Digital Circuits & Logic Design

Dhruva Hegde

Introduction to Digital Logic

Digital electronics is a field of electronics involving the study of digital signals and the engineering of devices that use or produce them.
Current technology majorly involves digital systems for storage and processing of data (in the form of digital signals) even though most signals appearing in nature are analog signals.

This document mainly explains the design of digital logic for circuits and systems, but does not delve into device (transistor) level implementation of these circuits. Digital logic design deals only with logical functionality of the required system.

Advantages of Digital over Analog:

- Effect of noise is reduced since the signal can take only two values and it is easy to recover original signal just by estimation if there are any small variations due to noise.
- Accuracy can be increased simply by increasing the number of quantized levels, meaning there is no need for new hardware design to enhance accuracy.
- Margin for human error in digital measurements is negated since there will be no parallax error (unlike in analog measurements).

1 Number Systems & Codes

A numeral system is a writing system for expressing numbers i.e, a mathematical notation for representing numbers of a given set using digits.
The base of a number system gives the maximum number of numbers that can be represented in that system using a single digit.

An 'n' digit number of base 'r' is represented as $b_n\ b_{n-1}\ \ldots\ b_2\ b_1\ b_0$ where b_n is the most significant bit (MSB) and b_0 is the least significant bit (LSB).
In case representation of fractions using decimal point is necessary, then a general number is represented as $b_n\ b_{n-1}\ \ldots\ b_2\ b_1\ b_0\ .\ b_{-1}\ b_{-2}\ \ldots$

Base of any number number must be greater than the maximum value of any single digit present in the number.
$\implies r \geq d_{max} + 1$

For example, if a number is 562.14, then the minimum possible base for it is 6+1 = 7. (Note that the base can be anything more than 7).

The **decimal numeral system** is the standard system for denoting integer and non-integer numbers, its base is 10.
Decimal equivalent of any number of base 'r' is given by,
$b_{n-1}r^{n-1} + b_{n-2}r^{n-2} + \ldots + b_2r^2 + b_1r + b_0 + b_{-1}r^{-1} + b_{-2}r^{-2} + \ldots$

1.1 Binary number system

Binary is most familiar number system to the digital systems. It has base 2 which has only 2 symbols: 0 and 1. These digits can be represented by off and on respectively, hence all digital systems are based on binary.

Binary to Decimal:
Since decimal system is used in general for showing results to interpret, it is important for digital systems to be able to convert the results which are obtained in binary to decimal number system, which is done using the method explained earlier.
For example, the binary number 100.101 in decimal will be $1 \times 2^2 + 0 \times 2 + 0 \times 1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2 - 3 = 4.625$

Decimal to Binary Conversion:
Similarly, as all numbers fed into a digital system are usually in decimal form, the system must be able to convert it to binary before performing the necessary computations.

A decimal number is converted to binary using repeated division for digits before point and repeated multiplication after point (the two can be done individually and added).

The procedure for converting an integral part decimal number is as follows.

- Take decimal number as dividend.
- Divide this number by 2 (2 is base of binary so divisor here).
- Store the remainder in an array (it will be either 0 or 1 because of divisor 2).
- Repeat the above two steps until the number is greater than zero.
- Print the array in reverse order (which will be equivalent binary number of given decimal number).

The procedure for converting an fractional part of decimal number is as follows.

- Take decimal number as multiplicand.
- Store the value of integer part of result in an array (it will be either 0 or 1 because of multiplier 2).
- Repeat the above two steps until the number became zero.
- Print the array (which will be equivalent fractional binary number of given decimal fractional number).

1.2 Octal number system

Octal system is the number system with base 8 and hence has 8 symbols: 0,1,2,3,4,5,6 and 7.

Octal to decimal conversion can be done using the formulae given earlier using $r = 8$.

Decimal to octal conversion is done using a method similar to the method used for binary to decimal conversion. The only difference is that the number used to divide (for whole number part) and to multiply (for fractional part) is 8 instead of 2.

Octal number can be converted to binary number by simply writing down the 3-bit binary equivalent of each octal digit.

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

For example, $(307.42)_8 = (011000111.100010)_2$.

Similarly, a binary number can be converted to octal number by grouping the bits in threes. In case the number of bits is not a multiple of 3, then extra zeros can be added before the whole number part and after the decimal part as they will not change the value.

For example, $(1011.1101)_2 = (001011.110100)_2 = (13.64)_8$

1.3 Hexadecimal number system

Hexadecimal system is the number system with base 16 and hence has 16 symbols: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Hexadecimal to decimal conversion can be done using the formulae given earlier using $r = 16$.

Decimal to hexadecimal conversion is done using a method similar to the method used for binary or octal to decimal conversion. The only difference is that the number used to divide (for whole number part) and to multiply (for fractional part) is 16 instead of 2 or 8.

Hexadecimal number can be converted to binary number by simply writing down the 4-bit binary equivalent of each octal digit.

Hexadecimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

For example, $(\text{AD5.B3})_{16} = (101011010101.10110011)_2$.

Similarly, a binary number can be converted to hexadecimal number by grouping the bits in fours. In case the number of bits is not a multiple of 4, then extra zeros can be added before the whole number part and after the decimal part as they will not change the value.

For example, $(11110.1001)_2 = (00011110.1001)_2 = (\text{1C.9})_{16}$

Note that conversion between octal and hexadecimal systems can be conveniently done by first converting to binary.

1.4 Representation of numbers in binary

In designing digital systems, all numbers have to be represented in binary.

Unsigned numbers are set of positive numbers (including zero). They do not need a sign bit to indicate the sign. Hence, the binary equivalent of the number is its unsigned binary representation.
This means all $'n'$ bits contribute to magnitude of the number only.
Eg: $(14)_{10} = 001110$ (6 bit representation)

Range of unsigned numbers that can be represented using 'n' bits: 0 to $2^n - 1$.

Signed numbers are set of both positive and negative numbers. They have the necessity of a sign bit to indicate the sign of the number, which will conventionally be the first bit.
There are 3 forms of representing signed numbers in binary:

1. Sign-magnitude form
2. 1's compliment form
3. 2's compliment form

Sign-magnitude form:
The first bit indicates the sign of the number and the rest of the bits give the magnitude.

- sign bit $= 0 \implies$ positive number
- sign bit $= 1 \implies$ negative number

After deciding the sign based on the above convention, the magnitude is simply found using the rest of the bits similar to an unsigned number.
Eg: $(+6)_{10} = 000110$ and $(-6)_{10} = 100110$ (6 bit representation)

Zero (0) will have two representations here i.e 000000 and 100000 both are 6 bit representations of 0.
Range of numbers that can be represented using 'n' bits in Sign-magnitude form: $-(2^{n-1} - 1)$ to $2^{n-1} - 1$.

1's compliment form:
The negative of a number is represented using its 1's compliment.
1's compliment operation just negates all the bits i.e 0 becomes 1 and 1 becomes 0.
$\implies (-N)_r = \text{1's compliment}[(+N)_r]$

As a convention, the numbers with first bit (sign bit) being 0 are positive and those with sign bit being 1 are negative. If a number is found to be negative, then the magnitude of that negative number is obtained by taking 1's compliment of it and finding the magnitude.
Eg: $(+6)_{10} = 000110$ and $(-6)_{10} = 111001$ (6 bit representation)

Zero (0) will have two representations here i.e 000000 and 111111 both are 6 bit representations of 0.
Range of numbers that can be represented using 'n' bits in 1's compliment form: $-[(2^{n-1} - 1]$ to $2^{n-1} - 1$.

2's compliment form:
The negative of a number is represented using its 2's compliment.
2's compliment operation is equivalent to taking 1's compliment and adding 1 to it.
$\implies (-N)_r = \text{2's compliment}[(+N)_r] = \text{1's compliment}[(+N)_r] + 1$

Again, as usual the numbers with first bit (sign bit) being 0 are positive and those with sign bit being 1 are negative.
If a number is found to be negative, then the magnitude of that negative number is obtained by taking 2's compliment of it and finding the magnitude. Or another way is to simply take the binary weight sum of the bits while putting a negative sign only to the left most bit.
Eg: $(+6)_{10} = 000110$ and $(-6)_{10} = 111010$ (6 bit representation)

Unlike the previous two forms, Zero (0) will have only one representation i.e 000000 (6 bit representation).
Range of numbers that can be represented using 'n' bits in 1's compliment form: -2^{n-1} to $2^{n-1} - 1$.

A few observations regarding the 3 representations of signed numbers:

- For positive numbers, all 3 representations are always the same.
- 2's compliment of a number is same as 2's compliment representation of its negative (applies for 1's compliment as well).
- The sign bit can be extended in the left without changing the magnitude in 2's and 1's compliment forms.
- The range for 2's compliment form is one more than the range for 1's compliment form and sign magnitude form due to non-ambiguity about zero, which is why it is more widely used.

1.4.1 2's compliment arithmetic

As mentioned earlier, due to ambiguity in representation of zero, sign-magnitude forms and 1's compliment form are not used, especially while performing arithmetic operations.
2's compliment form is always preferred. Also, it will be proved later that using 2's compliment form, addition and subtraction can be done using the same circuit.

If A and B are two numbers represented in 2's compliment form, then: $A - B = A + \text{2's compliment}(B)$.

Observations regarding 2's compliment addition:

- When the two numbers are of different signs, then the last carry bit is discarded.
- When the two numbers are of same signs, then the last carry bit is retained.

Shifting in binary is equivalent to scaling the magnitude by 2.

- Multiplying a number with 2^n is same as performing left shift n times with 0 padding.
- Dividing a number by 2^n is same as performing right shift n times with retention of the left most bit.

1.5 Binary Codes

Any sort of data is converted to a sequence of binary digits (bits) for convenient storage and transmission. A binary code represents any data using a two-symbol system (usually '1' and '0').

Alphanumeric code:
A binary code that can be used for representing both alphabets and numbers.
Eg: ASCII code

Numeric code:
A binary code that can be used for representing only numbers.
Eg: 8421 code, Grey code.

Weighted code will have positional weights whereas non-weighted code will not have positional weights.

1.5.1 Binary Coded Decimal (BCD)

Each decimal digit is represented using 4 bit binary representation. BCD code is also called 8421 code.

Decimal Digits	8421 code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Note that 1010, 1011, 1100, 1101, 1110 and 1111 are not valid BCD codes.

If there are 'k' decimal digits, then there will be '$4k$' bits in its BCD representation.

1.5.2 XS3 Code

XS3 code is obtained by adding 3 (0011) to the BCD code of each digit.

Decimal Digits	XS3 code
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

Note that 0000, 0001, 0010, 1101, 1110 and 1111 are not valid XS3 codes.

A binary code is said to be self-complimentary if the 9's compliment of a decimal digit is equal to the 1's compliment of the corresponding binary representation.
XS3 code is a self-complimentary code, and hence is used in decimal computations because it is very easy to find 9's compliment.

1.5.3 Grey Code

Disadvantage of BCD and XS3 codes is that changing state from one number to next number may require multiple bit compliments. For example, 7 to 8 needs all 4 bits to be complimented.

It will be computationally more efficient if numbers can be represented such that the number of bit positions differed by successive codes is one (1). Such codes are called 'Unit Distance Codes'.

The most popular unit distance code is Grey Code.
Grey code is used in simplifying Boolean expressions using K-map which will be dealt with later.
Grey code is non-weighted code and non-sequential code; hence it can't be used for arithmetic operations.

Binary to Grey conversion (4 bits):

$$\begin{aligned}
G_3 &= B_3\\
G_2 &= B_3 \oplus B_2\\
G_1 &= B_2 \oplus B_1\\
G_0 &= B_1 \oplus B_0
\end{aligned}$$

Grey to Binary conversion (4 bits):

$$\begin{aligned}
B_3 &= G_3\\
B_2 &= G_3 \oplus G_2\\
B_1 &= G_3 \oplus G_2 \oplus G_1\\
B_0 &= G_3 \oplus G_2 \oplus G_1 \oplus G_0
\end{aligned}$$

($\oplus$ is the XOR operator, will be defined eventually)

2 Boolean Algebra & K-Maps

The type of algebra which involves variables which can take only binary values i.e 0 or 1 is called 'boolean algebra'.

The basic boolean operations that are allowed are - **AND**, **OR** and **NOT**. Other operations such as **NAND**, **NOR**, **XOR** (Exclusive OR) and **XNOR** (Exclusive NOR) are derived from these.

Digital circuits are designed and analysed using boolean algebra techniques for simplification.

2.1 Boolean Algebra Laws and Terminology

If X,Y and Z are some boolean variables, $\overline{X}$, $\overline{Y}$ and $\overline{Z}$ are their complements.

Duality Principle:
By interchanging logical "AND" and logical "OR", "0" and "1", the dual function of the original function is obtained.

- $X.X = X \iff X + X = X$
- $X.\overline{X} = 0 \iff X + \overline{X} = 1$
- $X.1 = X \iff X + 0 = X$
- $X.0 = 0 \iff X + 1 = 1$

Commutative Law:
$X.Y = Y.X \iff X + Y = Y + X$

Associative Law:
$X.(Y.Z) = (X.Y).Z \iff X + (Y + Z) = (X + Y) + Z$

Distributive Law:
$X.(Y + Z) = X.Y + X.Z \iff X + (Y.Z) = (X + Y).(X + Z)$

- $X + XY = X$
- $X + (\overline{X}.Y) = X + Y$

Concensus Law:
$X.Y + \overline{X}.Z + Y.Z = X.Y + \overline{X}.Z$
$(X + Y).(\overline{X} + Z).(Y + Z) = (X + Y).(\overline{X} + Z)$

De-Morgan's Law:
$\overline{X + Y} = \overline{X}.\overline{Y}$
$\overline{X.Y} = \overline{X} + \overline{Y}$

All six 2-input functions are commutative. However, only AND, OR and XOR are associative. NAND, NOR and XNOR are not associative.

Steps to find complement of any boolean function F:

1. Write dual function of F, say F_d.
2. Complement each literal of F_d to get $\overline{F}$.

(note that a literal is simply a boolean variable or its complement)

2.1.1 Minterms and Maxterms

Minterms are product terms (i.e literals combined with AND operation) in which the number of literals is equal to the number of variables in the function.

Maxterms are sum terms (i.e literals combined with OR operation) in which the number of literals is equal to the number of variables in the function.

If there are n boolean variables in a function, then there will be 2^n minterms and 2^n maxterms.

2 variable example:

X	Y	Minterms	Maxterms
0	0	$\overline{X}.\overline{Y}$ (m_0)	$X + Y$ (M_0)
0	1	$\overline{X}.Y$ (m_1)	$X + \overline{Y}$ (M_1)
1	0	$X.\overline{Y}$ (m_2)	$\overline{X} + Y$ (M_2)
1	1	$X.Y$ (m_3)	$\overline{X} + \overline{Y}$ (M_3)

Respective minterms and maxterms are complements of each other.
$\implies m_i = \overline{M_i}$

Properties of minterms

- Sum of all minterms is equal to 1
- Product of any two different minterms is equal to 0

Properties of maxterms

- Product of all maxterms is equal to 0
- Sum of any two different maxterms is equal to 1

Canonical Sum of Products:
Canonical SOP form is the expression of a boolean function as a sum of minterms.
For example, $F = m_1 + m_2 = \overline{X}.Y + X.\overline{Y}$

Canonical Product of Sums:
Canonical POS form is the expression of a boolean function as a product of maxterms.
For example, $F = M_0.M_3 = (X + Y).(\overline{X} + \overline{Y})$

The Canonical SOP or POS forms must be reduced to minimal expressions using either boolean algebra or K-map before realization/implementation.

2.1.2 Limitations of simplification using boolean algebra

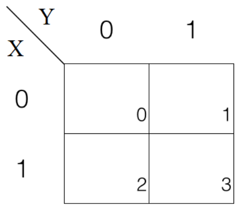
- Time consuming process if variables are more
- Absence of unique procedure which means a generalized computer program can't be written to simplify a given boolean function
- No guaranteed way of checking if simplified expression is simplest or even correct
- Prone to human errors since each step needs identification of common terms and redundant terms

These limitations can be overcome using K-maps.

2.2 Karnaugh Map Simplification

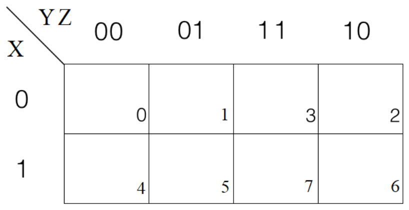
The Karnaugh map (KM or K-map) is a more generalized method of simplifying Boolean algebra expressions.
The Karnaugh map reduces the need for extensive calculations by taking advantage of humans' pattern-recognition capability.

2 variable K-map:



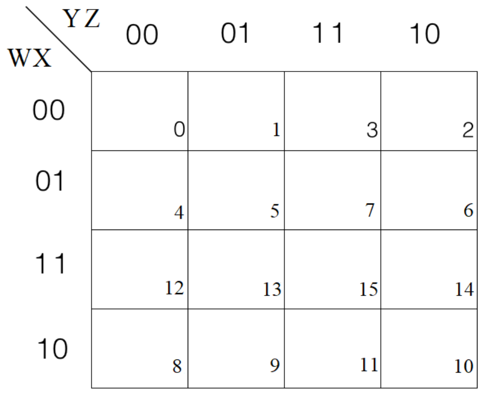
X \ Y	0	1
0	0	1
1	2	3

3 variable K-map:



X \ YZ	00	01	11	10
0	0	1	3	2
1	4	5	7	6

4 variable K-map:



WX \ YZ	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Prime Implicants
The product term which is obtained by grouping of adjacently placed 1s.

Essential Prime Implicants
A prime implicant is said to be essential if at least one 1 in the grouping is not a part of any other groupings.

