place above, form groups of cells as shown below. Larger groups will give a sum-term with fewer inputs. Fewer groups will yield fewer sum-terms in the result.



We have three groups, so we expect to have three sum-terms in our POS result above. The group of 4-cells yields a 2-variable sum-term. The two groups of 2-cells give us two 3-variable sum-terms. Details are shown for how we arrived at the Sum-terms above. For a group, write the binary group input address, then complement it, converting that to the Boolean sum-term. The final result is product of the three sums.

Example:

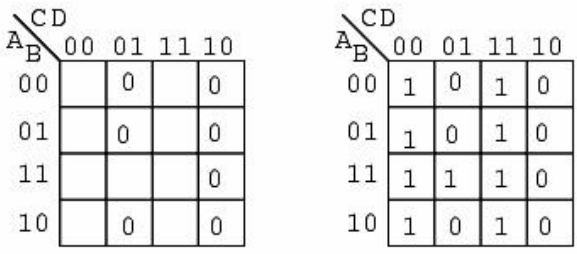
Simplify the Product-Of-Sums Boolean expression below, providing a result in SOP form.

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

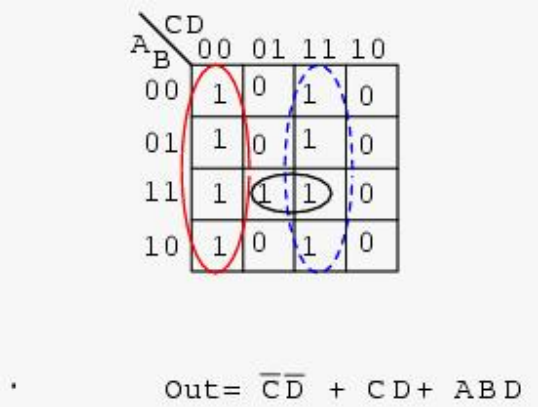
Solution:

This looks like a repeat of the last problem. It is except that we ask for a Sum-Of-Products Solution instead of the Product-Of-Sums which we just finished. Map the maxterm **0**s from the Product-Of-Sums given as in the previous problem, below left.

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

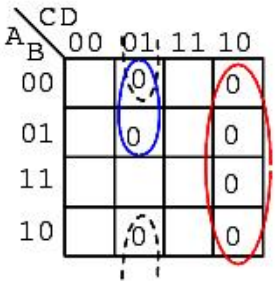


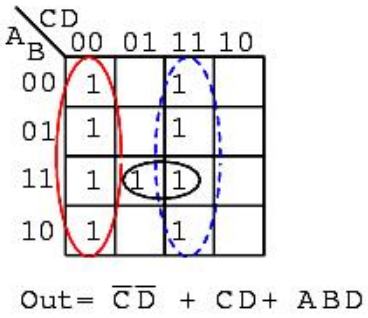
Then fill in the implied **1**s in the remaining cells of the map above right.



Form groups of **1**s to cover all **1**s. Then write the Sum-Of-Products simplified result as in the previous section of this chapter. This is identical to a previous problem.

