Syllabus Content:

3.3.2 Boolean algebra
- show understanding of Boolean algebra
- show understanding of De Morgan’s Laws
- perform Boolean algebra using De Morgan’s Laws
- simplify a logic circuit/expression using Boolean algebra

3.3.1 Logic gates and circuit design
- produce truth tables for common logic circuits including half adders and full adders
- derive a truth table for a given logic circuit

3.3.4 Flip-flops
- show understanding of how to construct a flip-flop (SR and JK)
- describe the role of flip-flops as data storage elements

3.3.2 Boolean algebra

We have met gate logic and combination of gates. Another way of representing gate logic is through Boolean algebra, a way of algebraically representing logic gates. You should have already covered the symbols, below is a quick reminder:

<table>
<thead>
<tr>
<th>Bitwise Operator</th>
<th>NOT(A)</th>
<th>AND(·)</th>
<th>OR(+)</th>
<th>XOR(⊕)</th>
<th>NAND(A, B)</th>
<th>NOR(A + B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>invert input</td>
<td>where exactly two 1s</td>
<td>where one or more 1s</td>
<td>where exactly one 1</td>
<td>where less than two 1s</td>
<td>where exactly two 0s</td>
</tr>
</tbody>
</table>

**Boolean Operations and Expressions**

“Variable”, “Complement”, and “Literal” are terms used in Boolean Algebra.

A **variable** is a symbol used to represent a logical quantity. Any single variable can have a

“1” or a “0” value

The **complement** is the inverse of a variable and is indicated by a bar over the variable. **The complement of a variable is not considered as a different variable.**
Every occurrence of a variable or its complement is called a **Literal**. It could be its true form or its complement, both of them are called Literals.

A **SUM TERM** is the SUM of literals \((A+B+C+D)\)

A sum term is equal to **1** if one or all of its inputs are **1**, and is equal to **0** only if all of its inputs are zero.

A **PRODUCT TERM** is the PRODUCT of literals \((A.B.C.D)\)

A product term is equal to **1** if all of its inputs are **1**. And is equal to **0** if any one (or all) of its inputs are zero.

**Describing Logic Circuits Algebraically:**

Any logic circuit, no matter how complex, may be completely described using the Boolean operations previously defined. Because the **OR** gate, **AND** gate, and **NOT** gate are the basic building blocks of digital circuits.

Boolean algebra provides a concise way to express the operation of a logic circuit formed by a combination of logic gates so that the output can be determined for various combinations of input values.

To derive the Boolean expression for a given logic circuit, begin at the left most inputs and work towards the final output, writing the expression for each gate.

**Laws and Rules of Boolean Algebra:**

**Equivalent and Complement of Boolean Expressions**

Two given Boolean expressions are said to be equivalent if one of them equals "1" only when the other also equals "1" and same case with "0".

They are said to be **complement** of each other if one expression equals "1" only when the other equals "0" and vice versa.

**Postulates of Boolean Algebra:**

The following are the important postulates of Boolean algebra:

1. \(1.1 = 1\) & \(0+0 = 0\)
2. \(1.0 = 0.1 = 0\) & \(0+1 = 1+0\)
3. \(0.0 = 0\) & \(1+1 = 1\)
4. \(1 = 0\) & \(0 = 1\)
Theorems of Boolean Algebra

Boolean theorems can be useful in simplifying a logic expression. That is, in reducing the number of terms in the expression.

It is useful in the sense that the number of gates, are reduced which in turn also reduces heat dissipation from the circuit (saves energy)

When this is done, the reduced expression will produce a circuit that is less complex than the one which the original expression would have produced.

Commutative Laws

Rule 1:

For Addition:
\[ X + Y = Y + X \]
(i) \( A.B = B.A \)  
(ii) \( A + B = B + A \)

For Multiplication:
\[ X.Y = Y.X \]

Associative Laws:

Rule 2:

For Addition:
\[ X+(Y+Z) = Y+(Z+X) = Z+(X+Y) \]

For Multiplication
\[ X.(Y.Z) = Y.(Z.X) = Z.(X.Y) \]

Distributive Laws

Rule 3:

\[ X.(Y+Z) = X.Y + X.Z \]  
\[ (X.Y) + (X.Z) = X(Y+Z) \]
Operations with ‘0’ and ‘1’

Rule 4:

**OR Laws:**

These laws use the OR operation. Therefore they are called as OR laws.

- 0+X = X
- 1+X = 1

**AND Laws:**

These laws use the AND operation. Therefore they are called as AND laws.