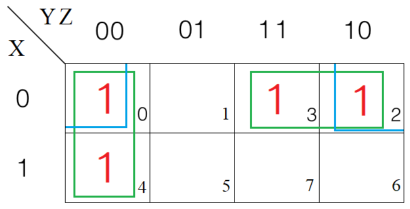
Simplification procedure for SOP form:

- Place 1s in respective minterm positions in the K-map
- Make groups of maximum possible adjacantly placed 1s in decreasing powers of 2
- Minimzied function = EPIs + optional PIs

In the K-maps illustrated in the following examples, the green boxes indicate the essential prime implicants (EPIs) and the blue boxes indicate the prime implicants (PIs).

Example 1:
Consider the following 3 variable expression written in SOP form.

$$f(X,Y,Z) = \sum_m (0,2,3,4)$$

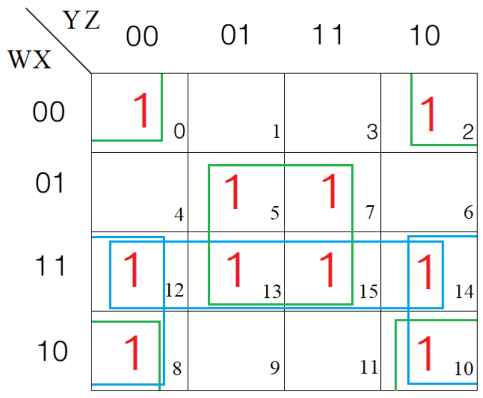


There are 2 EPIs and 1 unnecessary PI. Hence, the minimized expression will be as follows.

$$\implies f(X,Y,Z) = \overline{Y}.\overline{Z} + \overline{X}.Y$$

Example 2:
Consider the following 4 variable expression written in SOP form.

$$f(W,X,Y,Z) = \sum_m (0,2,5,7,8,10,12,13,14,15)$$

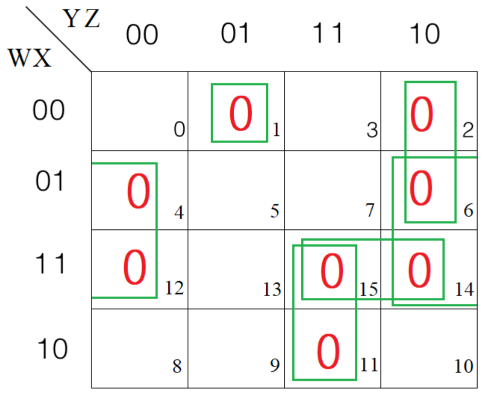


In this example, there are 2 EPIs and 2 PIs. To get the minimized expression, either one of the PIs has to be used (the logic will not change either way).

$$\implies f(W,X,Y,Z) = X.Z + \overline{X}.\overline{Z} + W.X = X.Z + \overline{X}.\overline{Z} + W.\overline{Z}$$

Example 3:
Consider the following 4 variable expression written in POS form.

$$f(W,X,Y,Z) = \Pi_M(1,2,4,6,11,12,14,15)$$



In this example, there are 5 EPIs and many PIs all of which are unnecessary. Hence, the minimized logic function will be as follows.

$$\implies f(W,X,Y,Z) = (\overline{X}+Z).(\overline{W}+\overline{X}+\overline{Y}).(\overline{W}+\overline{Y}+\overline{Z}).(W+\overline{Y}+Z).(W+X+Y+\overline{Z})$$

Don't Care Terms

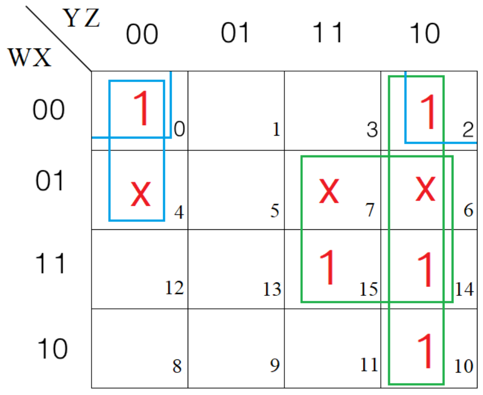
For non occurring input combinations in a circuit, the output can be taken as either 0 or 1 because it doesn't matter. Such terms are called don't care terms.
For example, outputs for input combinations from 1010 to 1111 in BCD code.

While obtaining minimal SOP expression, the don't care terms that can help minimize the function more are treated as 1s.
Note that it is not necessary to treat all don't cares as 1s. Only those that can combine with 1s to form larger groups are considered while the others are ignored.
Similarly for POS expression, they are treated as 0s.

Don't care terms are represented using $\times$.

Example:
Consider the following 4 variable expression containing don't care terms written in SOP form.

$$f(W, X, Y, Z) = \sum_m (0, 2, 10, 14, 15) + d(4, 6, 7)$$



There are 2 EPIs and 2 PIs. Hence, the minimized expression will be as follows.

$$f(W, X, Y, Z) = X.Y + Y.\overline{Z} + \overline{W}.\overline{X}.\overline{Z} = X.Y + Y.\overline{Z} + \overline{W}.\overline{Y}.\overline{Z}$$

3 Logic Gates

Logic gates are the fundamental functional blocks of any digital circuit. They are electronic circuits that produce a logic function. Each logic gate is named after the logic function it implements.

3.1 Types of Logic Gates

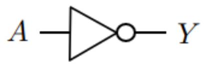
There are mainly 7 different logic gates. They are AND, OR, NOT, NAND, NOR, XOR and XNOR gates. The buffer is sometimes considered a logic gate too.

AND, OR, NOT are called **Fundamental Gates** since it is possible to realize any boolean function using a combination of these 3 gates.

NAND, NOR are called **Universal Gates** since it is possible to realize any boolean function using only NAND gates or only NOR gates.

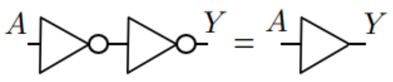
(Note that ideally the output of logic gates will change instantly with change in inputs, but practically there will be some latency or delay before the gate can respond. More on this will be elaborated eventually)

NOT Gate
The NOT Gate simply complements the input variable. Hence it also called the "Inverter".



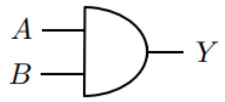
A	Y
0	1
1	0

Buffer
The Buffer simply outputs the input variable as it is. It can be modeled as 2 cascaded inverters.



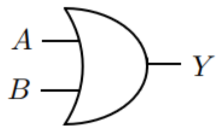
A	Y
0	0
1	1

AND Gate
The AND Gate takes 2 (or more) inputs and gives high if all the inputs are high, low otherwise.



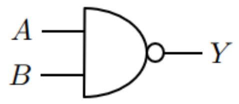
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate
The OR Gate takes 2 inputs and gives high if any one or more inputs are high, low only if all are low.



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

NAND Gate
The NAND Gate is the cascade of AND Gate followed by Inverter. Hence, it gives high if either one or all inputs are low.



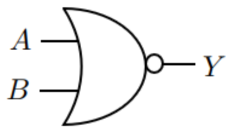
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

The boolean expression for the NAND function is given by,
$Q = \overline{A.B} \implies Q = \overline{A} + \overline{B}$

The NAND function can be realized by inverting inputs to the OR function. (This property will be useful in circuit simplification)

NOR Gate

The NOR Gate is the cascade of OR Gate followed by Inverter. Hence, it gives high only if all inputs are low.



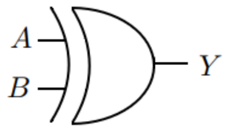
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

The boolean expression for the NOR function is given by,
$Q = \overline{A+B} \implies Q = \overline{A}.\overline{B}$

The NOR function can be realized by inverting inputs to the AND function. (This property will be useful in circuit simplification)

XOR Gate

The XOR Gate is a special gate which is modified version of OR gate. It gives high if the inputs are different, low if inputs are same.



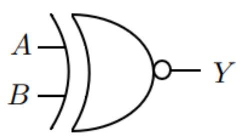
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

The boolean expression for the XOR function in terms of fundamental operations is given by,
$Q = A \oplus B = A.\overline{B} + \overline{A}.B$

The XOR function is used as a signal detector.
It is also used as even parity generator. Meaning, if number the input bit stream has odd number of 1s, the output will be 1 and if the input bit stream has even number of 1s, the output will be 0.

XNOR Gate

The XNOR Gate is a special gate which is the complement of XOR gate. It gives high if the inputs are same, low if inputs are different.



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

The boolean expression for the XNOR function in terms of fundamental operations is given by,
$Q = \overline{A \oplus B} = A.B + \overline{A}.\overline{B}$

The XNOR function is used as equality detector.
It is also used as odd parity generator. Meaning, if number the input bit stream has odd number of 1s, the output will be 0 and if the input bit stream has even number of 1s, the output will be 1.

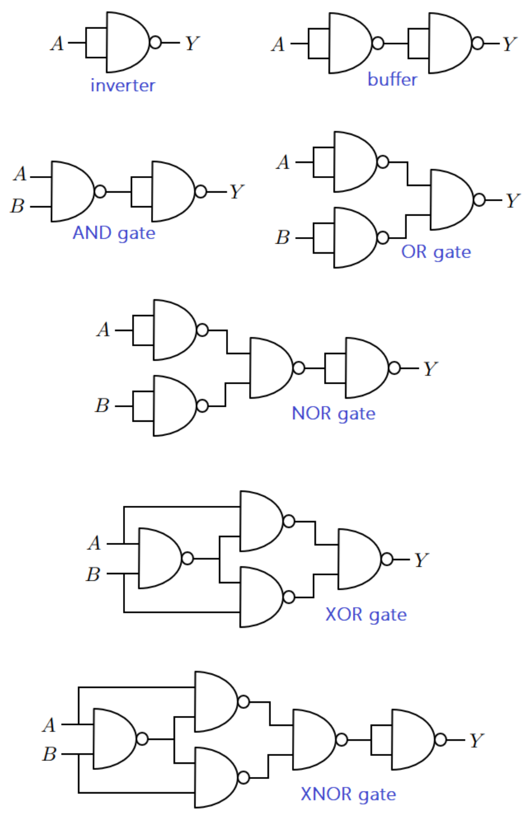
XOR and XNOR gates are called 'Special Gates' and these two are the only gates that are both compliments and duals of each other.

Propagation Delay:
The maximum time taken for the output of a gate to change after inputs are applied or after inputs change is called "Propagation Delay" (it is often denoted as t_{pd}).

Since logic gates are the basic building blocks of any digital circuit, when gates are cascaded with each other, the propagation delays get added up. This is why timing analysis is one of the most important aspects of digital design.

3.1.1 NAND only realization

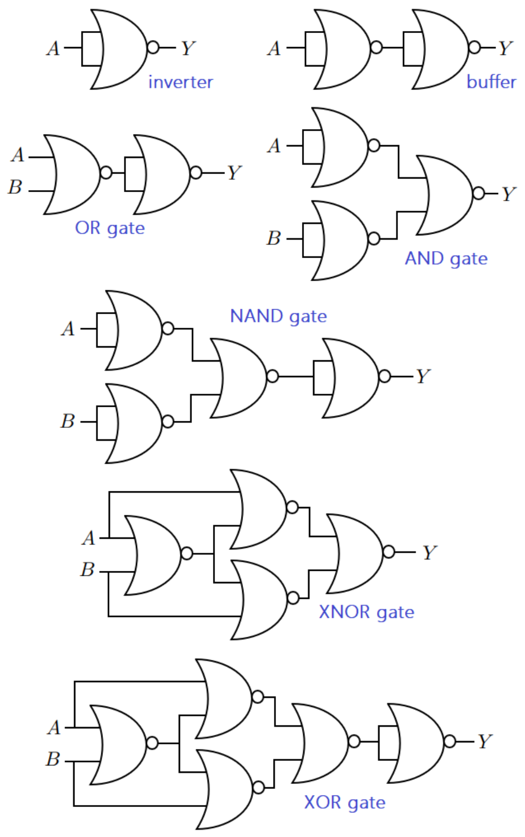
As mentioned earlier, NAND gate alone can be used to realize any boolean expression. The following figures show how the NAND gate is used to realize each of the other logic gates.



Realized logic Gate	Number of NAND Gates
Inverter	1
Buffer	2
AND	2
OR	3
NOR	4
XOR	4
XNOR	5

3.1.2 NOR only realization

As mentioned earlier, NOR gate alone can be used to realize any boolean expression. The following figures show how the NOR gate is used to realize each of the other logic gates.



Realized logic Gate	Number of NOR Gates
Inverter	1
Buffer	2
AND	3
OR	2
NAND	4
XOR	5
XNOR	4

3.1.3 Conversion

Since it has been proved that any logic function can be realized using either of the universal gates alone, it is useful to have methods to convert any given circuit consisting of basic gates to a circuit which uses only NAND gates or only NOR gates.

Conversion to NAND

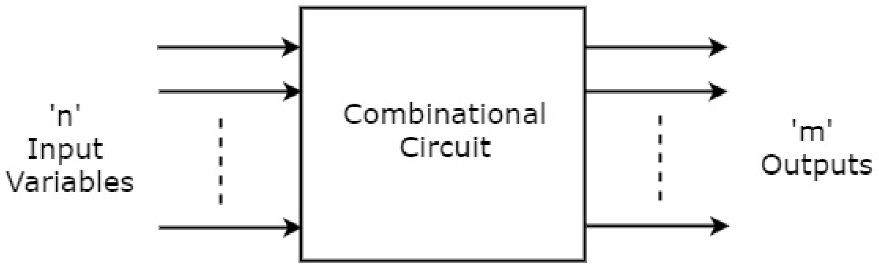
- Add two bubbles at the output of each AND gate.
- Add two bubbles at all inputs of each OR gate.
- Replace all input-bubbled OR gates and all output-bubbled AND gates with NAND gates.
- Remove all double bubbles, if any exist after replacement.
- Replace all remaining bubbles with input shorted NAND gates.

Conversion to NOR

- Add two bubbles at the output of each OR gate.
- Add two bubbles at all inputs of each AND gate.
- Replace all input-bubbled And gates and all output-bubbled OR gates with NOR gates.
- Remove all double bubbles, if any exist after replacement.
- Replace all remaining bubbles with input shorted NOR gates.

4 Combinational Circuits

Combinational circuits are circuits whose outputs depend only on the values of present inputs. Hence, combinational circuits do not use memory.



Combinational circuits are designed by obtaining the truth table and simplifying the outputs in terms of inputs using a technique such as K-map.

Combinational circuits can broadly be classified into 3 categories.

1. Arithmetic & Logic Circuits : Adders, Subtractors, Multipliers, Comparators.

2. Data Transmission Circuits (Multiplexers, Demultiplexers, Encoders, Decoders).

3. Code Converter Circuits

4.1 Arithmetic Circuits

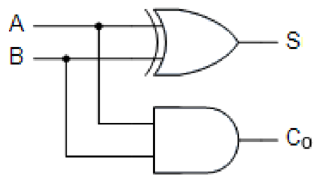
Arithmetic circuits are circuits that perform arithmetic operations. This subsection deals with only basic adders and subtractor circuits. Multipliers and advanced adders are elaborated in detail in "Digital IC Design".

4.1.1 Half Adder

A binary half adder is a circuit that takes two binary inputs (A and B) and gives two binary outputs which are sum S and carry C_o obtained by adding A and B.

A	B	S	C_o
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$S = A \oplus B \qquad\qquad C_o = A.B$$

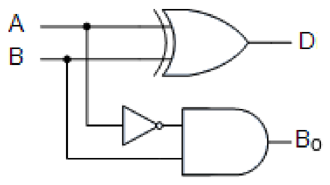


4.1.2 Half Subtractor

A binary half subtractor is a circuit that takes two binary inputs (A and B) and gives two binary outputs which are difference D and borrow B_o obtained by subtracting B from A.

A	B	D	B_o
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

$$D = A \oplus B \qquad\qquad B_o = \overline{A}.B$$

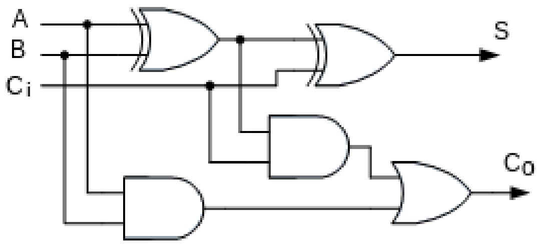


4.1.3 Full Adder

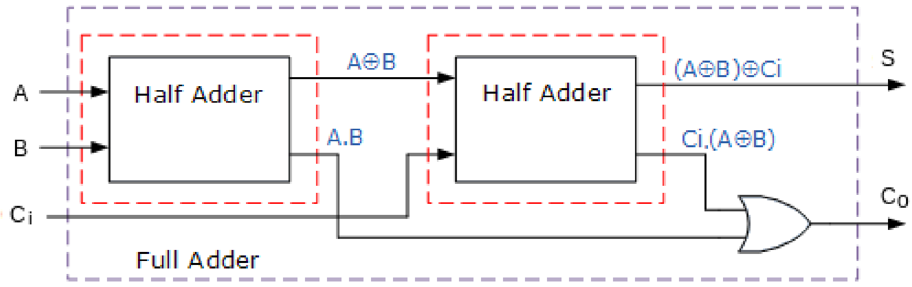
A binary full adder is a circuit that takes three binary inputs (A and B are the bits to be added and C_i is the carry bit generated from previous stage) and gives two binary outputs which are sum S and carry C_o obtained by adding A, B and C_i.

