LECTURE NOTES
ON

Digital Fundamentals
2101CS303

B.Tech. 3rd Semester
(Computer Science and Engineering)

Prepared By

Krunal D. Vyas

Assistant Professor

krunal.vyas@darshan.ac.in

9601901005

Darshan UNIVERSITY
योगः कर्मसु कौशलम्

Contents

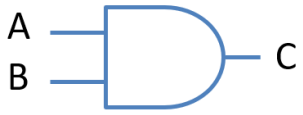
Unit-1

Fundamentals of Digital System

1.1. Logical Gates

1.1.1. AND Gate

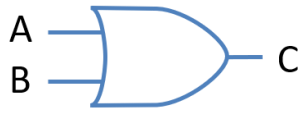
- An AND gate has two or more inputs but only one output.
- The output assumes the logic 1, only when each one of its inputs is at logic 1.
- The output assumes the logic 0 even if one of its inputs is at logic 0.
- The logic symbol & truth table are shown in below figure.
- Notation:- $C = A \cdot B$



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

1.1.2. OR Gate

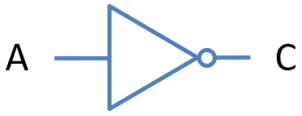
- An OR gate has two or more inputs but only one output.
- The output assumes the logic 0, only when each one of its inputs is at logic 0.
- The output assumes the logic 1 even if one of its inputs is at logic 1.
- The logic symbol & truth table are shown in below figure.
- Notation:- $C = A + B$



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

1.1.3. NOT Gate

- A NOT gate (also called an *inverter*) has only one input & one output.
- It is a device whose output is always the complement of its input.
- The output assumes the logic 1, when its input is at logic 0.
- The output assumes the logic 0, when its input is at logic 1.
- The logic symbol & truth table are shown in below figure.
- Notation:- $C = \bar{A}$



A	C
0	1
1	0

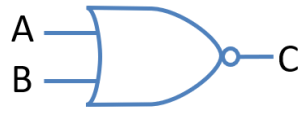
1.1.4. NAND Gate (Universal Gate)

- NAND means NOT AND, i.e. the AND output is NOTed.
- The output assumes the logic 0, only when each one of its inputs is at logic 1.
- The output assumes the logic 1 even if one of its inputs is at logic 0.
- The logic symbol & truth table are shown in below figure.
- Notation:- $C = \overline{A \cdot B}$

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

1.1.5. NOR Gate (Universal Gate)

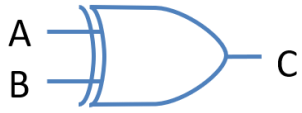
- NOR means NOT OR, i.e. the OR output is NOTed.
- The output assumes the logic 1, only when each one of its inputs is at logic 0.
- The output assumes the logic 0 even if one of its inputs is at logic 1.
- The logic symbol & truth table are shown in below figure.
- Notation:- $C = \overline{A + B}$



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

1.1.6. Exclusive OR Gate (Ex-OR Gate)

- An X-OR gate has two or more inputs but only one output.
- The output assumes the logic 1 when one and only one of its inputs assumes a logic 1.
- Under the conditions when both the inputs assume the logic 0, or when both the inputs assume the logic 1, the output assumes a logic 0.
- Since, an X-OR gate produces an output 1 only when the inputs are not equal, it is called an *anti-coincidence gate* or *inequality detector*.
- The logic symbol & truth table are shown in below figure.
- Notation:- $C = A \oplus B$



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

1.1.7. Exclusive NOR Gate (Ex-OR Gate)

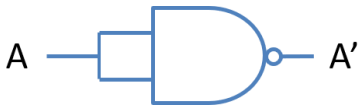
- An X-NOR gate has two or more inputs but only one output.
- The output assumes the logic 0 when one and only one of its inputs assumes a logic 0.
- Under the conditions when both the inputs assume the logic 1, or when both the inputs assume the logic 1, the output assumes a logic 0.
- Since, an X-NOR gate produces an output 1 only when the inputs are equal, it is called a *coincidence gate* or *equality detector*.
- The logic symbol & truth table are shown in below figure.
- Notation:- $C = A \odot B$

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

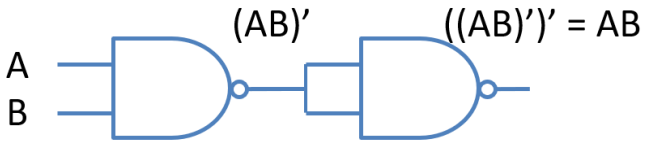
1.1.8. Basic Gates as Universal Gate

- ***Implementation of NOT, AND & OR gates using NAND gate only***

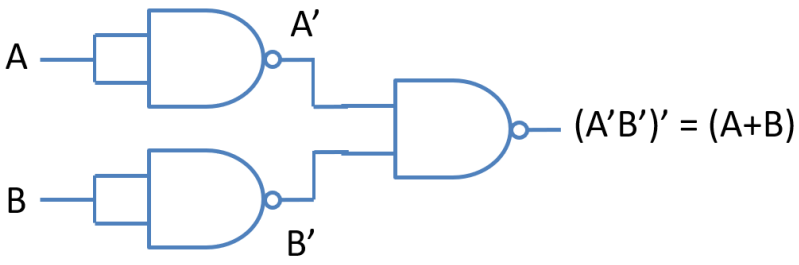
1. NOT using NAND gate
   - A NAND gate can also be used as an inverter by tying all its input terminals together and applying the signal to be inverted to the common terminal.