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

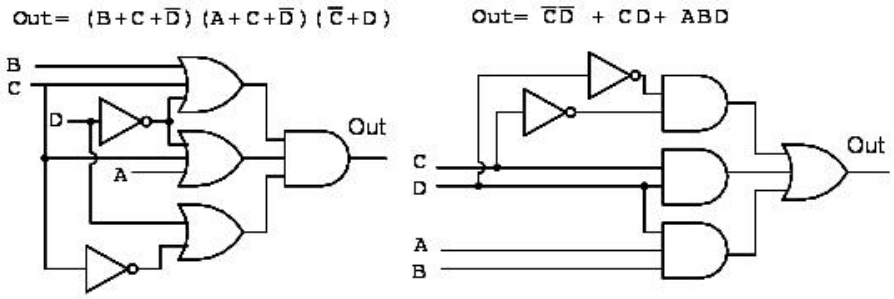




Out= $(B+C+\overline{D})(A+C+\overline{D})(\overline{C}+D)$

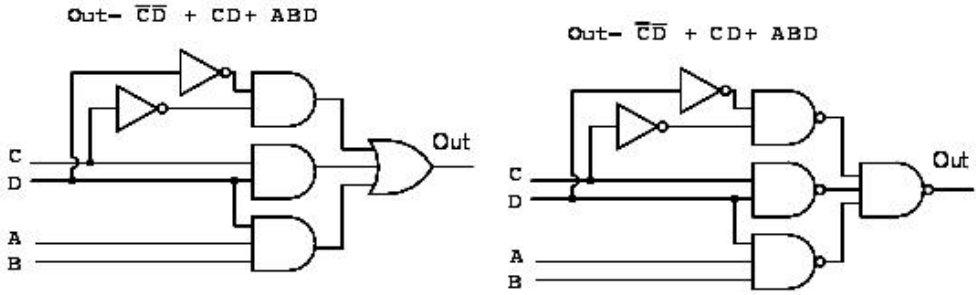
Above we show both the Product-Of-Sums solution, from the previous example, and the Sum-Of-Products solution from the current problem for comparison. Which is the simpler solution? The POS uses 3-OR gates and 1-AND gate, while the SOP uses 3-AND gates and 1-OR gate. Both use four gates each. Taking a closer look, we count the number of gate inputs. The POS uses 8-inputs; the SOP uses 7-inputs. By the definition of minimal cost solution, the SOP solution is simpler. This is an example of a technically correct answer that is of little use in the real world.

The better solution depends on complexity and the logic family being used. The SOP solution is usually better if using the TTL logic family, as NAND gates are the basic building block, which works well with SOP implementations. On the other hand, A POS solution would be acceptable when using the CMOS logic family since all sizes of NOR gates are available.

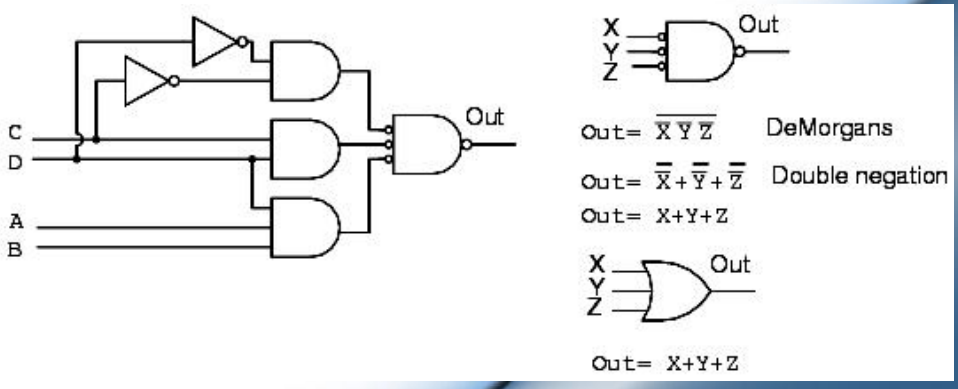


The gate diagrams for both cases are shown above, Product-Of-Sums left, and Sum-Of-Products right.

Below, we take a closer look at the Sum-Of-Products version of our example logic, which is repeated at left.



Above all AND gates at left have been replaced by NAND gates at right.. The OR gate at the output is replaced by a NAND gate. To prove that AND-OR logic is equivalent to NAND-NAND logic, move the inverter invert bubbles at the output of the 3-NAND gates to the input of the final NAND as shown in going from above right to below left.



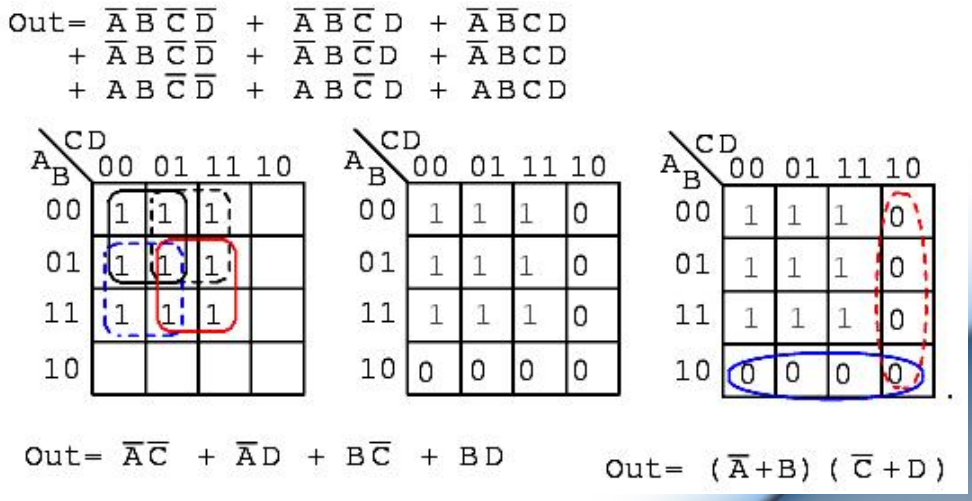
Above right we see that the output NAND gate with inverted inputs is logically equivalent to an OR gate by DeMorgan's theorem and double negation. This information is useful in building digital logic in a laboratory setting where TTL logic family NAND gates are more readily available in a wide variety of configurations than other types.