A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$S = A \oplus B \oplus C_i \qquad C_o = A.B + A.C_i + B.C_i = A.B + C_i.(A \oplus B)$$



Full Adder using 2 Half Adders:

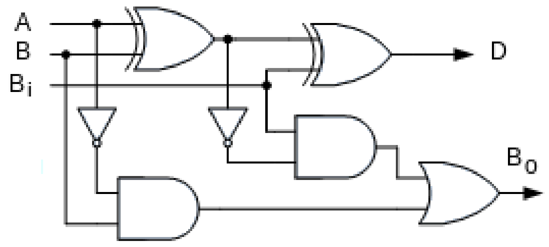


4.1.4 Full Subtractor

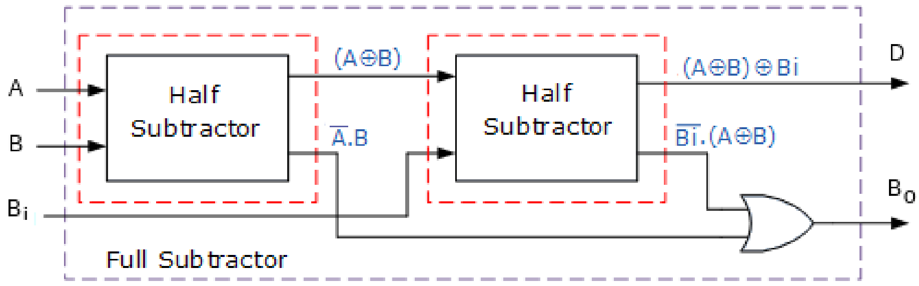
A binary full subtractor is a circuit that takes three binary inputs (B is to subtracted from A and B_i is the borrow bit generated from previous stage) and gives two binary outputs which are difference D and borrow B_o obtained by subtracting B from A with B_i.

A	B	B_i	D	B_o
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$D = A \oplus B \oplus B_i \qquad B_o = \overline{A}.B + \overline{A}.B_i + B.B_i = \overline{A}.B + B_i.\overline{(A \oplus B)}$$



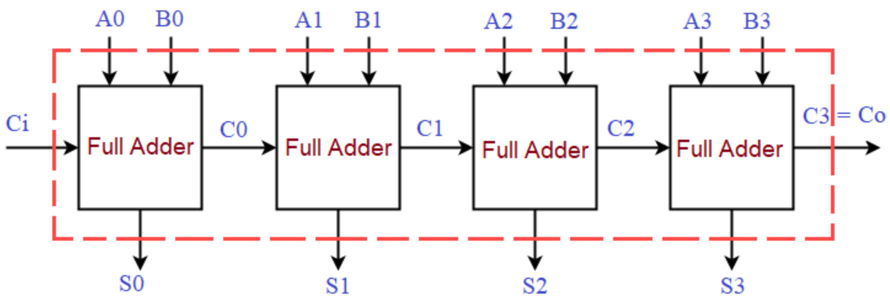
Full Subtractor using 2 Half Subtractors:



4.1.5 Ripple Carry Adder

Ripple carry adder or parallel binary adder is a cascade of n full adders (or it can be $n-1$ full adders and 1 half adder for first stage) in order to implement n bit binary addition.

4 bit ripple carry adder is illustrated.



Propagation delay in ripple carry adder:

Each gate used inside an adder will have some propagation delay. Hence, based on the inputs given, if the output bit has to change, the change will occur only after some propagation delay.

Consider an 'n' bit ripply carry adder and t_s is the sum delay for 1 stage and t_c is the carry delay for 1 stage.

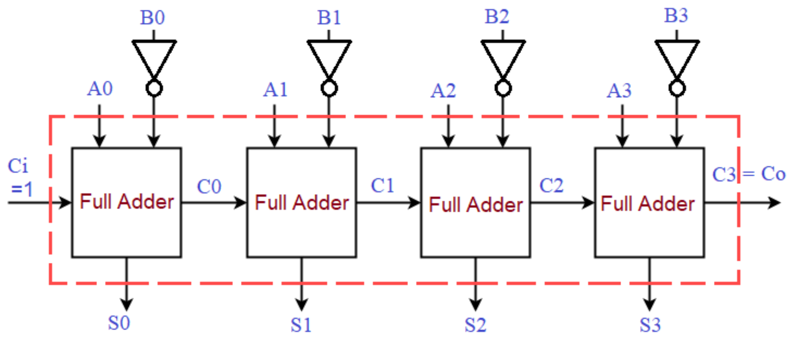
Best case delay: There is no carry propagation through any stage.
$\implies t_d = \max[t_s, t_c]$

Worst case delay: There is carry propagation through every stage.
$\implies t_d = \max[t_{sn}, t_{cn}]$
t_{sn} is the sum delay for n^{th} stage; $t_{sn} = t_s + (n-1)t_c$
t_{cn} is the carry delay for n^{th} stage; $t_{cn} = nt_c$

4.1.6 Ripple Carry Subtractor

Binary subtractor is not necessary to perform n-bit subtraction because the adder circuit itself can be used to perform subtraction using 2's compliment technique.

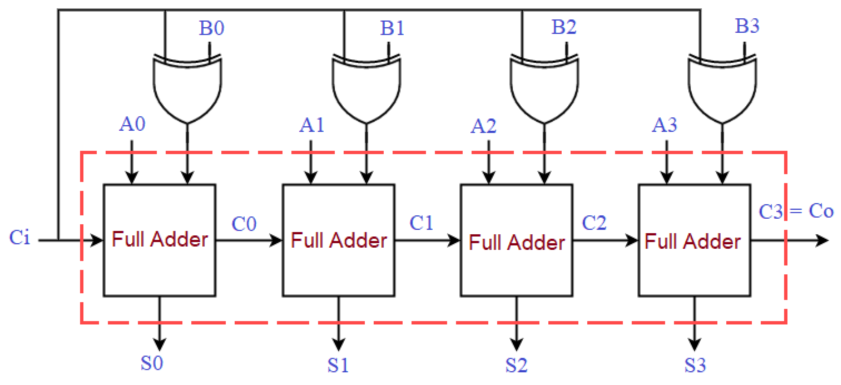
Since $A - B = A + 2$'s compliment$(B) = A + 1$'s compliment$(B) + 1$, the inputs to the adder circuit corresponding to B can be inverted and the carry input C_i can be permanently made 1. This will effectively perform subtraction operation.



4.1.7 Adder-Subtractor Circuit

The same adder circuit can be modified with some additional circuitry so that the common circuit can be used to perform both operations in different modes.
The first stage input carry C_i is always 0 in case of adder and always 1 in case of subtractor. Hence, it is evident that this input can also be used as a control input i.e $C_i = 0 \implies$ addition needs to be performed and$C_i = 1 \implies$ subtraction needs to be performed.

For addition, both $A[0:n]$ and $B[0:n]$ need to be fed to the adder unchanged. For subtraction, $B[0:n]$ needs to be complimented. Since $x \oplus 0 = x$ and $x \oplus 1 = \overline{x}$, the XOR gate can be used.
The inputs to the adder-subtractor circuit will be $A[0:n]$ unchanged and $B[0:n]$ XOR'ed with C_i.



It is evident that the main disadvantage of ripple carry adder is that every full adder has to wait for carry to be generated from previous full adder; which can cause very high delays when the number of bits is high (anything greater than $n = 4$ is not good).
To overcome the latency problem, there are several adder architectures that have been invented.

4.1.8 BCD Adder

BCD adder is used to operate on BCD numbers instead of binary numbers. Since each BCD code consists of 4 bits, BCD adder is implemented using 4-bit ripple carry adders.

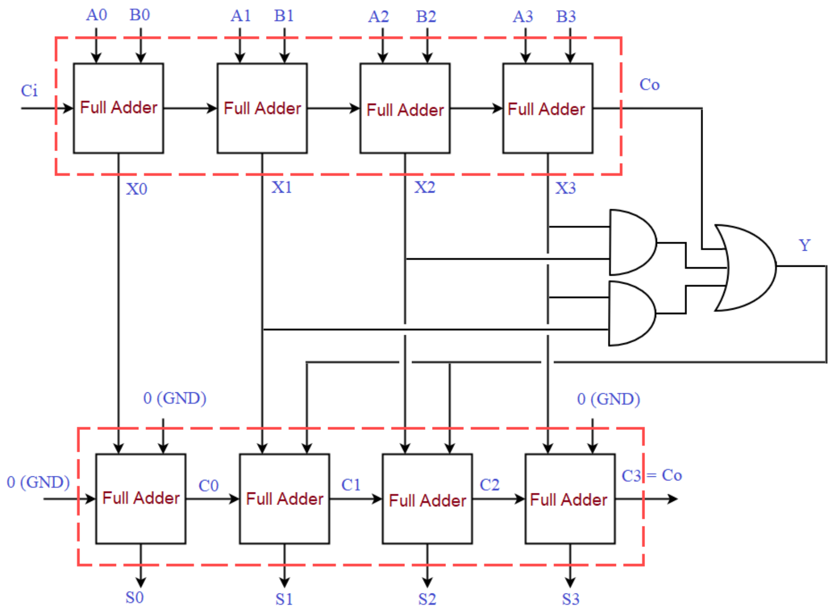
As mentioned earlier, there are only 10 valid combinations in BCD representation and each of them are equivalent to 4-bit binary representations of the numbers 0 to 9. Hence if the sum obtained by normal 4-bit binary addition is equal to one of these 10 combinations, it is same as the BCD sum. For example, if the two numbers to be added are 3 (0011) and 5 (0101), then the result is 8 (1000) which is a valid BCD code. So the BCD adder must output sum as 1000 with overflow set to 0.

If the sum obtained by normal 4-bit binary addition is not among these 10 combinations, then it means 4-bits are not enough to represent the sum i.e overflow must be generated. For example, if the two numbers to be added are 6 (0110) and 7 (0111), then the result is 13 (1101) which is not a valid BCD code. In BCD code, 13 is represented as 0001 0011. So the BCD adder must output the sum as 0011 with overflow set to 1.

The algorithm to implement BCD addition is as follows.

- Perform normal 4-bit binary addition.
- Check if the result is greater than 9 (1001).
- If result is not greater than 9, pass result as final sum, reset overflow.
- If result is greater than 9, subtract 10 from the result and make it the final sum, set overflow.

The simplified logic to identify if the 4-bit result (say $X_3X_2X_1X_3$ and C_o) is greater than 9 (1001) is $Y = X_3.X_2 + X_3.X_1 + C_o$. Hence if $Y = 1$, then 10 (1010) needs to be subtracted from the sum.
This is done using another 4-bit binary adder in subtractor configuration. The sum obtained from previous adder is added with 6 (0110) since 2's compliment of 1010 is 0101 + 1 = 0110.



Hence, the bits $S_3S_2S_1S_0$ represent the BCD sum and the bit Y represents the overflow (or BCD carry output).

4.1.9 Magnitude Comparator

The magnitude comparator circuit is a logic circuit compares two binary numbers and sets on of three possible outputs.
Let A and B are the two inputs (can be 'n' bits long). There will always be three outputs which are $G = A > B$, $E = A = B$ and $L = A < B$.

Consider 2 bit case ($A = A_1A_0$ and $B = B_1B_0$):

A_1	A_0	B_1	B_0	G	E	L
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

The output expressions can be derived using K-Map (or any method) as:
$E = (A_1 \oplus B_1).(A_0 \oplus B_0)$
$G = A_1.\overline{B_1} + (A_1 \oplus B_1).A_0.\overline{B_0}$
$L = \overline{A_1}.B_1 + (A_1 \oplus B_1).\overline{A_0}.B_0$

For 2 bit comparator, the total number of possible input combinations are $2^{2\times 2} = 16$. Out of these 16 combinations, 4 will have equal magnitude ($A = B$), 6 will have $A > B$ and 6 will have $A < B$.

This can be extended to $'n'$ bit case.
Total input combinations will be $N = 2^{2n}$ out of which $2n$ will have $A = B$, $(N - 2n)/2$ will have $A > B$ and $(N - 2n)/2$ will have $A < B$.

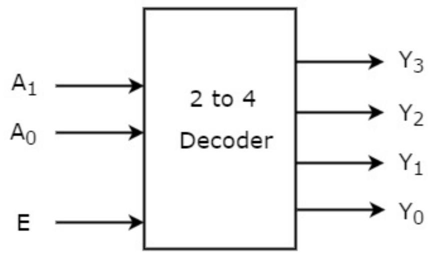
4.2 Data Transmission Circuits

Data transmission circuits are circuits that are used to manipulate signals while transmission or reception (used extensively in control unit).

4.2.1 Decoder

Decoder is a combinational circuit that has n input lines and maximum of 2^n output lines. One of these outputs will be activated based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code.

The most basic decoder is the 2×4 decoder. It consists of 2 inputs, an enable input and 4 outputs.



Truth table for 2×4 decoder (active high):

E	A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	$\times$	$\times$	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

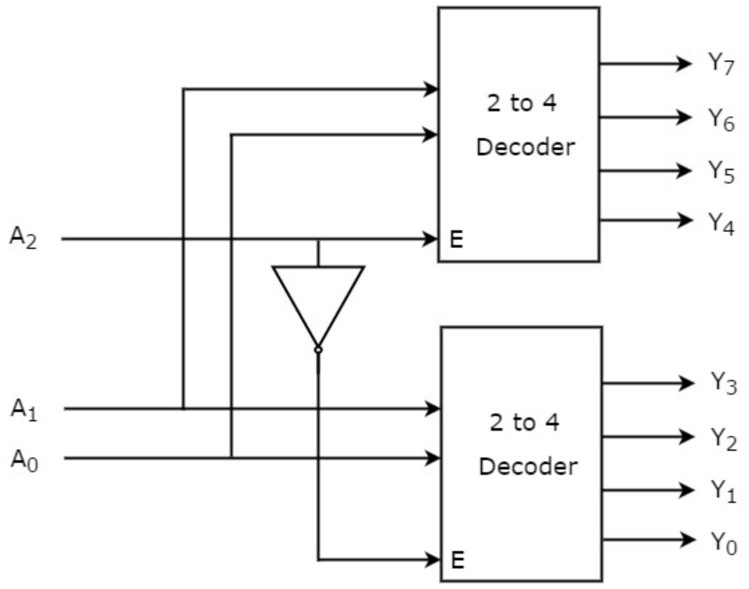
The outputs can be expressed as boolean functions of the inputs as follows.

$$Y_3 = E.A_1.A_0 \qquad Y_2 = E.A_1.\overline{A_0} \qquad Y_1 = E.\overline{A_1}.A_0 \qquad Y_0 = E.\overline{A_1}.\overline{A_0}$$

Higher order decoders such as 3×8 decoder and 4×16 decoder can be designed using similar logic.
However, it is usually more efficient to realize higher order decoders using a combination of lower order decoders.

3×8 **decoder using** 2×4 **decoders**:

Since 8 outputs are needed for 3×8 decoder, it is essential to have at least two 2×4 decoders.
Two select lines (A_1 and A_0) are directly fed to the select lines of the 2×4 decoders whereas the third select line (A_2) can be implemented using the enable input. If A_2 is low ($\overline{A_2} = 1$), then only the first decoder must be activated and if A_2 is high ($A_2 = 1$), then only the second decoder must be activated.

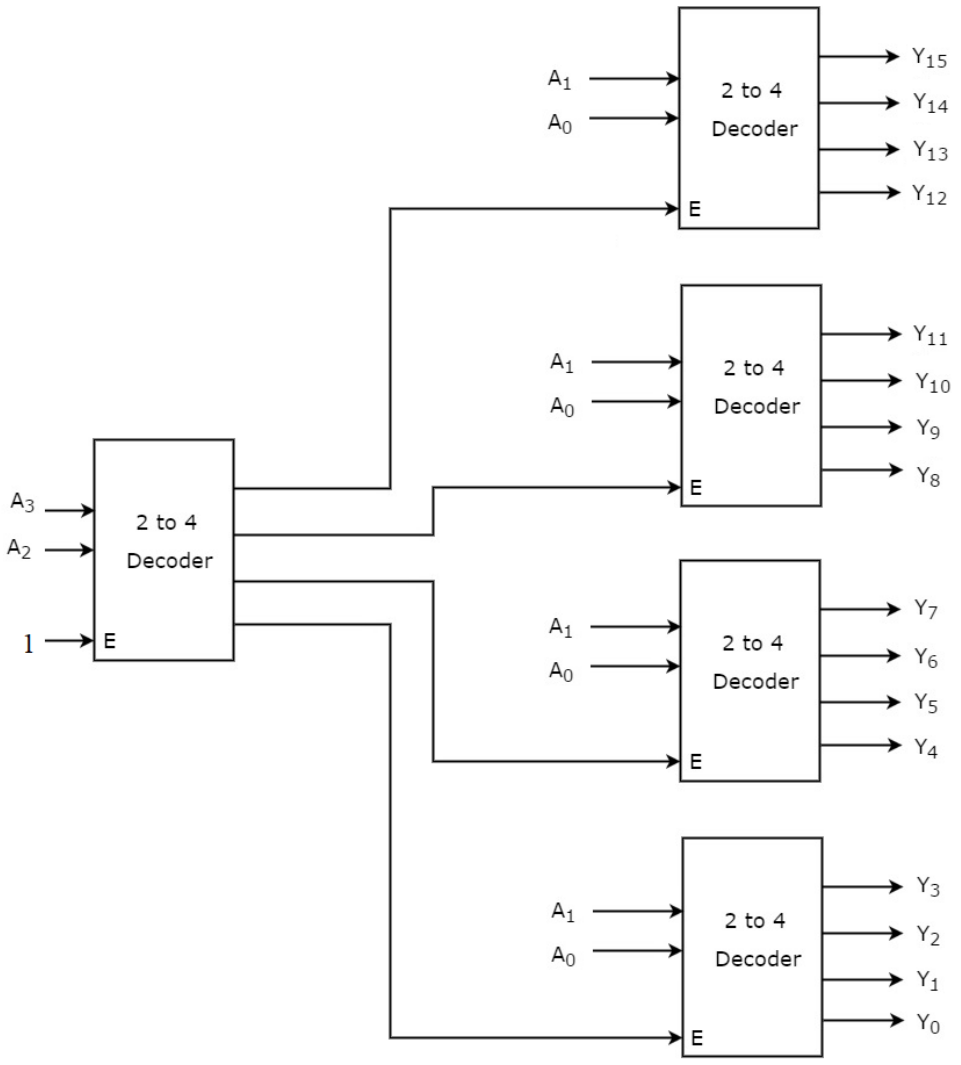


4×16 **decoder using** 2×4 **decoders**:

Since 16 outputs are needed for 4 decoder, it is essential to have at least four 2×4 decoders.
Two select lines (A_1 and A_0) are directly fed to the select lines of the 2×4 decoders whereas the other two select lines (A_3 and A_2) have to be given to an additional 2×4 decoder, whose outputs have to be fed to enable inputs of each of the four 2×4 decoder.