- 0.X = 0
- 1.X = X

**Idempotent or Identity Laws:**

**RULE 5:**

- X.X.X.X..............X = X

**RULE 7:**

- X+X+X+X+.............+X = X

**Complementation Law:**

**RULE 6:**

- X.X = 0

**RULE 8:**

- X+X = 1

**Involution Law / INVERSION law:**

**RULE 9:**

This law uses the NOT operation. The inversion law states that double inversion of a variable result in the original variable itself.

- X = X or \( \overline{\overline{A}} = A \)
Absorption Law or Redundancy Law:

RULE 10:
- \( X + X \cdot Y = X \)

RULE 11:
- \( X + X \cdot Y = X + Y \)

RULE 12:
- \((X + Y) \cdot (X + Z) = X + Z \cdot Y\)

Explanation of Rule 10, Rule 11, Rule 12

---

**DeMorgan’s Theorem**

DE Morgan’s theorems provide mathematical verification of the equivalency of the NAND and negative-OR gates & the equivalency of the NOR and negative-AND gates.

In electrical and computer engineering, De Morgan’s laws are commonly written as:

**Theorem 1:**
The compliment of the product of 2 variables is equal to the sum of the compliments of individual variables

\[ \overline{A \cdot B} \equiv \overline{A} + \overline{B} \]

**Theorem 2:**
The compliment of the sum of two variables is equal to the product of the compliment of each variable

\[ \overline{A + B} \equiv \overline{A} \cdot \overline{B} \]
DeMorgan’s Theorems

(a) \[ X_1 + X_2 + X_3 + \ldots + X_n = \overline{X_1 \cdot X_2 \cdot X_3 \cdot \ldots \cdot X_n} \]

(b) \[ X_1 \cdot X_2 \cdot X_3 \cdot \ldots \cdot X_n = \overline{X_1 + X_2 + X_3 + \ldots + X_n} \]

NAND to Negative-OR

NOR to Negative-AND
Syllabus Content:

3.3.1 Logic gates and circuit design
- produce truth tables for common logic circuits including half adders and full adders
- derive a truth table for a given logic circuit

Half Adder:

The simplest circuit that can be used for binary addition is the half adder. This can be represented by the diagram in the circuit takes two input bits and outputs a sum bit (S) and a carry bit (C).

With the help of half adder, we can design circuits that are capable of performing simple addition with the help of logic gates. Let us first take a look at the addition of single bits.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

These are the least possible single-bit combinations. But the result for 1+1 is 10. Though this problem can be solved with the help of an EXOR Gate, if you do care about the output, the sum result must be re-written as a 2-bit output.

Thus the above equations can be written as

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Here the output ‘1’ of ‘10’ becomes the carry-out. The result is shown in a truth-table below. ‘SUM’ is the normal output and ‘CARRY’ is the carry-out.
From the equation it is clear that this 1-bit adder can be easily implemented with the help of EXOR Gate for the output ‘SUM’ and an AND Gate for the carry. Take a look at the implementation below. For complex addition, there may be cases when you have to add two 8-bit bytes together. This can be done only with the help of full-adder logic.

**Full Adder**

This type of adder is a little more difficult to implement than a half-adder. The main difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs. The first two inputs are A and B and the third input is an input carry designated as CIN. When a full adder logic is designed we will be able to string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next.

The output carry is designated as COUT and the normal output is designated as S. Take a look at the truth-table.
From the above truth-table, the full adder logic can be implemented. We can see that the output $S$ is an EXOR between the input $A$ and the half-adder SUM output with $B$ and $CIN$ inputs. We must also note that the $COUT$ will only be true if any of the two inputs out of the three are HIGH.

Thus, we can implement a full adder circuit with the help of two half adder circuits. The first will half adder will be used to add $A$ and $B$ to produce a partial Sum. The second half adder logic can be used to add $CIN$ to the Sum produced by the first half adder to get the final $S$ output. If any of the half adder logic produces a carry, there will be an output carry. Thus, $COUT$ will be an OR function of the half-adder Carry outputs. Take a look at the implementation of the full adder circuit shown below.

Though the implementation of larger logic diagrams is possible with the above full adder logic a simpler symbol is mostly used to represent the operation. Given below is a simpler schematic representation of a one-bit full adder.

With this type of symbol, we can add two bits together taking a carry from the next lower order of magnitude, and sending a carry to the next higher order of magnitude. In a computer, for a multi-bit operation, each bit must be represented by a full adder and must be added simultaneously. Thus, to add two 8-bit numbers, you will need 8 full adders which can be formed by cascading two of the 4-bit blocks. The addition of two 4-bit numbers is shown below.
Syllabus Content:

3.3.4 Flip-flops
- show understanding of how to construct a flip-flop (SR and JK)
- describe the role of flip-flops as data storage elements

Combinational circuits & Sequential circuits

All of the circuits so far encountered were **combinational circuits**. For such a circuit the output is dependent only on the input values. An alternative type of circuit is a **sequential circuit** where the output depends on the input and on the previous output.