The Procedure for constructing NAND-NAND logic, in place of AND-OR logic is as follows:

- Produce a reduced Sum-Of-Products logic design.
- When drawing the wiring diagram of the SOP, replace all gates (both AND and OR) with NAND gates.
- Unused inputs should be tied to logic High.
- In case of troubleshooting, internal nodes at the first level of NAND gate outputs do NOT match AND-OR diagram logic levels, but are inverted. Use the NAND-NAND logic diagram. Inputs and final output are identical, though.
- Label any multiple packages U1, U2,.. etc.
- Use data sheet to assign pin numbers to inputs and outputs of all gates.

Example:

Let us revisit a previous problem involving an SOP minimization. Produce a Product-Of-Sums solution. Compare the POS solution to the previous SOP.

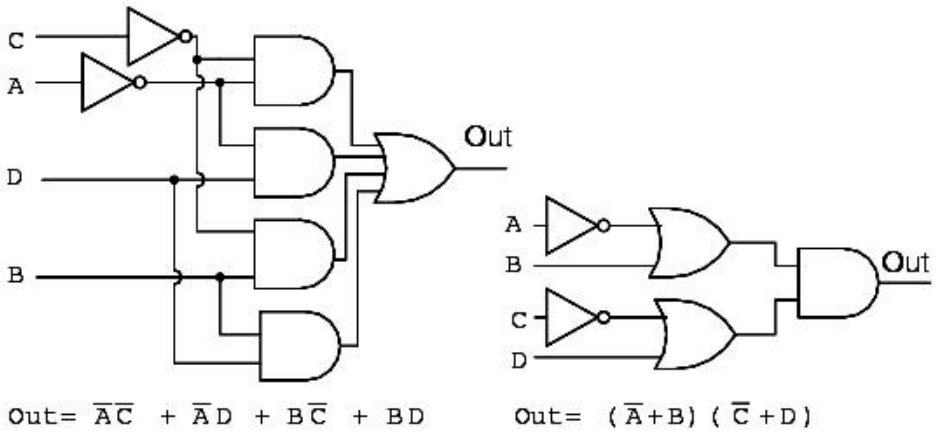


Solution:

Above left we have the original problem starting with a 9-minterm Boolean unsimplified expression. Reviewing, we formed four groups of 4-cells to yield a 4-product-term SOP result, lower left.

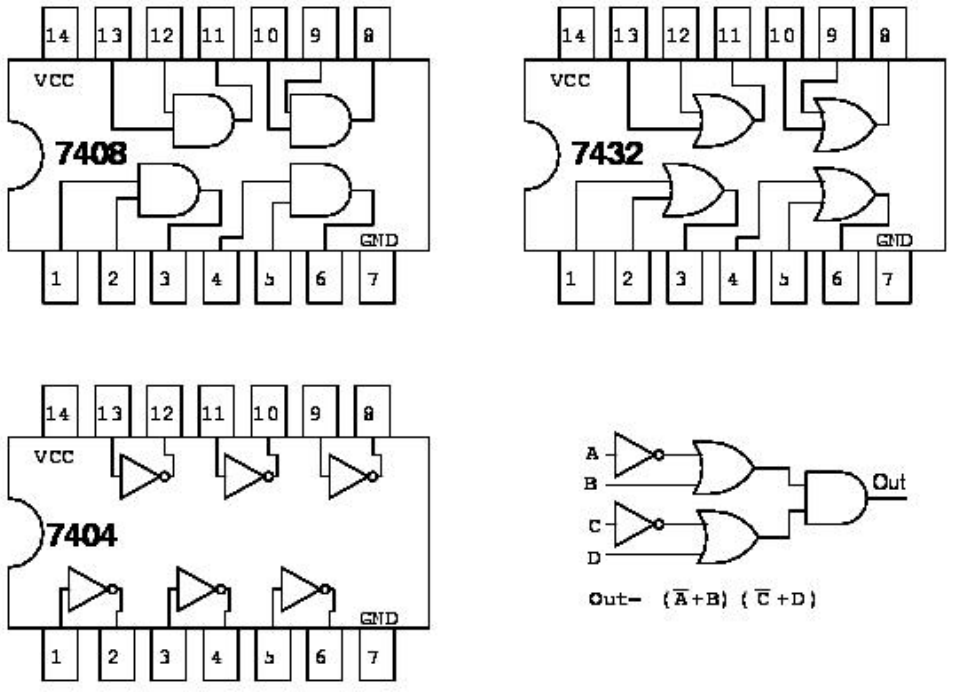
In the middle figure, above, we fill in the empty spaces with the implied **0**s. The **0**s form two groups of 4-cells. The solid blue group is **(A'+B)**, the dashed red group is **(C'+D)**. This yields two sum-terms in the Product-Of-Sums result, above right **Out = (A'+B)(C'+D)**