- First decoder must be enabled only when $\overline{A_3}.\overline{A_2} = 1$, so Y_0 of extra decoder is fed to enable of first decoder.

- Second decoder must be enabled only when $\overline{A_3}.A_2 = 1$, so Y_1 of extra decoder is fed to enable of second decoder.
- Third decoder must be enabled only when $A_3.\overline{A_2} = 1$, so Y_2 of extra decoder is fed to enable of third decoder.
- Fourth decoder must be enabled only when $A_3.A_2 = 1$, so Y_3 of extra decoder is fed to enable of fourth decoder.

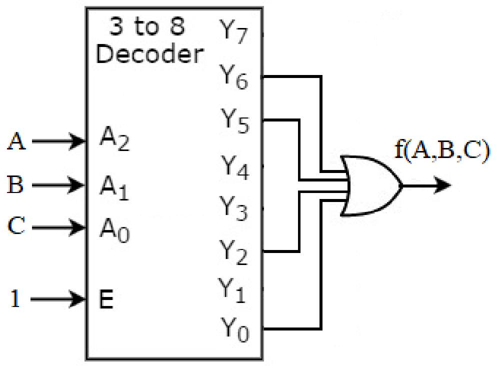


Realizing Boolean Functions using Decoders

Since each output of a decoder essentially represents a min-term, it can be used to implement any boolean function of the given inputs. Before going into that, it must be noted that a decoder can have active low inputs and outputs as well.

For example, consider implementing the given function using 3×8 decoder.
$f(A, B, C) = \overline{A}.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.\overline{B}.C + A.B.\overline{C} = \sum m(0, 2, 5, 6)$

This boolean function is realized as shown using an OR gate if the decoder consists of active high inputs and active high outputs.



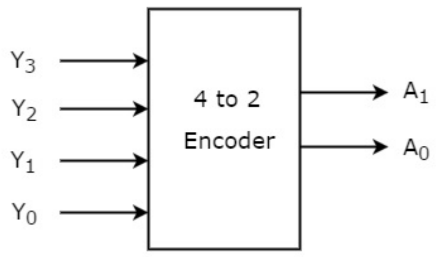
If the decoder had active high inputs but active low outputs, then the OR gate would have to be replaced with a NAND gate (since NAND gate with bubbled inputs is same as OR gate).

The same expression can be expressed as $f(A, B, C) = \Pi M(1, 3, 4, 7)$. Hence, it can be realized using a bubbled AND gate (i.e a NOR gate) if outputs are active high and just an AND gate if outputs are active low.

4.2.2 Encoder

Encoder is a combinational circuit that performs the reverse operation of a decoder. It takes 2^n inputs where at most only one of them will ever be high and has n output bits that are set based on which of the 2^n inputs are activated. Such an encoder is also called "**Simple Encoder**".

A 4×2 encoder is illustrated.



Truth table for 4×2 simple encoder:

Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	0	$\times$	$\times$
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

The boolean expressions for the outputs of simple encoder will be as follows.

$$A_1 = Y_3 + Y_2 \qquad A_0 = Y_3 + Y_1$$

Evidently, the problem with a simple encoder is that it doesn't have defined responses in case more than one of the input bits are activated.
In order to overcome that disadvantage, a modified encoder called "**Priority Encoder**" is used.

Truth table for 4×2 priority encoder:

Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	0	$\times$	$\times$
0	0	0	1	0	0
0	0	1	$\times$	0	1
0	1	$\times$	$\times$	1	0
1	$\times$	$\times$	$\times$	1	1

The boolean expressions for the outputs of priority encoder will be as follows.

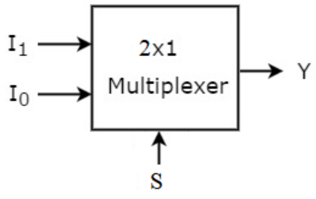
$$A_1 = Y_3 + Y_2 \qquad A_0 = Y_3 + \overline{Y_2}.Y_1$$

Higher order encoders can be implemented using similar logic.

4.2.3 Multiplexer

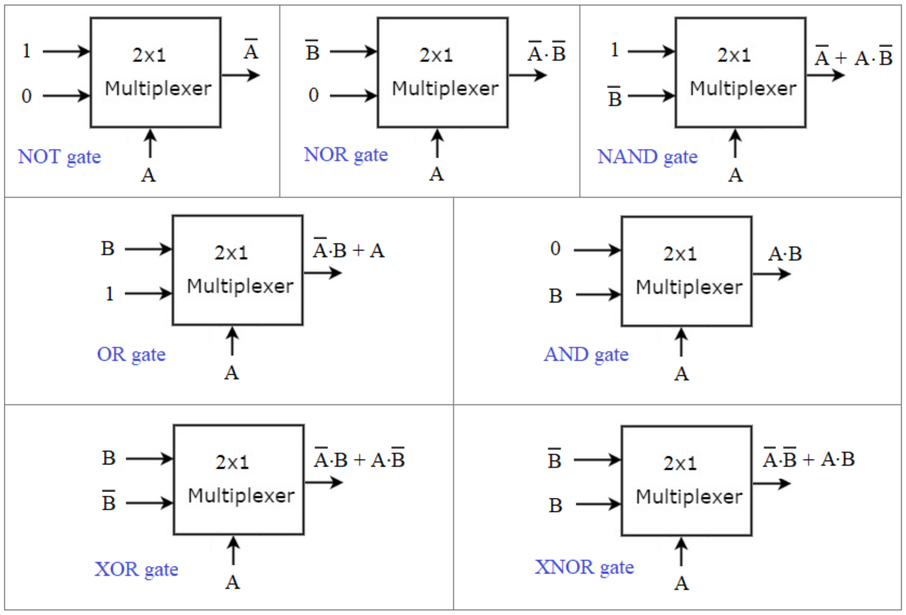
Multiplexer (Mux in short) is a combinational circuit that selects one of the 2^n data inputs based on the pattern at n select lines. Hence, a mux is also called as a data selector.

The simplest mux is a 2×1 mux which has two data inputs (I_0 and I_1) and a single select line (S). The output for the mux is given by $Y = \overline{S}.I_0 + S.I_1$



A mux can be realized using transmission gate logic instead of using logic gates. Hence, it can be considered as a basic logic circuit.

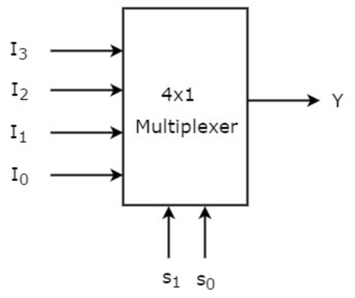
A 2×1 mux can be used to realize any of the basic gates.



1, 0, 2x1 Multiplexer, $\overline{A}$, A, NOT gate
$\overline{B}$, 0, 2x1 Multiplexer, $\overline{A}\cdot\overline{B}$, A, NOR gate
1, $\overline{B}$, 2x1 Multiplexer, $\overline{A} + A\cdot\overline{B}$, A, NAND gate
B, 1, 2x1 Multiplexer, $\overline{A}\cdot B + A$, A, OR gate
0, B, 2x1 Multiplexer, $A\cdot B$, A, AND gate
B, $\overline{B}$, 2x1 Multiplexer, $\overline{A}\cdot B + A\cdot\overline{B}$, A, XOR gate
$\overline{B}$, B, 2x1 Multiplexer, $\overline{A}\cdot\overline{B} + A\cdot B$, A, XNOR gate

It is evident from the above illustrations that a mux can be used to implement arbitrary boolean functions in general.

Consider the 4×1 multiplexer shown.



Truth table for the 4×1 multiplexer:

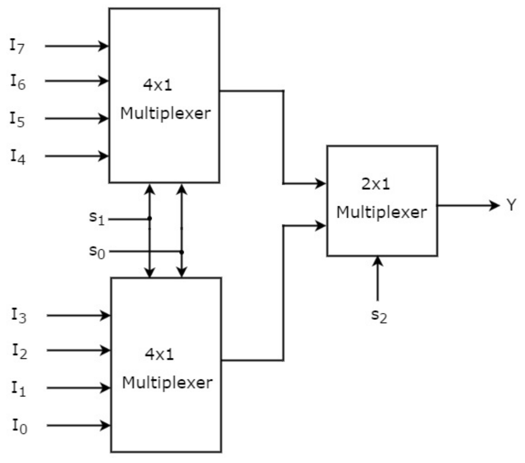
s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

The boolean function for the output of the 4×1 multiplexer is obtained as

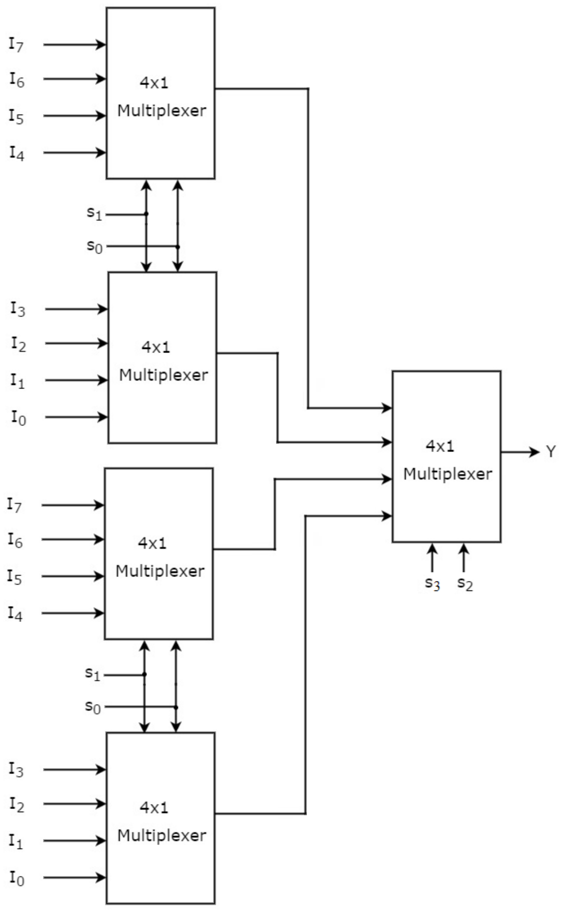
$$Y = \overline{s_1}.\overline{s_0}.I_0 + \overline{s_1}.s_0.I_1 + s_1.\overline{s_0}.I_2 + s_1.s_0.I_3$$

Higher order multiplexers can be realized using similar logic or by combinations of lower order multiplexers.

8×1 multiplexer is realized using two 4×1 multiplexers in first stage, followed by a 2×1 multiplexer.



16×1 multiplexer is realized using four 4×1 multiplexers in first stage, followed by another 4×1 multiplexer.



As mentioned earlier, a mux can be used to realize boolean expressions.

Consider realizing a boolean function of 3 variables using an 8×1 mux. $Y = f(A, B, C) = A.B + B.C + A.C$. If expressed in standard SoP form, it will be $Y = A.B.C + A.B.\overline{C} + \overline{A}.B.C + A.\overline{B}.C = \sum m(3, 5, 6, 7)$.

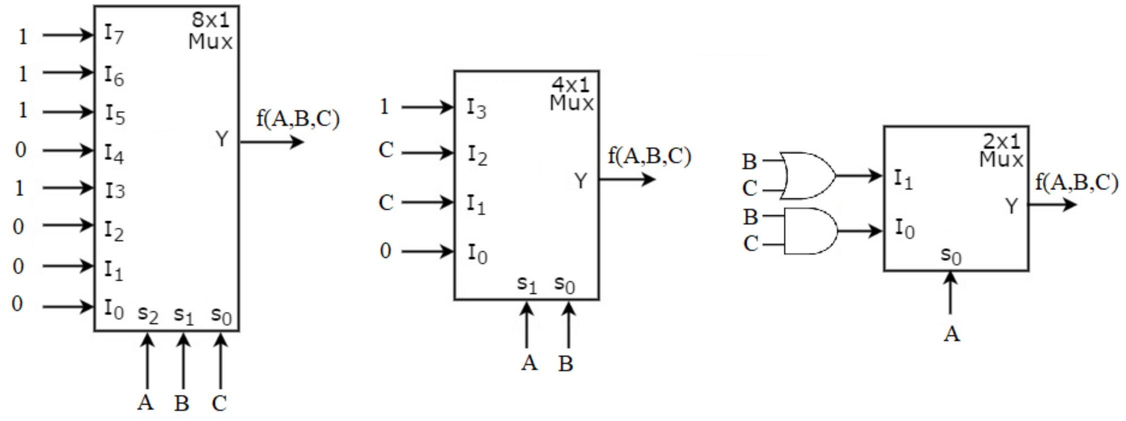
Hence, the select lines can be connected to the function inputs A, B and C while the data inputs to the 8×1 mux corresponding to the minterms 3,5,6,7 should be made high and the rest should be made low.

The same function can also be realized using a 4×1 mux. Two inputs (say A

and B) are fed to the two select lines. The data inputs are fed with functions of C (i.e 0,1,C or $\overline{C}$) as required.

If additional logic gates are allowed, then the same function can be realized using 2×1 mux as well. An input (say A) is fed to the select line. The data inputs are fed with functions of B and C as required.

Realization of the same function $Y = f(A, B, C)$ using 8×1 mux, 4×1 mux and 2×1 mux (with extra gates):



4.2.4 De-multiplexer

De-multiplexer (Demux in short) is a combinational circuit that passes the given data input to one of the 2^n output lines based on the pattern at n select lines. Hence, a demux is also called as a data distributor.

A de-multiplexer circuit is essentially the same as a decoder circuit. The enable input of the decoder can be thought of as the data input of the de-multiplexer, and the n inputs of the decoder can be thought of as the n select lines of the de-multiplexer.

Consider a 1×4 de-multiplexer with data input I, select lines s_0, s_1 and outputs Y_0, Y_1, Y_2 and Y_3.
The boolean expressions for the outputs in terms of the inputs will be

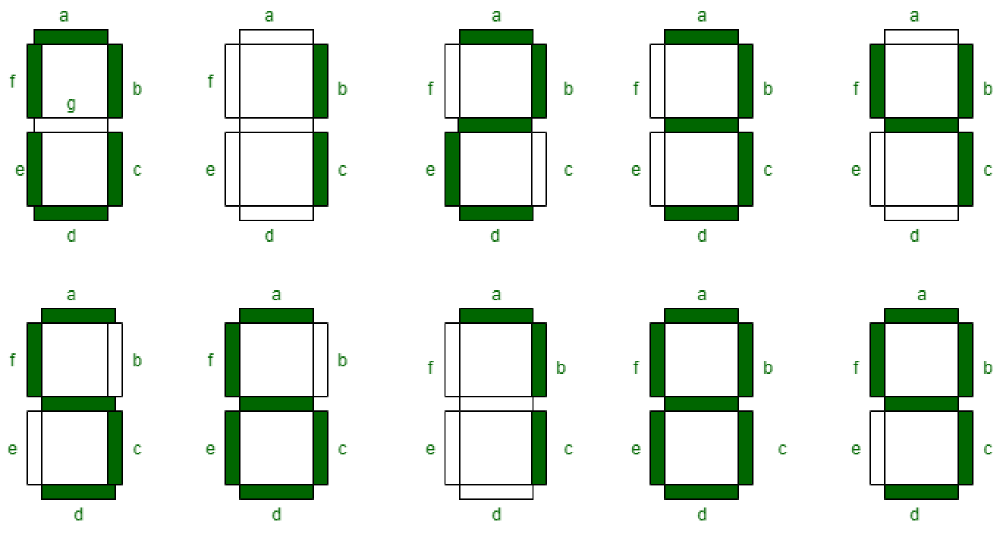
$$Y_3 = s_1.s_0.I \qquad Y_2 = s_1.\overline{s_0}.I \qquad Y_1 = \overline{s_1}.s_0.I \qquad Y_0 = \overline{s_1}.\overline{s_0}.I$$

4.3 Code Converters

A code converter circuit will convert coded information in one form to a different coding form. Some examples of code converters are BCD to XS3 converter, Binary to Grey converter, Grey to Binary converter, Binary to BCD converter, etc.

Design procedure for a code converter is same as any combinational circuit. The best example that can be used to illustration is binary to 7 segment display conversion.

7 segment display:
A 7 segment display is an output display device that provides a simple way to display information (usually a number from 0 to 9).
It consists of 7 LEDs that have to be controlled by the inputs to display what is needed. The following figure illustrates how a 7 segment display shows each of the 10 digits.



A code converter circuit has to be used to generate signals that can drive the 7 segment display from usual binary code.

Note that there will be 4 input bits ($b_3\, b_2\, b_1\, b_0$) and 7 output bits ($a\, b\, c\, d\, e\, f\, g$). Each of the 7 output bits have to be expressed as a boolean function of 4 the input bits.

The following truth table translates the given figure, i.e it tells which of the 7 outputs must be on (or off) for each of the 10 binary input combinations.

b_3	b_2	b_1	b_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Using K-map, the simplified boolean expression for each of the outputs in terms of inputs can be found as follows.