**KEY TERMS**

- **Combinational circuit**: a circuit in which the output is dependent only on the input values
- **Sequential circuit**: a circuit in which the output depends on the input values and the previous output

**SR Flip-Flop**

The **SR flip-flop**, also known as a **SR Latch**, can be considered as one of the most basic sequential logic circuit possible. This simple flip-flop is basically a one-bit memory bistable device that has two inputs, one which will “SET” the device (meaning the output = “1”), and is labelled S and another which will “RESET” the device (meaning the output = “0”), labelled R.

**The Basic SR Flip-flop**

Then the SR description stands for “Set-Reset”. The reset input resets the flip-flop back to its original state with an output Q that will be either at a logic level “1” or logic “0” depending upon this set/reset condition.
A basic NAND gate SR flip-flop circuit provides feedback from both of its outputs back to its opposing inputs and is **commonly used in memory circuits to store a single data bit.**

Then the SR flip-flop actually has three inputs, Set, Reset and its current output Q relating to its current state or history. The term “Flip-flop” relates to the actual operation of the device, as it can be “flipped” into one logic Set state or “flopped” back into the opposing logic Reset state.

**The NAND Gate SR Flip-Flop**

A basic NAND gate SR flip-flop circuit provides feedback from both of its outputs back to its opposing inputs and is commonly used in memory circuits to store a single data bit.

Then the SR flip-flop actually has three inputs, Set, Reset and its current output Q relating to its current state or history.

As well as using NAND gates, it’s possible to construct simple one-bit **SR Flip-flops** using two cross-coupled NOR gates connected in the same configuration. The circuit will work in a similar way to the NAND gate circuit above, except that the inputs are active HIGH and the invalid condition exists when both its inputs are at logic level “1”, and this is shown below.

**The NOR Gate SR Flip-flop:**

We have to assume first time $S$ $R$ and $\bar{Q}$ to 0 for outputs in case of NOR Gate Flip-flop.
Truth Table for this Set-Reset Function:

<table>
<thead>
<tr>
<th>State</th>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1 no change</td>
</tr>
<tr>
<td>Reset</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0 no change</td>
</tr>
<tr>
<td>Invalid</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1 Invalid Condition</td>
</tr>
</tbody>
</table>

NOR & NAND Gate SR Flip-flops with Truth Tables:

The JK flip-flop

In addition to the possibility of entering an invalid state there is also the potential for a circuit to arrive in an uncertain state if inputs do not arrive quite at the same time. In order to prevent
this, a circuit may include a clock pulse input to give a better chance of synchronizing inputs. The JK flip-flop is an example.

The JK flip-flop can be illustrated by the symbol shown in Figure (a). A possible circuit is shown in Figure (b).

![Figure (a) A symbol for a JK flip-flop and (b) a possible circuit](image)

The workings of the circuit are viewed in terms of the value of the Q output immediately after the circuit detects a clock pulse. The J input acts as a set input and the K as a clear, so there is some similarity to the functioning of the SR flip-flop. However, if both J and K are input as a 1 then Q always switches value.

The significant part of the truth table is shown as Table.

![Truth table for a JK flip-flop](image)

**Typical applications of SR flip-flop**

The **basic building block that makes computer memories possible**, and is also used in many sequential logic circuits is the flip-flop or bi-stable circuit.

Just two interconnected logic gates make up the basic form of this circuit whose output has two stable output states. When the circuit is triggered into either one of these states by a suitable input pulse, it will ‘remember’ that state until it is changed by a further input pulse, or until power is removed.

For this reason the circuit may also be called a Bi-stable Latch.

Some of the common uses of the Flip-Flops are as follows:

- Registers
- Counters
- Memory, etc.
**Registers:** A register is composed of a group of flip-flops to store a group of bits.

**Counters:** Counters are used for count the events. Electrical pulses corresponding to the event are produced using transducers & these pulses counted using a counter.

**Random access memory:** In computers, digital control systems, information processing systems it is necessary to store digital data and retrieve the data as desired.

Flip-Flops can be used for making memories in which data can be stored for any desired length of time and then readout whenever required.

**References:**
AS & A level by Silvia Langfield and Dave Duddell
https://en.wikipedia.org/wiki/De_Morgan%27s_laws
http://www.electronicshub.org/boolean-algebra-laws-and-theorems/
http://www.electronics-tutorials.ws/sequential/seq_1.html
https://www.elprocus.com/half-adder-and-full-adder/
http://www.learnabout-electronics.org/Digital/dig52.php