Comparing the previous SOP simplification, left, to the POS simplification, right, shows that the POS is the least cost solution. The SOP uses 5-gates total, the POS uses only 3-gates. This POS solution even looks attractive when using TTL logic due to simplicity of the result. We can find AND gates and an OR gate with 2-inputs.



The SOP and POS gate diagrams are shown above for our comparison problem.

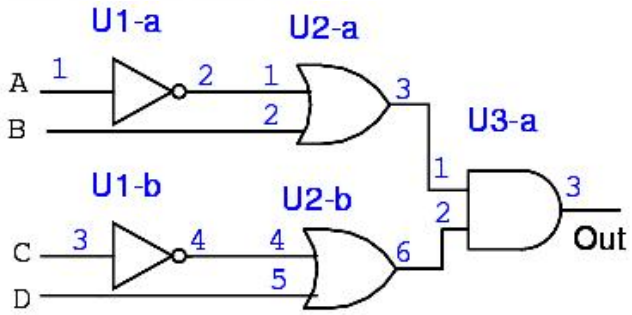
Given the pin-outs for the TTL logic family integrated circuit gates below, label the maxterm diagram above right with Circuit designators (U1-a, U1-b, U2-a, etc), and pin numbers.



Each integrated circuit package that we use will receive a circuit designator: U1, U2, U3. To distinguish between the individual

gates within the package, they are identified as a, b, c, d, etc. The 7404 hex-inverter package is U1. The individual inverters in it are U1-a, U1-b, U1-c, etc. U2 is assigned to the 7432 quad OR gate. U3 is assigned to the 7408 quad AND gate. With reference to the pin numbers on the package diagram above, we assign pin numbers to all gate inputs and outputs on the schematic diagram below.

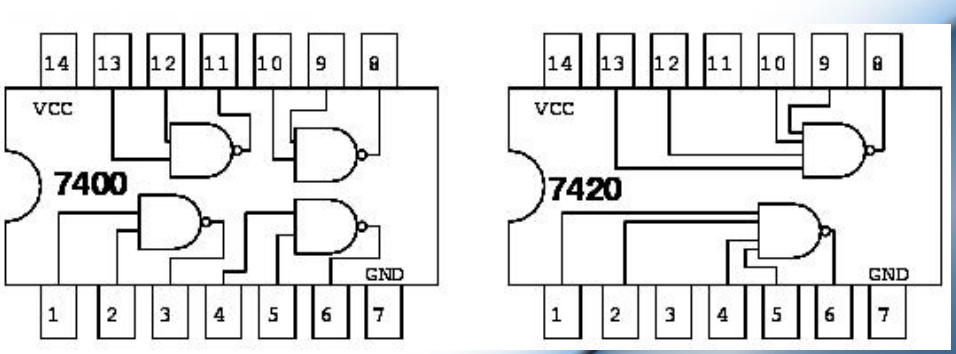
We can now build this circuit in a laboratory setting. Or, we could design a *printed circuit board* for it. A printed circuit board contains copper foil "wiring" backed by a non conductive substrate of phenolic, or epoxy-fiberglass. Printed circuit boards are used to mass produce electronic circuits. Ground the inputs of unused gates.



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

U1 = 7404
U2 = 7432
U3 = 7408

Label the previous POS solution diagram above left (third figure back) with Circuit designators and pin numbers. This will be similar to what we just did.

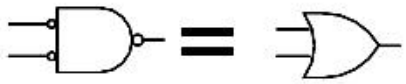


We can find 2-input AND gates, 7408 in the previous example. However, we have trouble finding a 4-input OR gate in our TTL catalog. The only kind of gate with 4-inputs is the 7420 NAND gate shown above right.