$\rightarrow a = \overline{b_2}.\overline{b_0} + b_2.b_0 + b_1 + b_3$

$\rightarrow b = \overline{b_1}.\overline{b_0} + b_1.b_0 + \overline{b_2}$

$\rightarrow c = b_2 + \overline{b_1} + b_0$

$\rightarrow d = \overline{b_2}.\overline{b_0} + \overline{b_2}.b_1 + b_2.\overline{b_1}.b_0 + b_1.\overline{b_0} + b_3$

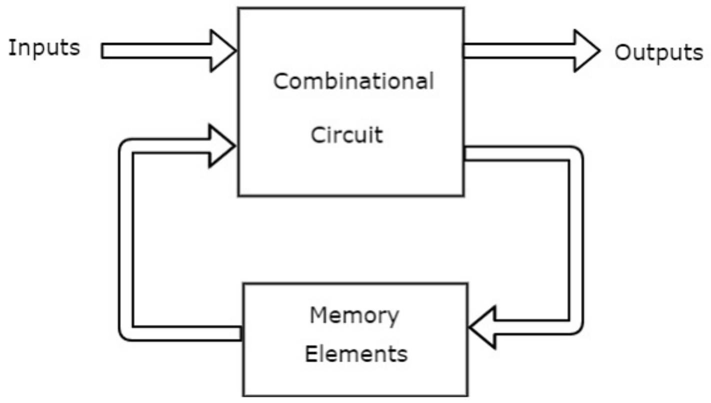
$\rightarrow e = \overline{b_2}.\overline{b_0} + b_1.\overline{b_0}$

$\rightarrow f = \overline{b_1}.\overline{b_0} + b_2.\overline{b_1} + b_2.\overline{b_0} + b_3$

$\rightarrow g = b_2.\overline{b_1} + \overline{b_2}.b_1 + b_2.\overline{b_0} + b_3$

5 Sequential Circuits

Sequential circuits are circuits whose outputs depend on values of present inputs as well as past outputs. Hence, sequential circuits are combinational circuits along with memory elements.



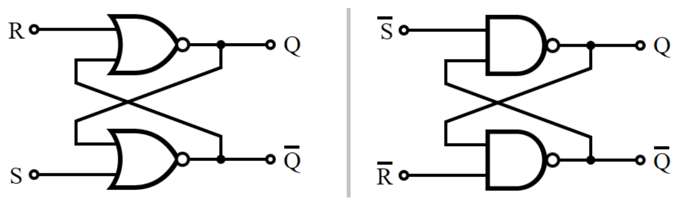
5.1 Latches

A latch is a bistable multi-vibrator i.e, a device with exactly two stable states. These states are high-output and low-output. A latch has a feedback path, so information can be retained by the device. Therefore, latches can be memory devices and can store one bit of data for as long as the device is powered.

A latch is an asynchronous circuit and hence the outputs can change as soon as the inputs do (after a small propagation delay).

5.1.1 S-R Latch

The most basic latch is an S-R latch that is obtained by using cross coupled NOR gates (or NAND gates with complimented inputs).



- When the inputs are $S = 0$ and $R = 1$, the output $Q = 0$ and the latch is said to be ”reset”.
- When the inputs are $S = 1$ and $R = 0$, the output $Q = 1$ and the latch is said to be ”set”.
- When the inputs $S = 0$ and $R = 0$, the output will retain itself from previous state; hence this is the memory operation of the latch.

When the inputs $S = 1$ and $R = 1$, there is a problem. It can be observed that the latch is being told to simultaneously produce $Q = 0$ and $Q = 1$. Practically, whichever flip flop succeeds in changing first will feedback to the other and assert itself. But there is no way to predict this and hence, this configuration is ”illegal”.

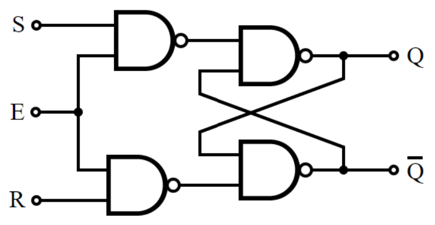
Characteristic table defining the operation of S-R latch:

S	R	Q^+	$\overline{Q}^+$
0	0	Q	$\overline{Q}$
0	1	0	1
1	0	1	1
1	1	1*	1*

5.1.2 Gated S-R Latch

A gated S-R latch is a modified S-R latch with an enable input. The enable input just gives more control for the operation of the S-R latch.

Gated S-R latch using NAND gates is shown.



Note that the same circuit can be realized using cross coupled NOR gates from normal S-R latch and AND gates to incorporate the enable input.

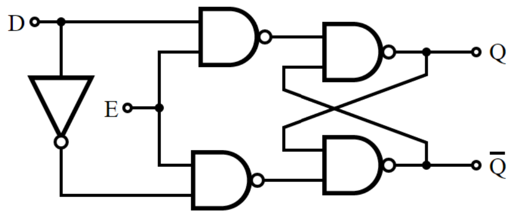
The functioning is evidently exactly the same as the S-R latch except that it works only when the enable input is high ($E = 1$). When the enable input is low ($E = 0$), the latch does not respond to any changes in R and S, and the output Q remains in previous state (like memory operation).

Characteristic table defining the operation of gated S-R latch:

E	S	R	Q^+	$\overline{Q}^+$
0	$\times$	$\times$	Q	$\overline{Q}$
1	0	0	Q	$\overline{Q}$
1	0	1	0	1
1	1	0	1	1
1	1	1	1*	1*

5.1.3 D Latch

D latch (or Gated D latch) is an extension of the gated S-R latch which eliminates the possibility of invalid inputs (i.e $S = R = 1$).
This is accomplished by connecting $S = D$ and $R = \overline{D}$. Hence, S and R can never be equal. But this circuit still gives provision for memory operation due to presence of enable input E.



- If enable is high ($E = 1$), then the output follows the input ($Q = D$).
- If enable is low ($E = 0$), then the output is retained from previous state.

Characteristic table defining the operation of D latch:

E	D	Q^+
0	$\times$	Q
1	0	0
1	1	1

5.2 Flip-Flops

A flip-flop, like a latch is a bistable multi-vibrator, having two states and a feedback path that allows it to store a bit of information.
A flip-flop on the other hand, always runs on a "clock signal" and is edge-triggered i.e only changes state when the clock signal goes from high to low or low to high.

Note that a clock signal is simply a stream pulses changing from high to low and low to high at a certain frequency.

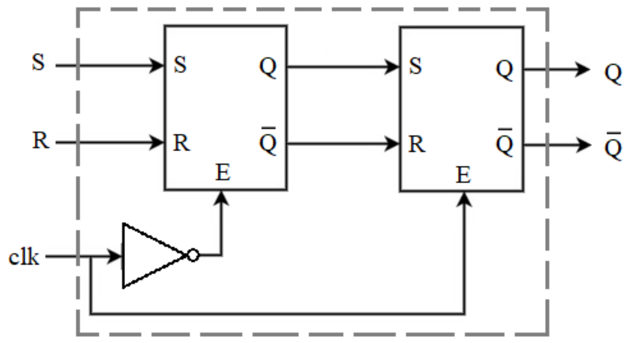
All flip-flops are either triggered by every positive edge of the clock or by every negative edge of the clock. Here, positive edge triggered flip-flops are illustrated, but apart from sensitivity to one type of triggering edge, there is no change in the operations between the two types of flip-flops.

Flip-flops are designed using Latches in "Master-Slave" configuration.

5.2.1 S-R Flip-Flop

The operation of an S-R flip-flop is similar to that for a gated S-R latch, except that it only changes state at every rising edge (or falling edge).

A positive edge triggered S-R flip-flop is designed using two gated S-R latches in master-slave configuration as shown.



(If *clk* and $\overline{clk}$ are interchanged, the obtained flip-flop will be negative edge triggered S-R flip-flop)

The working of a positive edge triggered S-R flip-flop is explained.
During every rising edge of clock,

- If $S = 0$ and $R = 0$, then the output will remain same as previous i.e 'memory state'
- If $S = 1$ and $R = 0$, then the output $Q^+ = 1$ i.e 'set state'
- If $S = 0$ and $R = 1$, then the output $Q^+ = 0$ i.e 'reset state'
- If $S = 1$ and $R = 1$, then the output is in-determinant. This is because different results will be obtained based on propagation delays of the gates. Due to this unpredictability, this configuration is not used.

At all other times, the output will remain as it is.

Characteristic table of S-R flip-flop:

CLK	S	R	Q^+	$\overline{Q}^+$
-	$\times$	$\times$	Q	$\overline{Q}$
$\uparrow$	0	0	Q	$\overline{Q}$
$\uparrow$	0	1	0	1
$\uparrow$	1	0	1	1
$\uparrow$	1	1	1*	1*

By treating the present output Q as one of the inputs, the next output Q^+ can be expressed as follows.

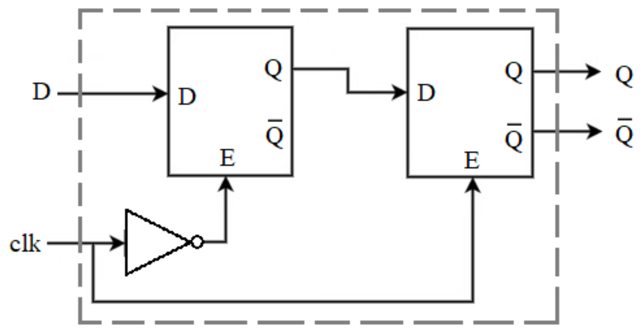
$$Q^+ = S + Q.\overline{R}$$

This is the characteristic equation of the S-R flip-flop.

5.2.2 D Flip-Flop

The logic behind the utility of D flip-flop is same as of D latch. It simply eliminates the possibility of occurrence of invalid inputs. The data at the input is simply captured by the flip-flop at every rising (or falling) edge of the clock pulse.

A positive edge triggered D flip-flop is designed using two D latches in master-slave configuration as shown.



(If clk and $\overline{clk}$ are interchanged, the obtained flip-flop will be negative edge triggered D flip-flop)

Characteristic table of D flip-flop:

CLK	D	Q^+
-	$\times$	Q
$\uparrow$	0	0
$\uparrow$	1	1

The working of a positive edge triggered D flip-flop is explained.
During every positive edge of the clock,

- If $D = 0$, then the output $Q^+ = 0$.
- If $D = 1$, then the output $Q^+ = 1$.

At all other times, the output will remain as it is.

Hence, the output follows the input. Which is why this is called "Data Flip-flop". Since the output is same as input with a delay of 1 stage, it is also called "Delay Flip-flop".

By treating the present output Q as one of the inputs, the next output Q^+ can be expressed as follows.

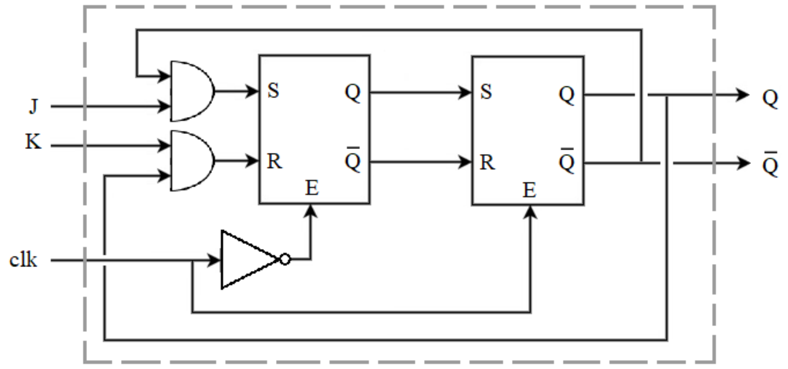
$$Q^+ = D$$

This is the characteristic equation of the D flip-flop.

5.2.3 J-K Flip-Flop

J-K flip-flop is a modified or improved version of S-R flip-flop that introduces utility to the condition where both inputs are 1.

This is done by feeding back the outputs of the S-R flip-flop into the inputs as illustrated. The given circuit is a positive edge triggered J-K flip-flop.



(If *clk* and $\overline{clk}$ are interchanged, the obtained flip-flop will be negative edge triggered J-K flip-flop)

Due to the feedback mechanism, it can be observed that when the inputs are $J = 1$ and $K = 1$, the output will toggle (at positive edge of the clock).

J-K Latch:
The same feedback logic can be used to design a J-K latch as well.
However, J-K latch will not exactly eliminate the problem of invalid input combination. This is because since the latch is level triggered, the outputs will keep toggling endlessly when enable and both inputs are high. This is called "race around condition". The outputs settle only when the enable is removed (made low) or one of the inputs change. But the toggling itself is not useful since it happens too fast and is impossible to predict at which state it will settle to when enable is removed.

This is why J-K latch is not commonly used. But J-K flip-flop solves this problem since it is edge triggered. The outputs toggle just once during the clock edge and race around condition does not occur.

Characteristic table of J-K flip-flop:

CLK	J	K	Q^+	$\overline{Q}^+$
-	$\times$	$\times$	Q	$\overline{Q}$
$\uparrow$	0	0	Q	$\overline{Q}$
$\uparrow$	0	1	0	1
$\uparrow$	1	0	1	1
$\uparrow$	1	1	$\overline{Q}$	Q

The working of a positive edge triggered J-K flip-flop is explained.
During every rising edge of clock,

- If $J = 0$ and $K = 0$, then the output will remain same as previous i.e 'memory state'
- If $J = 1$ and $K = 0$, then the output $Q^+ = 1$ i.e 'set state'
- If $J = 0$ and $K = 1$, then the output $Q^+ = 0$ i.e 'reset state'
- If $J = 1$ and $K = 1$, then the output $Q^+ = \overline{Q}$ i.e 'toggle state'

At all other times, the output will remain as it is.

By treating the present output Q as one of the inputs, the next output Q^+ can be expressed as follows.

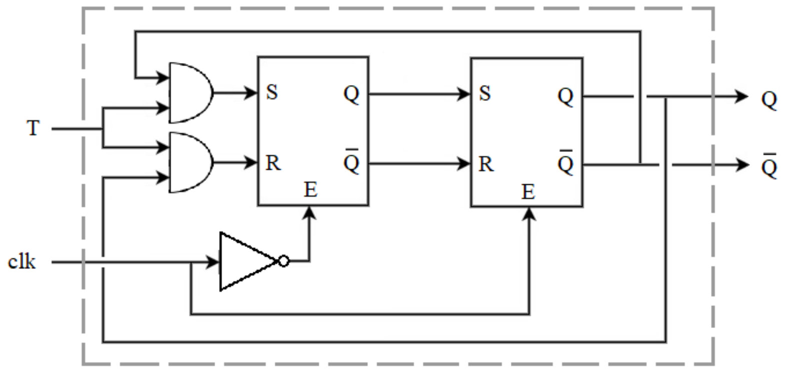
$$Q^+ = J.\overline{Q} + \overline{J}.Q$$

This is the characteristic equation of the J-K flip-flop.

5.2.4 T Flip-Flop

In a J-K latch, if the J and K inputs are both shorted and connected to input T, the resultant latch will have only two modes: memory state and toggle state. This is called a **T latch**. However the race around condition occurs in toggle state, which is why it can't be used.

The flip-flop corresponding to the T latch realized using master-slave configuration is called a T flip-flop. In a T flip-flop, the toggle state can be used correctly since it is edge triggered and race around condition is eliminated. The positive edge triggered T flip-flop is illustrated.



(If clk and $\overline{clk}$ are interchanged, the obtained flip-flop will be negative edge triggered T flip-flop)

Characteristic table of T flip-flop:

CLK	T	Q^+
-	$\times$	Q
$\uparrow$	0	Q
$\uparrow$	1	$\overline{Q}$

The working of a positive edge triggered T flip-flop is explained.
During every positive edge of the clock,

- If $T = 0$, then the output $Q^+ = Q$.
- If $T = 1$, then the output $Q^+ = \overline{Q}$.

At all other times, the output will remain as it is.

Hence, the output toggles if the input T is set (high) and does not toggle if the input T is reset (low). Which is why this is called "Toggle Flip-flop".

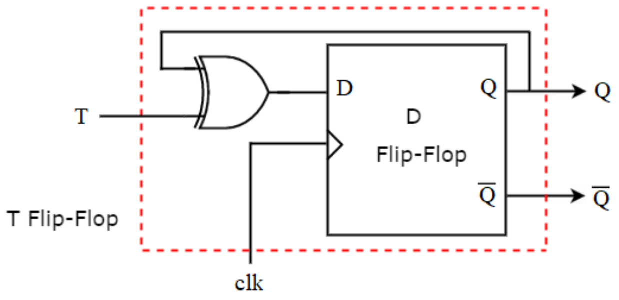
By treating the present output Q as one of the inputs, the next output Q^+ can be expressed as follows.

$$Q^+ = T \oplus Q$$

This is the characteristic equation of the T flip-flop.

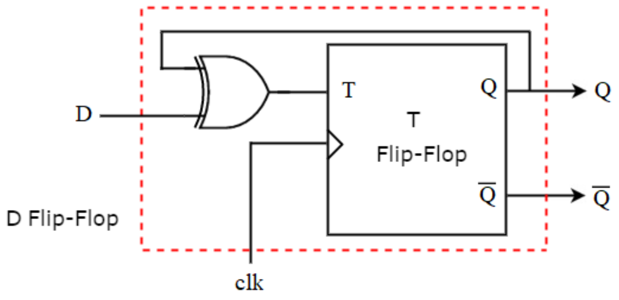
Realizing T flip-flop using D flip-flop:

Available function: $Q^+ = D$
Required function: $Q^+ = Q \oplus T$
If $D = Q \oplus T$, then the required function is realized using the available function.



Realizing D flip-flop using T flip-flop:

Available function: $Q^+ = Q \oplus T$
Required function: $Q^+ = D$
If $T = D \oplus Q$, then $Q^+ = Q \oplus (D \oplus Q) = D \oplus (Q \oplus Q) = D \oplus 0 = D$.
Hence the required function is realized using the available function.



Flip-flops that are used in practice generally are equipped with two control inputs.

- "Preset" or "Set" input which when activated makes the flip-flop output to 1 irrespective of what the data inputs are.
- "Clear" or "Reset" input which when activated makes the flip-flop output to 0 irrespective of what the data inputs are.