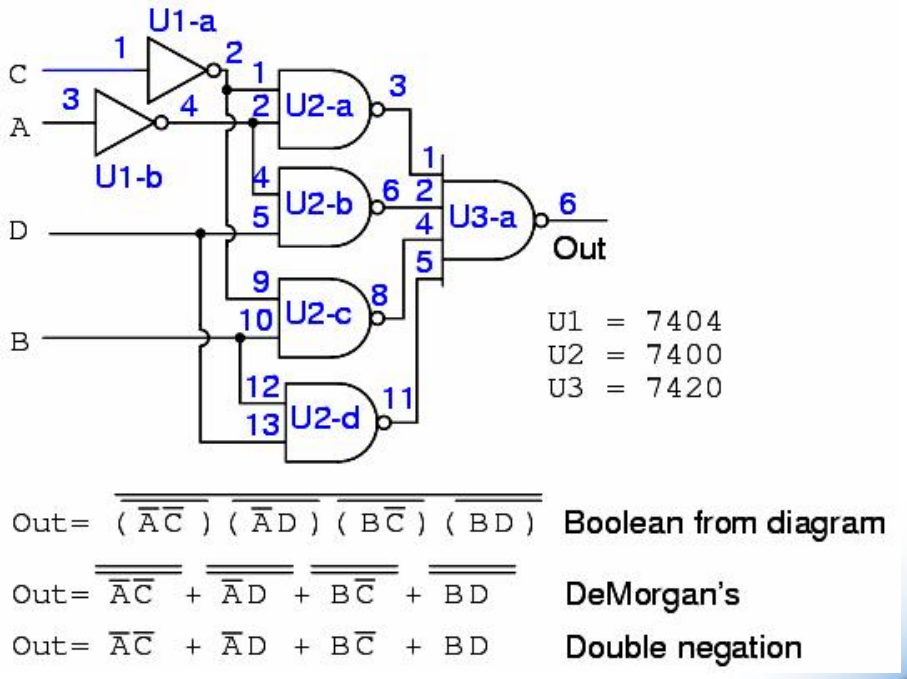
We can make the 4-input NAND gate into a 4-input OR gate by inverting the inputs to the NAND gate as shown below. So we

will use the 7420 4-input NAND gate as an OR gate by inverting the inputs.

$\overline{Y} = \overline{A}\,\overline{B} = \overline{A+B}$ DeMorgan's
$Y = A+B$ Double negation



We will not use discrete inverters to invert the inputs to the 7420 4-input NAND gate, but will drive it with 2-input NAND gates in place of the AND gates called for in the SOP, minterm, solution. The inversion at the output of the 2-input NAND gates supply the inversion for the 4-input OR gate.



The result is shown above. It is the only practical way to actually build it with TTL gates by using NAND-NAND logic replacing AND-OR logic.

4.5 QUINE MCCLUSKY METHOD

The Quine-McCluskey method is an exact algorithm which finds a minimum-cost sum-of-products implementation of a Boolean function. This handout introduces the method and applies it to several examples.

There are 4 main steps in the Quine-McCluskey algorithm:

1. Generate Prime Implicants
2. Construct Prime Implicant Table

3. Reduce Prime Implicant Table
    1. Remove Essential Prime Implicants
    2. Row Dominance
    3. Column Dominance
4. Solve Prime Implicant Table

In Step #1, the prime implicants of a function are generated using an iterative procedure. In Step #2, a prime implicant table is constructed. The columns of the table are the prime implicants of the function. The rows are minterms of where the function is 1, called *ON-set minterms*. The goal of the method is to cover all the rows using a minimum-cost *cover* of prime implicants.

The reduction step (Step #3) is used to reduce the size of the table. This step has three sub-steps **which are iterated until no further table reduction is possible!** At this point, the reduced table is either (i) empty or (ii) non-empty. If the reduced table is empty, the removed essential prime implicants form a minimum-cost solution. However, if the reduced table is *not* empty, the table must be ‘‘solved” (Step #4). The table can be solved using either ‘‘Petrick’s method” or the ‘‘branching method”. This handout focuses on Petrick’s method. The branching method is discussed in the books by McCluskey, Roth, etc., but you will not be responsible for the branching method.

The remainder of this handout illustrates the details of the Quine-McCluskey method on 3 examples. Example #1 is fairly straightforward, Examples #2 is more involved, and Example #3 applies the method to a function with ‘‘don’t-cares”. But first, we motivate the need for *column dominance* and *row dominance*.

Example #1:

$$F(A,B,C,D) = \Sigma m(0,2,5,6,7,8,10,12,13,14,15)$$

The Notation. The above notation is a shorthand to describe the Karnaugh map for F. First, it indicates that F is a Boolean function of 4 variables: A, B, C, and D. Second, each *ON-set minterm* of F is listed above, that is, minterms where the function is 1: 0, 2, 5, ….. Each of these numbers corresponds to one entry (or square) in the Karnaugh map. For example, the decimal number 2 corresponds to the minterm $ABCD = 0010$, (0010 is the binary representation of 2). That is, $ABCD = 0010$ is an ON-set minterm of F ; *i.e.,* it is a 1 entry. All remaining minterms, not listed above, are assumed to be 0.