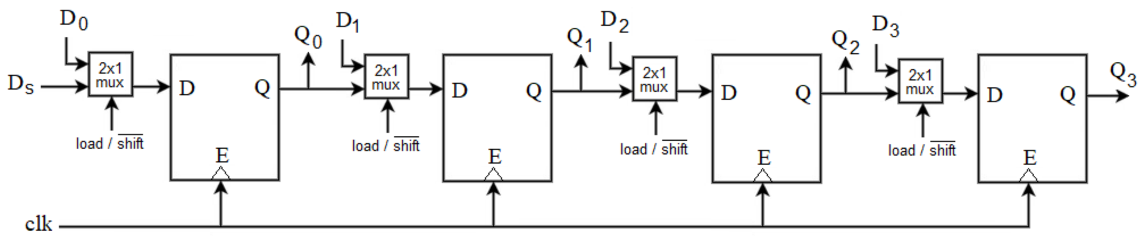
5.3 Shift Registers

Shift register is a digital circuit made using a cascade of flip flops where the output of one flip-flop is connected to the input of the next. They share a single clock signal, which causes the data stored in the system to shift from one location to the next.

Shift registers operate in one of four different modes with the basic movement of data. They are:

- Serial-in to Parallel-out (SIPO) - Register is loaded with serial data one bit at a time, with the stored data being available at the output in parallel form.
- Serial-in to Serial-out (SISO) - Data is shifted serially in and out of the register one bit at a time in either a left or right direction under clock control.
- Parallel-in to Serial-out (PISO) - Data is loaded into the register simultaneously and is shifted out of the register serially one bit at a time under clock control.
- Parallel-in to Parallel-out (PIPO) - Data is loaded into the register simultaneously and transferred together to their respective outputs by the same clock pulse.

The flip-flop that is used to implement a shift register is obviously a D flip-flop. The figure shown below is that of a general 4-bit shift register that can be used in any of the 4 configurations (or modes) mentioned.



Using the $load/\overline{shift}$ control signal, this shift register can be operated in any of the 4 modes.

- When $load/\overline{shift} = 1$, the parallel inputs D_0, D_1, D_2, D_3 are stored in the register and they can be accessed at Q_0, Q_1, Q_2, Q_3 respectively.
- When $load/\overline{shift} = 0$, the serial input D_s is shifted into the register and all data previously stored in the flip-flops get shifted to the next flip-flops (final bit is lost). After 4 clock cycles, D_s will be available for access at Q_3.

Note that if the output of the last D flip-flop is connected as to the serial input of first D flip-flop, then the shift register can be used to perform rotate operation.

5.4 Counters

Counter is a sequential circuit which is used for a counting pulses. A counter is made from a group of flip-flops and additional combinational logic if required. It is the widest application of flip-flops.

A counter can be of two types:

- Asynchronous Counter
- Synchronous Counter

5.4.1 Asynchronous Counters

Asynchronous counters are made of flip-flops where all the flip-flops are not driven by the same clock pulse. Instead, each flip-flop is driven by the output of the flip-flop preceding it and only the first flip-flop is driven by the clock. They are also called "Ripple Counters".

- **Up Counter**: An N-bit up counter counts from 0 to $2^N - 1$ (and repeats the pattern).
- **Down Counter**: An N-bit down counter counts from $2^N - 1$ to 0 (and repeats the pattern).

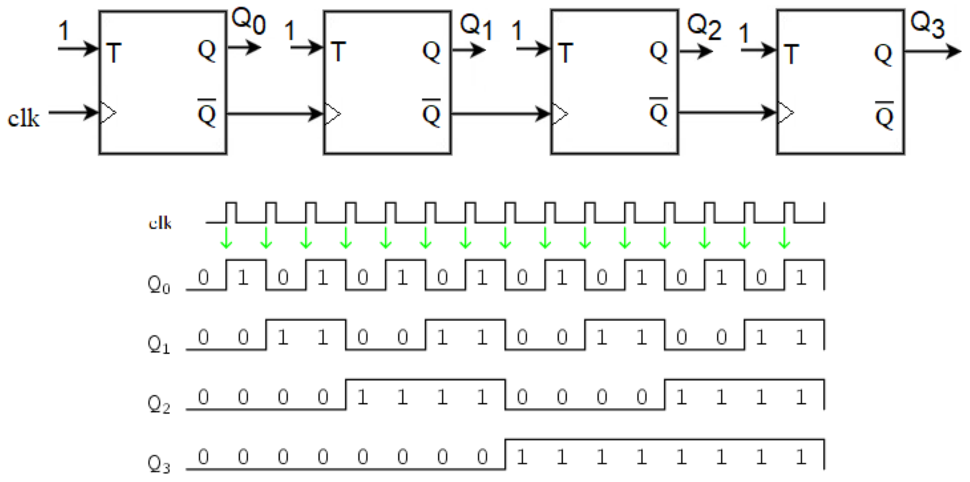
Hence, up counters and down counters generally have 2^N states and are modulo-2^N counters.

The most efficient way is to build up and down counters is to use T flip-flops that are always in toggle mode (i.e $T = 1$).
As mentioned earlier, the first T flip-flop is run by a clock pulse. The output of the first T flip-flop is fed as clock to the second T flip-flop and so on.

- If positive edge triggered T flip-flops are used:
  - If $\overline{Q}$ of each stage is fed to clk of next stage, the obtained counter will be an up counter.
  - If Q of each stage is fed to clk of next stage, the obtained counter will be a down counter.
- If negative edge triggered T flip-flops are used:
  - If $\overline{Q}$ of each stage is fed to clk of next stage, the obtained counter will be a down counter.
  - If Q of each stage is fed to clk of next stage, the obtained counter will be an up counter.

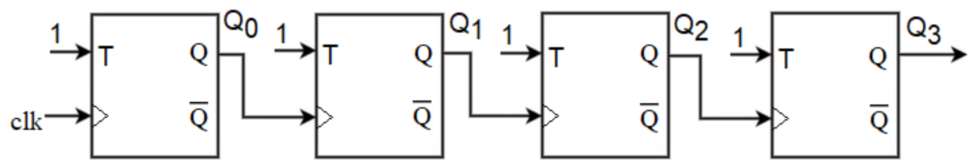
Also, it can be noted that if complemented outputs are taken (i.e $\overline{Q}$ instead of Q), then an up-counter will become down-counter and vice versa.

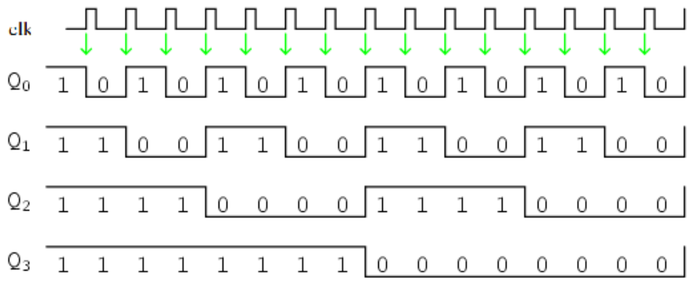
The figure below shows 4-bit up counter and the output waveforms.



1
T
Q
Q̄
clk
Q0
Q1
Q2
Q3
Q0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Q1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
Q2 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
Q3 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

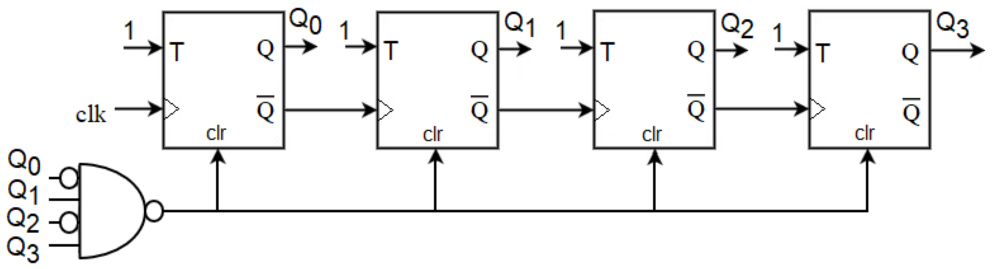
The figure below shows 4-bit down counter and the output waveforms.





It is evident that these counters can only count up to or down from $2^N - 1$ i.e the number of states is always some power of 2. However, they can be modified to count to any arbitrary number using reset or set inputs.

For example, consider that a modulo-10 up counter is to be designed (i.e it has to count from 0 to 9). It is evident that 4-bits are essential but the complete 4-bit up counter cycle should not be covered.
In order to do this, the counter (i.e all the 4 flip-flops) can simply be reset as soon as it counts to 10 (1010). This way, the counter restarts counting from 0 after 9 and hence, modulo-10 up counter is obtained.



Note that the same counters can be implemented using D flip-flops where each input D is connected to Q or $\overline{Q}$ depending on the operation required.

A disadvantage of asynchronous counters is that since every stage will have some propagation delay, the overall delay of the counter will be high.

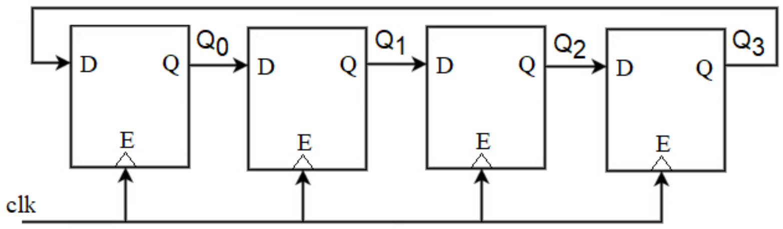
5.4.2 Synchronous Counters

Synchronous counters are made of flip-flops where all the flip-flops are driven by the same clock pulse. Hence, the flip-flops are all in sync with each other. Since each flip-flop is running on the same clock, it means that the delay of each stage does not add up. Hence, synchronous counters are faster than asynchronous counters.

Arbitrary up and down counters can be designed using synchronous design. Before going into that, a couple of special synchronous counters are explored.

Ring Counter:
A Ring counter is the simplest type of asynchronous counter. It is made using cascaded D flip-flops with the output of the last flip-flop connected to the input of the first flip-flop. It circulates a single bit around the ring. Hence, it is basically a SISO shift register in rotate mode.

A 4-bit ring counter is illustrated.



State Table of 4-bit ring counter:

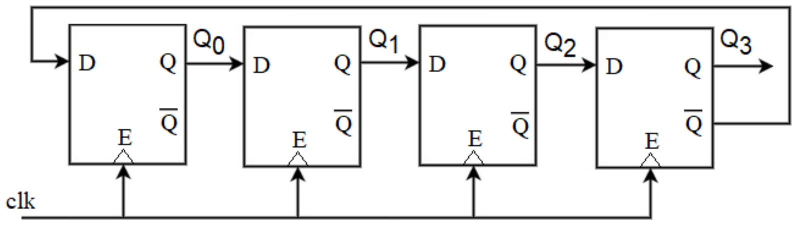
Clock	Q_0	Q_1	Q_2	Q_3
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	1	0	0	0

It is evident that a 4-bit ring counter has 4 states. Hence, an N-bit ring counter has N states, which is why it is a modulo-N counter.

For ideal working of a ring counter, one of the bits (i.e on of the flip-flops) has to be set initially while all others bits must be reset (or cleared).

Johnson Counter:
A Johnson counter is a modified version of the Ring counter where the complement of the output of the last flip-flop is connected to the input of the first flip-flop. It is also called "Twisted ring counter".

A 4-bit Johnson counter is illustrated.

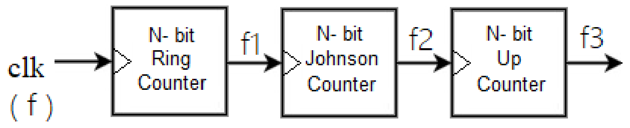


State Table of 4-bit Johnson counter:

Clock	Q_0	Q_1	Q_2	Q_3
1	0	0	0	0
2	1	0	0	0
3	1	1	0	0
4	1	1	1	0
5	1	1	1	1
6	0	1	1	1
7	0	0	1	1
8	0	0	0	1
9	0	0	0	0

It is evident that a 4-bit Johnson counter has 8 states. Hence, an N-bit Johnson counter has $2N$ states, which is why it is a modulo-$2N$ counter.

Frequency Division:
An important application of counters is frequency division. If a clock of frequency f is used to drive a modulo-M counter, then the final output bit will be of frequency f/M.



$$f_1 = \frac{f}{N} \qquad f_2 = \frac{f_1}{2N} = \frac{f}{2N^2} \qquad f_3 = \frac{f_2}{2^N} = \frac{f}{2^{N+1}N^2}$$

5.5 Finite State Machines

A Finite State Machine (FSM) is an abstract machine that can be in exactly one of a finite number of states at any given time.

Hence, finite state machines are basically synchronous counters with extra control inputs and indicator outputs.

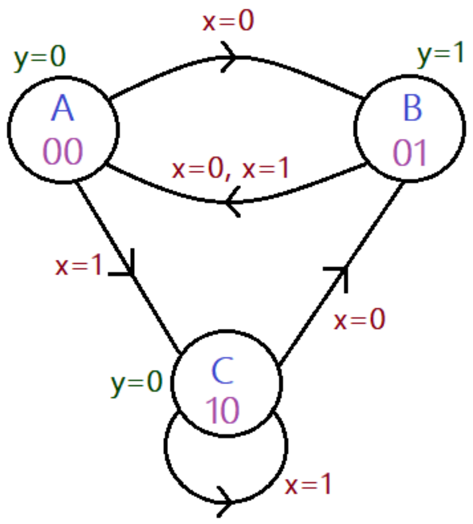
- State of FSM is the data stored in the FSM.
- Inputs of FSM control transition of states of the FSM.
- Outputs of FSM indicate that the FSM is in a certain state (or one of some certain states).

There are two types of FSMs, namely the Moore machine and the Mealy machine. They are both elaborated in the subsequent subsubsections.

5.5.1 Moore Machines

A Moore machine is a finite state machine whose outputs are determined only by its current state.

Consider the state diagram illustrated.



It is evident that the output y depends only on the present state of the FSM since $y = 1$ if it is in state B and $y = 0$ if it is in either state A or state C, regardless of how the transition occurred (i.e no dependence on input x).

A Moore machine can be designed using any flip-flops. Consider that the give state diagram has to be realized using D flip-flops. 2 flip-flops are necessary since there are 3 states.

Note that state A is when $Q_1Q_0 = 00$, state B is when $Q_1Q_0 = 01$ and state C is when $Q_1Q_0 = 10$. Also, let default state be A.

State transition table:

Q_1	Q_0	x	Q_1^+	Q_0^+	y
0	0	0	0	1	1
0	0	1	1	0	0
0	1	$\times$	0	0	0
1	0	0	0	1	1
1	0	1	1	0	0
1	1	$\times$	0	0	0

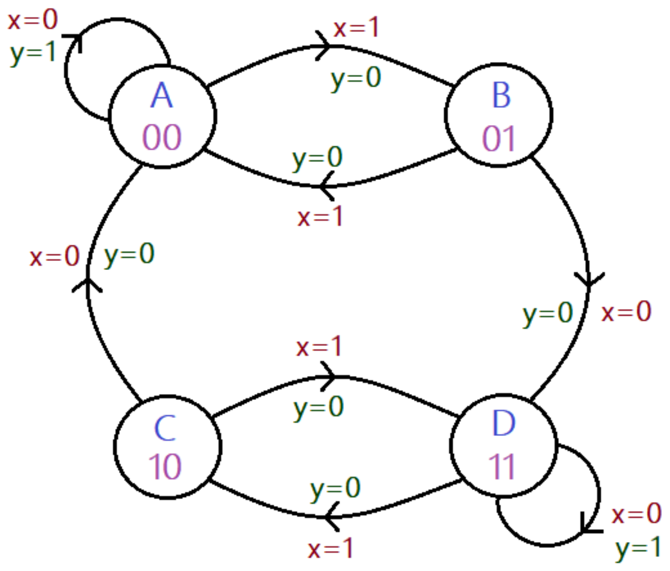
Since $Q_1^+ = D_1$ and $Q_0^+ = D_0$, the inputs to the 2 D flip-flops can be obtained using the above table. The output expression is also quite obvious.

$$D_1 = x.\overline{Q_0} \qquad D_0 = \overline{x}.\overline{Q_0} \qquad y = Q_0.\overline{Q_1}$$

5.5.2 Mealy Machines

A Mealy machine is a finite state machine whose outputs are determined by its current state as well as current inputs.
This can also be interpreted as: outputs depend on the transition that has presently occurred.

Consider the state diagram illustrated.



It is evident that the output y depends on the present state of the FSM as well as the input x since $y = 1$ only if it is in state A or state D and if $x = 0$. For all other cases, $y = 0$. This means, $y = 1$ only when the transition happening is either from A to itself or from D to itself (which only happens if $x = 1$). Hence, dependence of y on x is established.

A Mealy machine can be designed using any flip-flops. Consider that the give state diagram has to be realized using D flip-flops. 2 flip-flops are necessary since there are 4 states.

Note that state A is when $Q_1Q_0 = 00$, state B is when $Q_1Q_0 = 01$, state C is when $Q_1Q_0 = 10$ and state D is when $Q_1Q_0 = 11$. Also, consider the default state to be A.

State transition table:

Q_1	Q_0	x	Q_1^+	Q_0^+	y
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	1	0	0

Since $Q_1^+ = D_1$ and $Q_0^+ = D_0$, the inputs to the 2 D flip-flops can be obtained using the above table. The output expression is also quite obvious.

$$D_1 = Q_0.\overline{x} + Q_1.x \qquad D_0 = Q_0 \oplus Q_1 \qquad y = \overline{x}.(\overline{Q_0 \oplus Q_1})$$

6 Semiconductor Memories

Memory obviously refers to the hardware that stores data. Registers made from flip-flops are basic memory locations. Semiconductor memories are registers which have features such as read from memory or write into memory or both. The two major categories of semiconductor memories are the RAM and the ROM.

6.1 Read Only Memory (ROM)

ROM is the type of memory in which data is stored permanently or semi-permanently. Data can be read from a ROM, but there is no write operation and hence the data entered in a ROM (while manufacturing) can't be re-written directly.
Because ROMs retain stored data even if power is turned off, they are non-volatile memories.