Step 1: Generate Prime Implicants.

List Minterms

Column I	
0	0000
2	0010
8	1000
5	0101
6	0110
10	1010
12	1100
7	0111
13	1101
14	1110
15	1111

Combine Pairs of Minterms from Column I

A check (✓) is written next to every minterm which can combined with another minterm.

Column I		*Column II*	
0	0000 ✓	(0,2)	00-0
2	0010 ✓	(0,8)	-000
8	1000 ✓	(2,6)	0-10
5	0101 ✓	(2,10)	-010
6	0110 ✓	(8,10)	10-0
10	1010 ✓	(8,12)	1-00
12	1100 ✓	(5,7)	01-1
7	0111 ✓	(5,13)	-101
13	1101 ✓	(6,7)	011-

14 1110 √ (6,14) -110

15 1111 √ (10,14) 1-10

(12,13) 110-

(12,14) 11-0

(7,15) -111

(13,15) 11-1

(14,15) 111-

Combine Pairs of Products from Column II

A check (√) is written next to every product which can combined with another product.

Column III contains a number of duplicate entries, *e.g.* (0,2,8,10) and (0,8,2,10). Duplicate entries appear because a product in Column III can be formed in several ways. For example, (0,2,8,10) is formed by combining products (0,2) and (8,10) from Column II, and (0,8,2,10) (the same product) is formed by combining products (0,8) and (2,10).

Duplicate entries should be crossed out. The remaining unchecked products cannot be combined with other products. These are the prime implicants: (0,2,8,10), (2,6,10,14), (5,7,13,15), (6,7,14,15), (8,10,12,14) and (12,13,14,15); or, using the usual product notation: $B'D'$, CD', BD, BC, AD' and AB.

Column I			Column II			Column III	
0	0000	√	(0,2)	00-0	√	(0,2,8,10)	-0-0
2	0010	√	(0,8)	-000	√	(0,8,2,10)	-0-0
8	1000	√	(2,6)	0-10	√	(2,6,10,14)	-10
5	0101	√	(2,10)	-010	√	(2,10,6,14)	-10
6	0110	√	(8,10)	10-0	√	(8,10,12,14)	1-0
10	1010	√	(8,12)	1-00	√	(8,12,10,14)	1-0

12	1100	✓	(5,7)	01-1	✓	(5,7,13,15)	-1-1
7	0111	✓	(5,13)	-101	✓	(5,13,7,15)	-1-1
13	1101	✓	(6,7)	011-	✓	(6,7,14,15)	-11-
14	1110	✓	(6,14)	-110	✓	(6,14,7,15)	-11-
15	1111	✓	(10,14)	1-10	✓	(12,13,14,15)	11-
			(12,13)	110-	✓	(12,14,13,15)	11-
			(12,14)	11-0	✓		
			(7,15)	-111	✓		
			(13,15)	11-1	✓		
			(14,15)	111-	✓		

Step 2: Construct Prime Implicant Table.

	$B'D'$	CD'	BD	BC	AD'	AB
	(0,2,8,10)	(2,6,10,14)	(5,7,13,15)	(6,7,14,15)	(8,10,12,14)	(12,13,14,15)
0	X					
2	X	X				
5			X			
6		X		X		
7			X	X		
8	X				X	
10	X	X			X	
12					X	X

13			X			X
14		X		X	X	X
15			X	X		X

Step 3: Reduce Prime Implicant Table.
Iteration #1.
(i) Remove Primary Essential Prime Implicants

	$B'D'(*)$	CD'	$BD(*)$	BC	AD'	AB
	(0,2,8,10)	(2,6,10,14)	(5,7,13,15)	(6,7,14,15)	(8,10,12,14)	(12,13,14,15)
(▫) 0	X					
2	X	X				
(▫) 5			X			
6		X		X		
7			X	X		
8	X				X	
10	X	X			X	
12					X	X
13			X			X
14		X		X	X	X
15			X	X		X

* indicates an essential prime implicant
▫indicates a distinguished row, i.e. a row covered by only 1 prime implicant

In step #1, *primary essential prime implicants* are identified. These are implicants which will appear in *any* solution. A row which is covered by only 1 prime implicant is called a *distinguished row*. The prime implicant which covers it is an *essential prime implicant*. In this step, essential prime implicants are identified and removed. The corresponding column is crossed out. Also, each row where the column contains an X is completely crossed out, since these minterms are now covered. These essential implicants will be

added to the final solution. In this example, $B'D'$ and BD are both primary essentials.

(ii) Row Dominance

The table is simplified by removing rows and columns which were crossed out in step (i). (*Note:* you do not need to do this, but it makes the table easier to read. Instead, you can continue to mark up the original table.)

	CD'	BC	AD'	AB
	(2,6,10,14)	**(6,7,14,15)**	**(8,10,12,14)**	**(12,13,14,15)**
6	**X**	**X**		
12			**X**	**X**
14	**X**	**X**	**X**	**X**

Row 14 *dominates* both row 6 and row 12. That is, row 14 has an ''X" in every column where row 6 has an ''X" (and, in fact, row 14 has ''X"'s in other columns as well). Similarly, row 14 has in ''X" in every column where row 12 has an ''X". Rows 6 and 12 are said to be *dominated by* row 14.

A *dominating* row can always be eliminated. To see this, note that every product which covers row 6 also covers row 14. That is, if some product covers row 6, row 14 is *guaranteed* to be covered. Similarly, any product which covers row 12 will also cover row 14. Therefore, row 14 can be crossed out.

(iii) Column Dominance

	CD'	BC	AD'	AB
	(2,6,10,14)	**(6,7,14,15)**	**(8,10,12,14)**	**(12,13,14,15)**
6	**X**	**X**		
12			**X**	**X**

Column CD' *dominates* column BC. That is, column CD' has an ''X" in every row where column BC has an ''X". In fact, in this example, column BC also dominates column CD', so each is *dominated by* the other. (Such columns are said to *co-dominate* each other.) Similarly, columns AD' and AB dominate each other, and each is dominated by the other.

A *dominated* column can always be eliminated. To see this, note that every row covered by the dominated column is also

covered by the dominating column. For example, $C'D$ covers every row which BC covers. Therefore, the dominating column can always replace the dominated column, so the dominated column is crossed out. In this example, CD' and BC dominate each other, so either column can be crossed out (but not both). Similarly, AD' and AB dominate each other, so either column can be crossed out.

Iteration #2.
(i) Remove Secondary Essential Prime Implicants

	$CD'(**)$	$AD'(**)$
	(2,6,10,14)	(8,10,12,14)
( )6	X	
( )12		X

** indicates a secondary essential prime implicant
indicates a distinguished row

In iteration #2 and beyond, *secondary essential prime implicants* are identified. These are implicants which will appear in *any* solution, *given* the choice of column-dominance used in the previous steps (if 2 columns co-dominated each other in a previous step, the choice of which was deleted can affect what is an ‘‘essential” at this step). As before, a row which is covered by only 1 prime implicant is called a *distinguished row*. The prime implicant which covers it is a *(secondary) essential prime implicant*.

Secondary essential prime implicants are identified and removed. The corresponding columns are crossed out. Also, each row where the column contains an X is completely crossed out, since these minterms are now covered. These essential implicants will be added to the final solution. In this example, both CD' and AD' are secondary essentials.

Step 4: Solve Prime Implicant Table.

No other rows remain to be covered, so no further steps are required. Therefore, the minimum-cost solution consists of the primary and secondary essential prime implicants $B'D'$, BD, CD' and AD':

$$F = B'D' + BD + CD' + AD'$$

Example #2:

$$F(A,B,C,D) = \Sigma m(0,2,3,4,5,6,7,8,9,10,11,12,13)$$

Step 1: Generate Prime Implicants.

Use the method described in Example #1.

Step 2: Construct Prime Implicant Table.

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
2	X	X		X	X				
3				X	X				
4	X		X			X	X		
5						X	X		
6	X			X		X			
7				X		X			
8		X	X					X	X
9								X	X
10		X			X			X	
11					X			X	
12			X				X		X
13							X		X

Step 3: Reduce Prime Implicant Table.

Iteration #1.

(i) Remove Primary Essential Prime Implicants

There are no primary essential prime implicants: each row is covered by at least two products.

(ii) Row Dominance

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
2	X	X		X	X				
3				X	X				
4	X		X			X	X		
5						X	X		

6	X			X		X			
7				X		X			
8		X	X					X	X
9								X	X
10		X			X			X	
11					X			X	
12			X				X		X
13							X		X

There are many instances of row dominance. Row 2 dominates 3, 4 dominates 5, 6 dominates 7, 8 dominates 9, 10 dominates 11, 12 dominates 13. Dominating rows are removed.