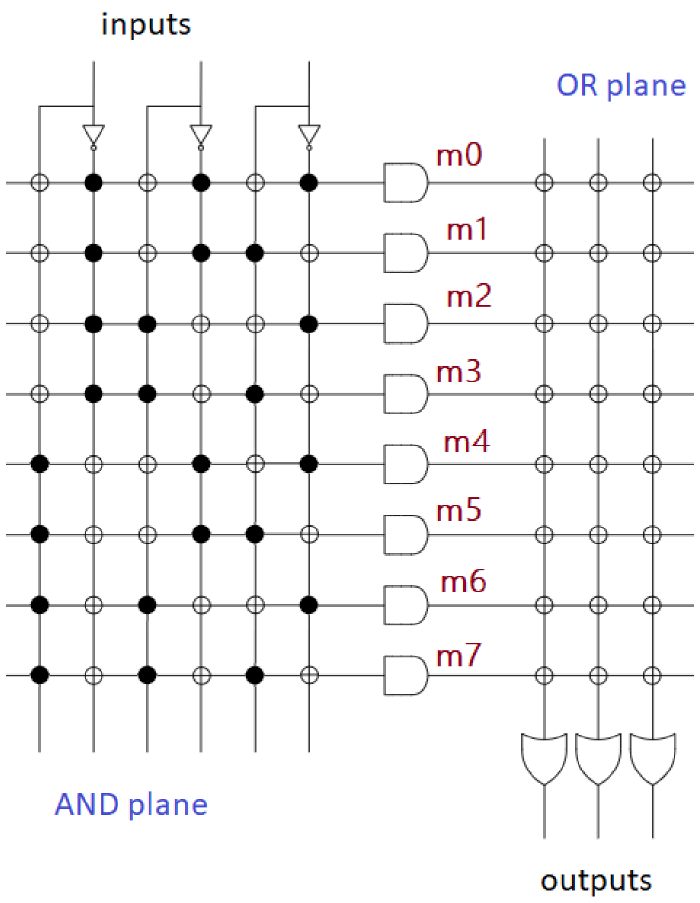
A ROM basically consists of a matrix (or plane) of fixed number of AND gates and another matrix (or plane) of OR gates. Initial models of ROM had fixed OR gate plane.
Eventually, with improvement of technology, the OR gate plane was -

- programmable (PROM)
- erasable via UV rays and programmable (EPROM)
- electronically erasable and programmable (EEPROM)
- erasable and alterable (EAPROM, Flash Memory)

Hence, A ROM is also considered as a programmable logic device (more on these devices will be elaborated in the next section).

A ROM consists of input buffers and inverters to generate all inputs and their complements. These are connected to the fixed AND gate array to generate all min-terms. Since N input variables can generate 2^N min-terms, there will be 2^N fixed AND gates. The required boolean functions can then be realized by using OR gates to sum these min-terms together (SOP form).

General structure of a 3 input, 3 output ROM:



For example, consider the output data as the following functions of 3 variables A, B and C.

$$Y_1 = A.B.C + A.\overline{B}.\overline{C} \qquad Y_2 = \overline{A}.B.C + A.B.\overline{C} + A.\overline{B}.C$$

$$\implies Y_1 = \sum_m (4,7) \qquad Y_2 = \sum_m (3,5,6)$$

Hence, the OR gate plane must be programmed such that the first output is sum of min-terms m_4, m_7 and the second output is sum of min-terms m_3, m_5, m_6.

6.2 Random Access Memory (RAM)

RAM is the type of memory in which data is stored temporarily. Data can be randomly written into a RAM or randomly read from a RAM during run-time of a system.
Because RAMs lose stored data when power is turned off, they are volatile memories.

6.2.1 Static RAM

Data written in SRAM remains intact as long as power supply is present.

6.2.2 Dynamic RAM

Data written in DRAM slowly gets erased with time even if supply is present (due to capacitor discharge). Hence, periodic refreshing of data is needed in a DRAM in order for it to not lose the stored data.

7 Programmable Logic Devices

A programmable logic device (PLD) is an electronic component used to build reconfigurable digital circuits. Unlike integrated circuits which consist of logic gates and have a fixed function, a PLD has an undefined function at the time of manufacture. Before the PLD can be used in a circuit, it must be programmed (reconfigured) by using a specialized program.

PLDs are of vital importance because their reconfigurable nature can be very useful for testing circuits before fabricating them as ICs, for creating hardware that can perform complex computations, etc.

There are different types of PLDs. They are listed below.

1. Programmable Logic Array (PLA)
2. Programmable Array Logic (PAL)
3. Complex Programmable Logic Device (CPLD)
4. Field Programmable Logic Array (FPGA)

Each of these PLDs will be introduced and elaborated in detail in the upcoming subsections.

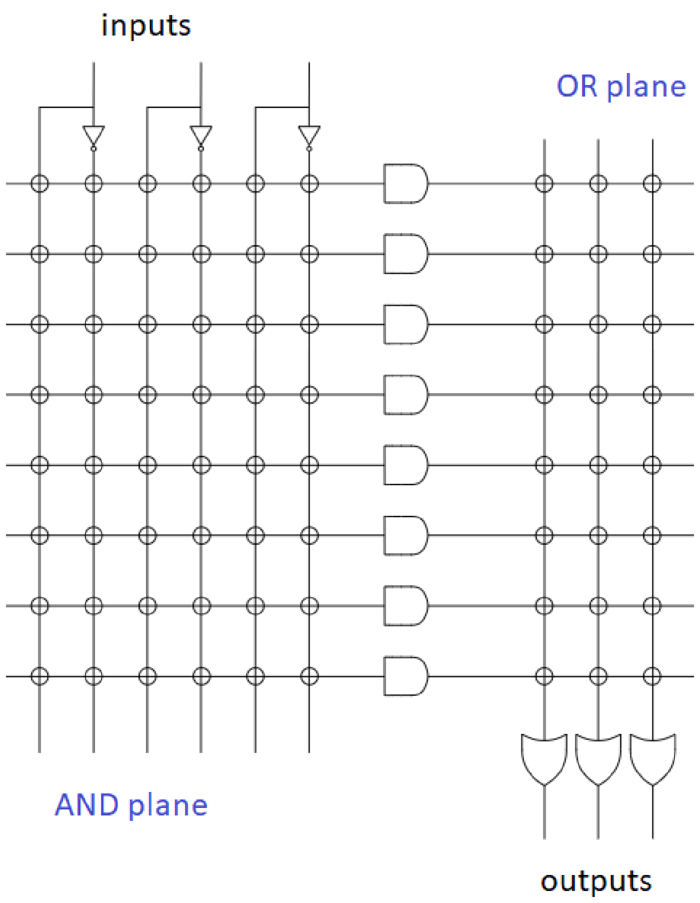
7.1 Programmable Logic Array

A programmable logic array (PLA) is a kind of programmable logic device used to implement combinational logic circuits. A PLA has a set of programmable AND gate planes, which link to a set of programmable OR gate planes, which can then be conditionally complemented to produce an output.

It has 2^N AND gates for N input variables, and for M outputs from the PLA, there should be M OR gates, each with programmable inputs from all of the AND gates. This layout allows for many logic functions to be synthesized in the sum of products canonical forms.

(note the resemblance between PLA and ROM)

General structure of a 3 input, 3 output PLA:



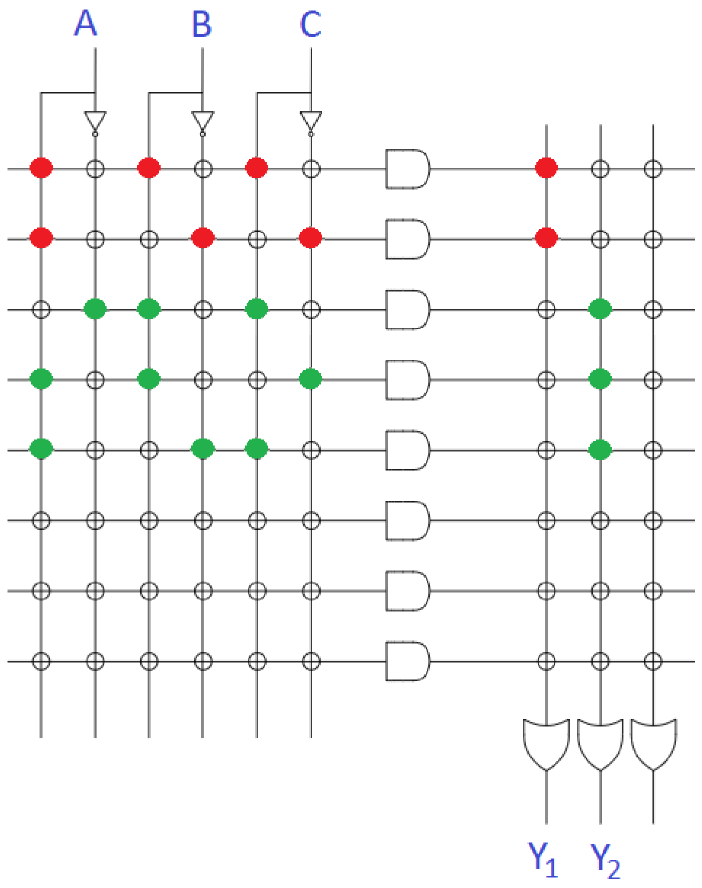
Boolean logic implementation procedure on a PLA:

- Prepare logic/boolean function in SOP (sum of products) form.
- Obtain the minimum SOP form to reduce the number of product terms to a minimum.
- Decide the input connection of the AND matrix for generating the required product term.
- Decide the input connections of OR matrix to generate the sum terms.
- Program the PLA accordingly.

Example: A PLA is to be programmed to realize these 2 logic functions.

$$Y_1 = A.B.C + A.\overline{B}.\overline{C} \qquad\qquad Y_2 = \overline{A}.B.C + A.B.\overline{C} + A.\overline{B}.C$$

There are a total of 5 min-terms and 2 outputs. Hence, out of the 8 AND gates, only 5 will be used and out of 3 OR gates, only 2 will be used.

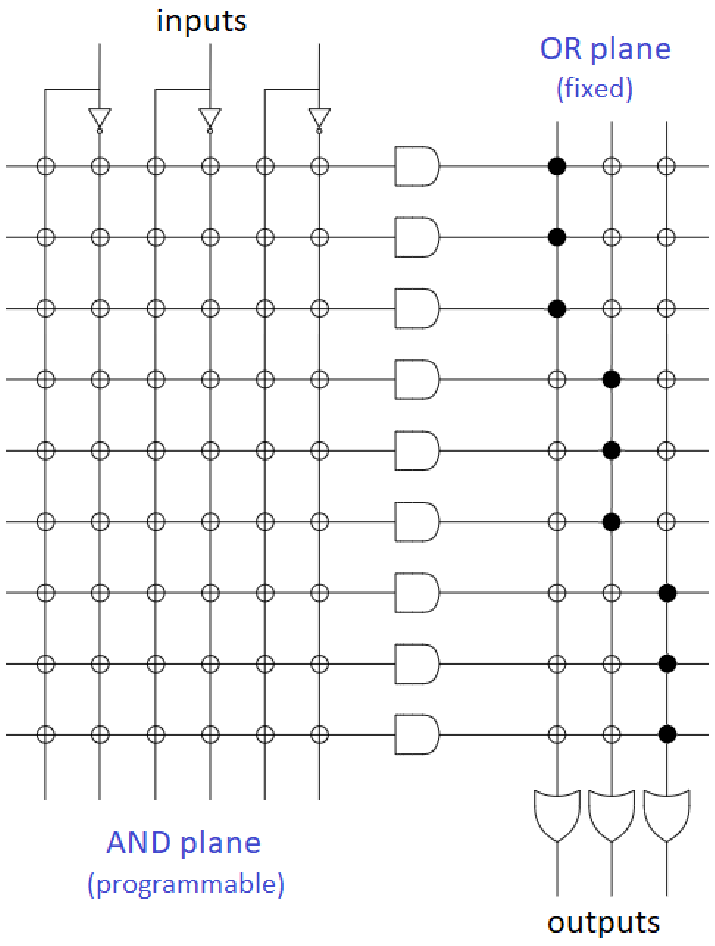


7.2 Programmable Array Logic

Programmable Array Logic (PAL) is a commonly used programmable logic device which has programmable AND gate array and fixed OR gate array. Because only the AND gate array is programmable, it is easier to use but not flexible as compared to Programmable Logic Array (PLA).

The number of AND gates corresponding to each output will be equal to the number of input variables itself. Hence if there are N inputs, there will be N AND gates and 1 OR gate for every boolean function that is to be realized.

General structure of a 3 input, 3 output PAL:



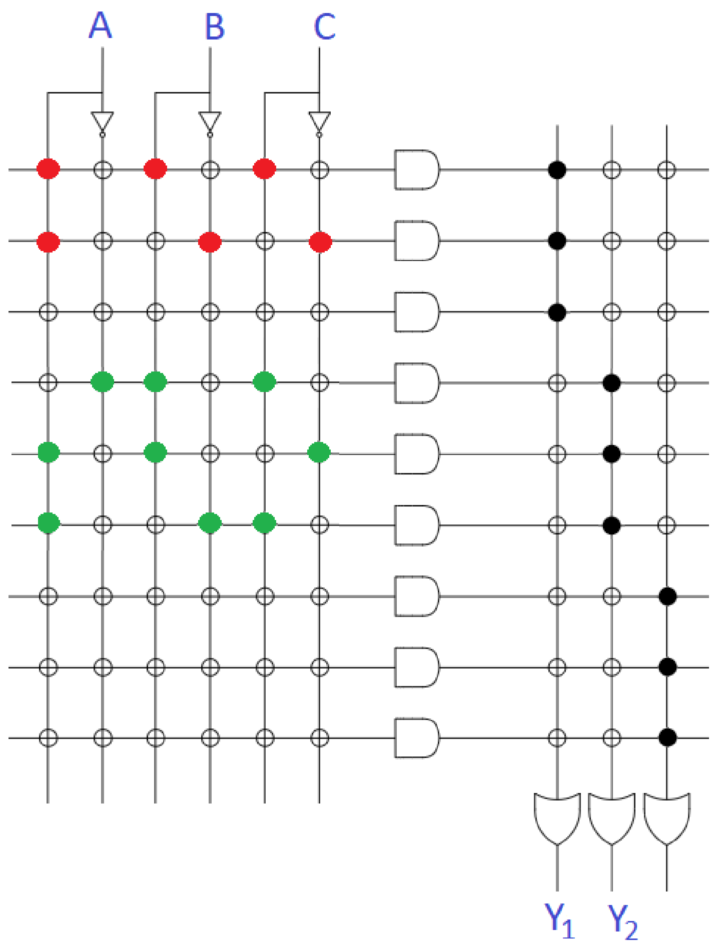
Boolean logic implementation procedure on a PAL are same as that for PLA except for the step where connections of OR matrix has to be decided. This step is not necessary since the OR matrix is fixed in a PAL.

After minimizing the boolean function, usually the number of min-terms will be less than the maximum number and hence some of the AND gates in the AND matrix will be left without utility.

Example: A PAL is to be programmed to realize these 2 logic functions.

$$Y_1 = A.B.C + A.\overline{B}.\overline{C} \qquad\qquad Y_2 = \overline{A}.B.C + A.B.\overline{C} + A.\overline{B}.C$$

There are a total of 5 min-terms and 2 outputs. Hence, out of the 9 AND gates, only 5 will be used and out of 3 OR gates, only 2 will be used.



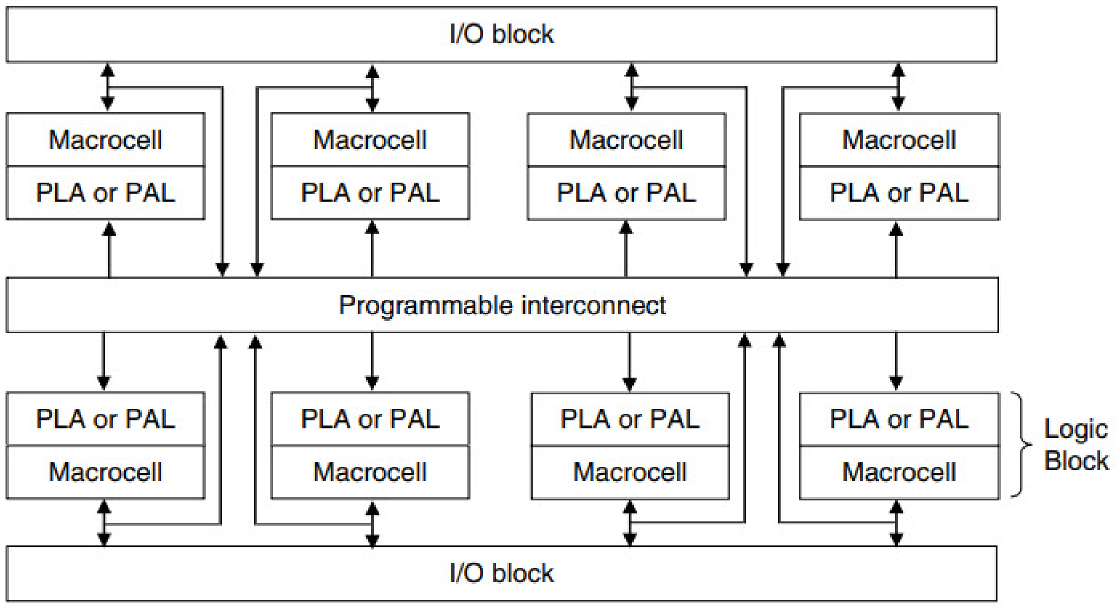
It is evident that the PAL and the PLA are quite similar despite the difference in OR gate array (i.e it being fixed or programmable).
These are the most basic types of PLDs and are applicable to realize arbitrary combinational circuits only.

7.3 Complex Programmable Logic Device

A complex programmable logic device (CPLD) is a programmable logic device with complexity between that of PALs and FPGAs, and architectural features of both.