(iii) Column Dominance

	$A'D'$	$B'D'$	$C'D'$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
0	X	X	X						
3				X	X				
5						X	X		
7				X		X			
9								X	X
11					X			X	
13							X		X

Columns $A'D'$, $B'D'$ and $C'D'$ each dominate one another. We can remove any two of them.

Iteration #2.

(i) Remove Secondary Essential Prime Implicants

	$A'D'(**)$	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
( )0	X						
3		X	X				
5				X	X		
7		X		X			
9						X	X

11			X			X	
13					X		X

** indicates a secondary essential prime implicant
▫indicates a distinguished row

Product $A'D'$ is a secondary essential prime implicant; it is removed from the table.

(ii) Row Dominance

No further row dominance is possible.

(iii) Row Dominance

No further column dominance is possible.

	$A'C$	$B'C$	$A'B$	BC'	AB'	AC'
3	X	X				
5			X	X		
7	X		X			
9					X	X
11		X			X	
13				X		X

There are no additional secondary essential prime implicants, and no further row- or column-dominance is possible.

There are two solutions. Both solutions have a minimal number of prime implicants, so either can be used. With either choice, we must include the secondary essential prime implicant, $A'D'$, identified earlier. Therefore, the two minimum-cost solutions are:

$$F = A'D' + A'C + BC' + AB'$$

$$F = A'D' + B'C + A'B + AC'$$

Example #3: Don't-Cares

$$F(A, B, C, D) = \Sigma m(2, 3, 7, 9, 11, 13) + \Sigma d(1, 10, 15)$$

Step 1: Generate Prime Implicants.

The don't-cares are *included* when generating prime implicants.

Note: As indicated earlier, you should learn this basic method for generating prime implicants (Step #1).

List Minterms

Column I		
1	0001	
2	0010	
3	0011	
9	1001	
10	1010	
7	0111	
11	1011	
13	1101	
15	1111	

Combine Pairs of Minterms from Column I

A check (✓) is written next to every minterm which can combined with another minterm.

Column I		*Column II*	
1	0001 ✓	(1,3)	00-1
2	0010 ✓	(1,9)	-001
3	0011 ✓	(2,3)	001-
9	1001 ✓	(2,10)	-010
10	1010 ✓	(3,7)	0-11
7	0111 ✓	(3,11)	-011
11	1011 ✓	(9,11)	10-1
13	1101 ✓	(9,13)	1-01

15 1111 √ (10,11) 101-

(7,15) -111

(11,15) 1-11

(13,15) 11-1

Combine Pairs of Products from Column II

A check (√) is written next to every product which can combined with another product.

Column I			*Column II*			*Column III*	
1	0001	√	(1,3)	00-1	√	(1,3,9,11)	-0-1
2	0010	√	(1,9)	-001	√	(2,3,10,11)	-01-
3	0011	√	(2,3)	001-	√	(3,7,11,15)	-11
9	1001	√	(2,10)	-010	√	(9,11,13,15)	1-1
10	1010	√	(3,7)	0-11	√		
7	0111	√	(3,11)	-011	√		
11	1011	√	(9,11)	10-1	√		
13	1101	√	(9,13)	1-01	√		
15	1111	√	(10,11)	101-	√		
			(7,15)	-111	√		
			(11,15)	1-11	√		
			(13,15)	11-1	√		

The unchecked products cannot be combined with other products. These are the prime implicants: (1,3,9,11), (2,3,10,11), (3,7,11,15) and (9,11,13,15); or, using the usual product notation: $B'D$, $B'C$, CD and AD.

Step 2: Construct Prime Implicant Table.
The don't-cares are *omitted* when constructing the prime implicant table, since they do not need to be covered.

	$B'D$	$B'C$	CD	AD
	(1,3,9,11)	(2,3,10,11)	(3,7,11,15)	(9,11,13,15)
2		X		
3	X	X	X	
7			X	
9	X			X
11	X	X	X	X
13				X

Step 3: Reduce Prime Implicant Table.
(i) Remove Essential Prime Implicants

	$B'D$	$B'C(*)$	$CD(*)$	$AD(*)$
	(1,3,9,11)	(2,3,10,11)	(3,7,11,15)	(9,11,13,15)
2		X		
3	X	X	X	
( )7			X	
9	X			X
11	X	X	X	X
( )13				X

* indicates an essential prime implicant
indicates a distinguished row

Step 4: Solve Prime Implicant Table.
The essential prime implicants cover all the rows, so no further steps are required. Therefore, the minimum-cost solution consists of the essential prime implicants $B'C$, CD and AD:

$$F = B'C + CD + AD$$

Thank You