A CPLD uses a programmable interconnect that connects several logic blocks that are basic programmable structures to and from input/output blocks. Each CPLD logic block is basically a registered PLA or PAL i.e it consists of a macro-cell (i.e a register and a mux) connected to a PLA or PAL.

General block diagram of a CPLD is illustrated.



CPLDs can be used to implement much more complicated digital systems since they consist of multiple programmable blocks and memory elements.

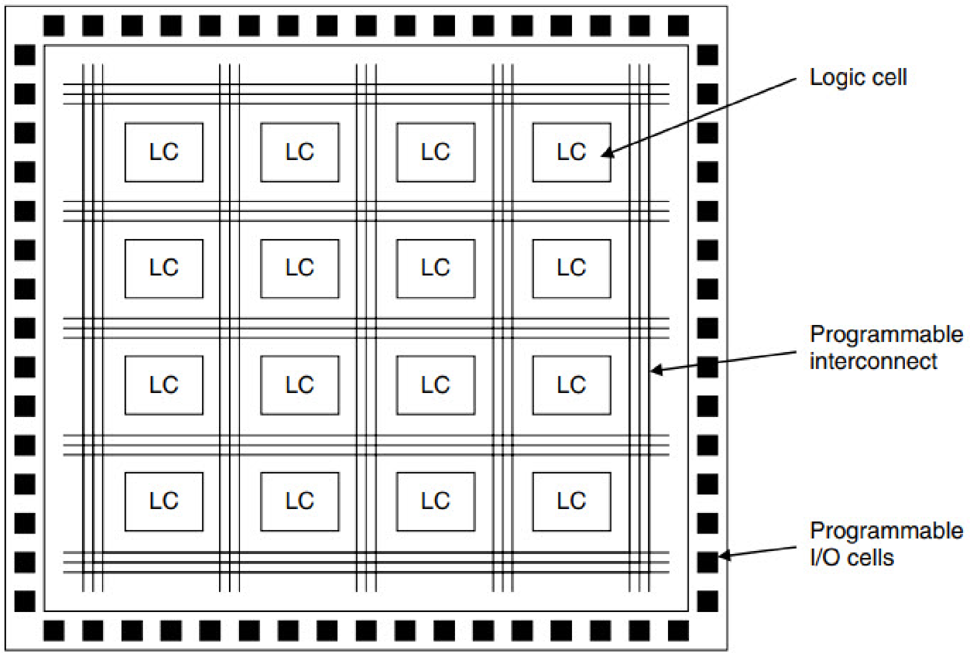
Programming CPLDs is obviously more complicated than programming PALs or PLAs. They require using a programming language such as VHDL to provide instructions.

7.4 Field Programmable Logic Array

A field programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing (hence the term "field-programmable").
The FPGA configuration is generally specified using a hardware description language such as Verilog or VHDL.

FPGAs contain an array of programmable logic blocks, and a hierarchy of "reconfigurable interconnects" allowing blocks to be "wired together", like many logic gates that can be inter-wired in different configurations.

The basic block diagram illustrating the structure of an FPGA is shown.

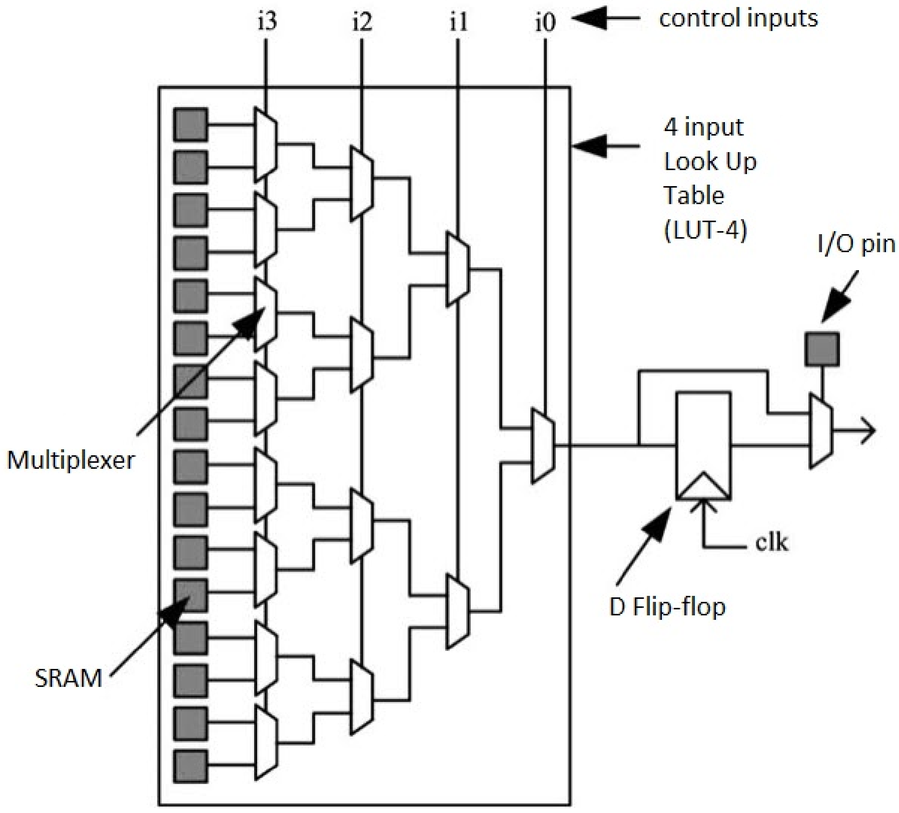


Components of an FPGA:
Typically, an FPGA consists of three basic components.

- Programmable Logic Cells (or Logic Blocks) – responsible for implementing the core logic functions.
- Programmable Routing – responsible for connecting the Logic Blocks.

- I/O Blocks – help to make external connections (they are connected to the Logic Blocks through the routing).

Logic Blocks are also called Configurable Logic Blocks (CLBs) or Logic Array Blocks (LABs). They are the basic components of an FPGA, which provide both the logic and storage functionalities.



A Logic Block can be made up of a single Basic Logic Element or a set of interconnected Basic Logic Elements, where a Basic Logic Element is a combination of a Look-up table (which is in turn made up of SRAM and Multiplexers) and a Flip-flop.

An LUT with 'n' inputs consists of 2^n configuration bits, which are implemented by SRAM Cells. Using these 2^n SRAM Bits, the LUT can be configured to implement any logical function.

8 Data Converters

Data converters are circuits that convert data from one form to another. These are necessary because most signals that are directly input from nature are analog signals. Similarly, most signals that need to be output for appropriate usage also have to be analog signals.
As mentioned at the beginning, digital domain is used because it makes processing faster and takes less storage space.
Hence, circuits that convert analog signals to digital signals and vice versa are of extreme importance.

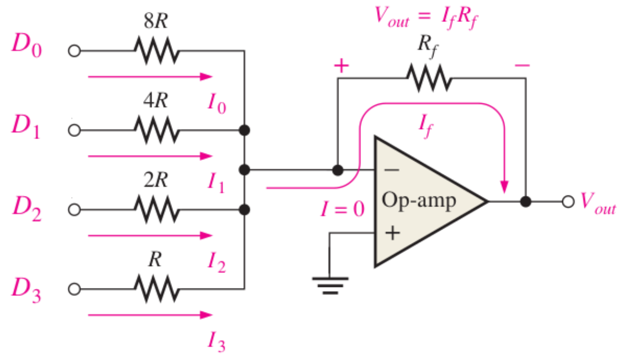
8.1 Digital to Analog Converters

A digital to analog converter (DAC) converts a digital signal i.e a sequence of bits to analog voltage. This is most conveniently done using Operational Amplifiers (covered in Analog Electronic Circuits in great detail).

8.1.1 Binary Weighted Resistor type DAC

The binary weighted resistor type DAC used resistors of that are proportional to powers of 2.

4-bit DAC circuit is illustrated below.



The digital inputs are D_0, D_1, D_2 and D_3 where D_3 is the MSB and D_0 is the LSB.
If any digital input is "1", then the voltage at that node will be $+V$ and if any digital input is "0", then the voltage at that node will be 0.

Using this, the currents can be found to be: $I_0 = VD_0/8R, I_1 = VD_1/4R, I_2 = VD_2/2R$ and $I_3 = VD_3/8R$ and the total current $I_f = I_0 + I_1 + I_2 + I_3$. From this, the voltage V_{out} can be obtained as-

$$V_{out} = \frac{VR_f}{R}\left(\frac{D_3}{1} + \frac{D_2}{2} + \frac{D_1}{4} + \frac{D_0}{8}\right)$$

The ratio R_f/R decides the gain or amplification factor.
The resolution of a DAC is the smallest value it can convert or catch i.e the output value corresponding to the digital input 0001. Here, the resolution will be $VR_f/8R$.

General formula for output voltage of an n-bit binary weighted type DAC:

$$V_{out} = \frac{VR_f}{R}\left(\frac{D_{n-1}}{1} + \frac{D_{n-2}}{2} + ... + \frac{D_1}{2^{n-2}} + \frac{D_0}{2^{n-1}}\right)$$

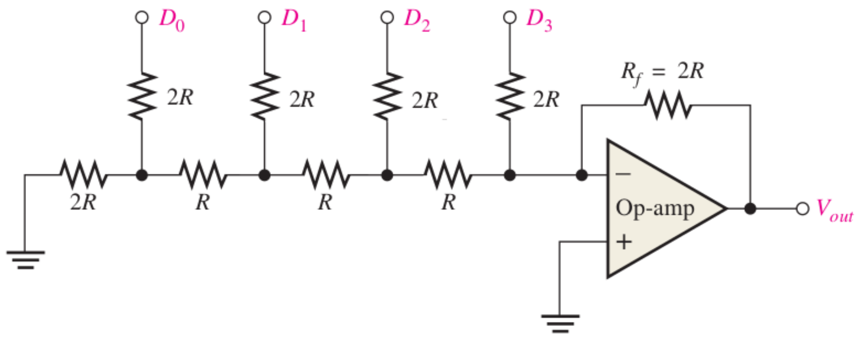
General formula for resolution of an n-bit binary weighted type DAC:

$$\text{Res} = \frac{VR_f}{2^{n-1}R}$$

The advantage of this circuit is speed of operation is high. However, it requires a large range of specific resistors, which may not be available.

8.1.2 R-2R Ladder type DAC

The R-2R ladder type DAC using the R-2R ladder circuit in combination with OpAmp to achieve the same operation as the binary weighted resistor type DAC but with only two values of resistors.



Upon analysing the R-2R ladder type DAC circuit, it can be observed that the behaviour is exactly same as the binary weighted resistor type DAC. Hence, the expressions for output voltage and resolution are the same.

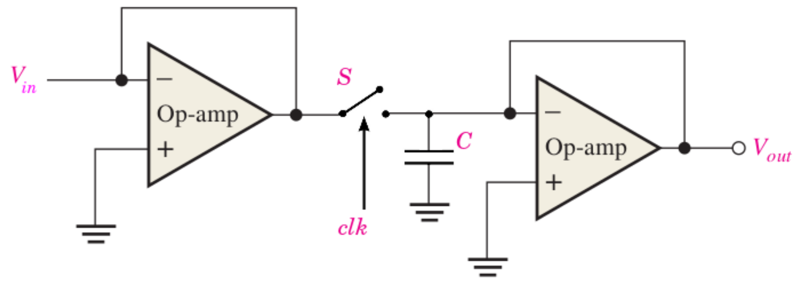
Since there inputs affect the output sequentially, the speed of operation of the R-2R ladder type DAC is relatively low. However, since only two resistor values are necessary for any number of input bits, it overcomes the major disadvantage of the binary weighted resistor type DAC.

8.2 Analog to Digital Converters

An analog to digital converter (ADC) converts an analog input voltage to a sequence of bits i.e digital signal. There are several ways of doing this.

8.2.1 Sample & Hold Circuit

A sample & hold circuit is a basic building block of an analog to digital converter. Given an analog input signal, this circuit samples the analog values holds the sampled values till the next sampling time instant.



The clock signal (clk) which is also called control signal specifies the frequency at which the sampling should be done. The switching device S is usually a MOSFET and a holding capacitor C is used.

The input signal V_{in} is an analog signal. This signal will be sampled by switching operating as per the clock signal when clock is high.
When clock is low, there is no path connecting the input to the output, but the holding capacitor has no discharge path, so the previously sampled value will still appear at the output V_{out}.

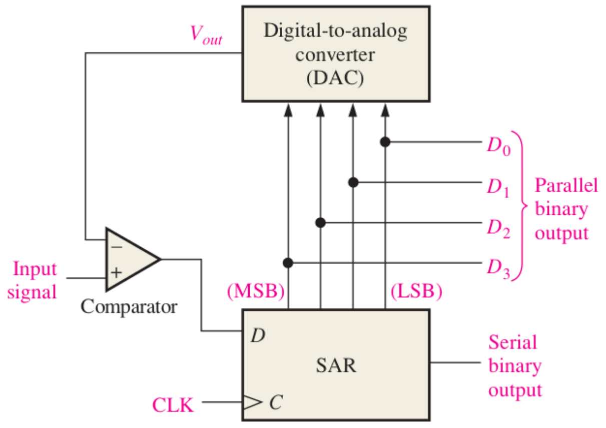
After sample-hold operation is achieved, the obtained signal has to be quantized and encoded to obtain a stream of digital bits corresponding to the input analog signal (covered in Communication Systems).

8.2.2 Successive Approximation type ADC

The successive approximation type ADC is the most widely used ADC since it has fixed conversion time. The working is as follows.

- The input bits of the DAC are enabled (made equal to a 1) one at a time, starting with the most significant bit (MSB).
- As each bit is enabled, the comparator produces an output that indicates whether the input signal voltage is greater or less than the output of the DAC.
- If the DAC output is greater than the input signal, the comparator's output is LOW, causing the bit in the register to reset.
- If the output is lesser than the input signal, the 1 bit is retained in the register. The system does this with the MSB first, then the next most significant bit, then the next, and so on.
- After all the bits of the DAC have been tried, the conversion cycle is complete.

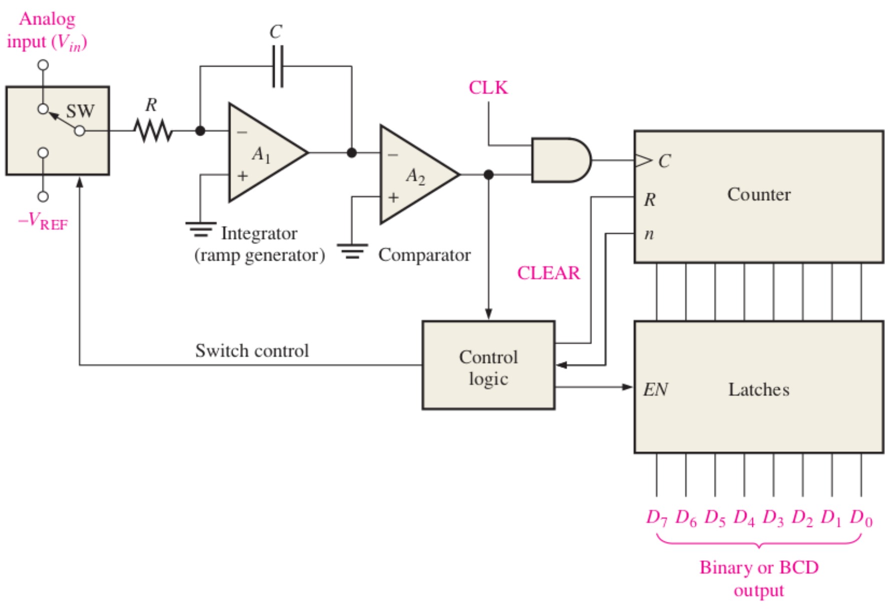
Basic block diagram of a 4-bit successive approximation ADC is shown.



8.2.3 Dual Slope type ADC

The dual slope type ADC uses an integrator and hence is also called 'integrator type ADC'.

Block diagram of dual slope ADC is shown.



Working principle of dual slope type ADC is explained.

- The unknown input voltage (V_{in}) is applied to the integrator and is allowed to ramp for a fixed period of time (t_u).
- Then a known reference voltage (V_{ref}) of opposite polarity is applied to the integrator and is allowed to ramp till the integrator output becomes equal to zero (t_d).
- The input voltage is computed as:

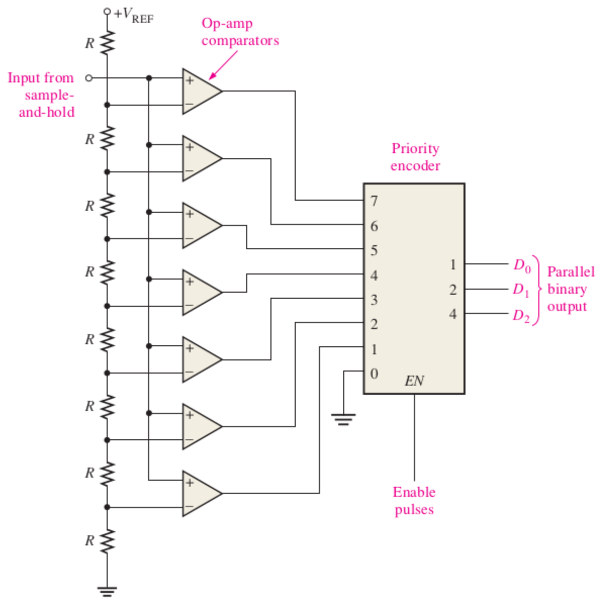
$$V_{in} = -V_{ref} \frac{t_d}{t_d}$$

Note that longer integration times allow for higher resolutions, so speed has to sacrificed for accuracy.

8.2.4 Flash type ADC

The flash method utilizes special high-speed comparators that compare reference voltages with the input voltage. The output of a sample and hold circuit is given as input to the flash type ADC. These sampled values are compared with standard references and the outputs of the comparators are fed to a priority encoder which gives the required sequence of bits.

A 3-bit flash type ADC is shown.



The number of comparators used here is 7. In general, an n-bit flash type ADC requires 2^{n-1} comparators.