2. AND using NAND gate
   - NAND means NOT AND, i.e. the AND output is NOTed.
   - So, a NAND gate is combination of an AND gate and a NOT gate.

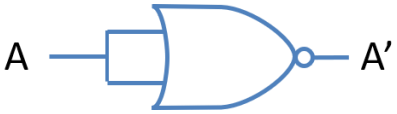


3. OR using NAND gate
   - By inverting inputs in NAND gate, a OR gate is constructed via De Morgan's theorem.
   - $\overline{\bar{A}\,\bar{B}} = \bar{\bar{A}} + \bar{\bar{B}} = A + B$

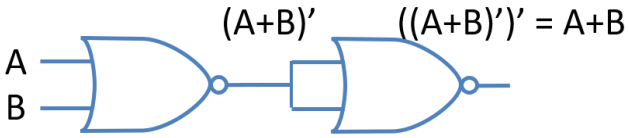


- ***Implementation of NOT, AND & OR gates using NOR gate only***

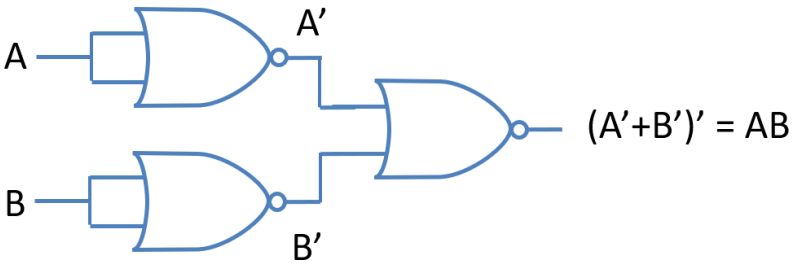
1. NOT using NOR gate
   - A NOR gate can also be used as an inverter by tying all its input terminals together and applying the signal to be inverted to the common terminal.



2. OR using NOR gate
   - NOR means NOT OR, i.e. the OR output is NOTed.
   - So, a NOR gate is combination of an OR gate and a NOT gate.



3. AND using NOR gate
   - By inverting inputs in NOR gate, a AND gate is constructed via De Morgan's theorem.
   - $\overline{\bar{A} + \bar{B}} = \bar{\bar{A}}\bar{\bar{B}} = A\,B$



1.2. Number Systems

- There are mainly four number systems which are used in digital electronics platform.

1. Decimal number system

- The decimal number system contains ten unique symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- The base or radix is 10.
- 9's and 10's complements are possible for any decimal number.

2. Binary number system

- The binary number system contains two unique symbols 0, 1.
- The base or radix is 2.
- 1's and 2's complements are possible for any binary number.

3. Octal number system

- The octal number system contains eight unique symbols 0, 1, 2, 3, 4, 5, 6, 7.
- The base or radix is 8.
- 7's and 8's complements are possible for any octal number.

4. Hexadecimal number system

- The hexadecimal number system contains sixteen unique symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.
- The base or radix is 16.
- 15's and 16's complements are possible for any hexadecimal number.

In general, if radix or base of a number system is "r", then there is possibility of r's complement and (r-1)'s complement of a number.

1.2.1. Decimal to Binary conversion

- The decimal integer is converted to the binary integer number by successive division by 2, and the decimal fraction is converted to the binary fraction number by successive multiplication by 2.
- In the successive division-by-2 method, the given decimal integer number is successively divided by 2 till the quotient is 0.
- The remainders read from bottom to top give the equivalent binary integer number.
- In the successive multiplication-by-2 method, the given decimal fraction and the subsequent fractions are successively multiplied by 2, till the fraction part of the product is 0 or till the desired accuracy is obtained.
- The integers read from top to bottom give the equivalent binary integer number.
- To convert a mixed number to binary, convert the integer and fraction parts separately to binary and then combine them.
- Example:- $(125.6875)_{10} = (\ )_2$

2	125	1
2	62	0
2	31	1
2	15	1
2	7	1
2	3	1
2	1	1
	0	

$0.6875 \times 2 = 1.3750 \quad 1 + 0.3750$
$0.3750 \times 2 = 0.7500 \quad 0 + 0.7500$
$0.7500 \times 2 = 1.5000 \quad 1 + 0.5000$
$0.5000 \times 2 = 1.0000 \quad 1 + 0.0000$

Hence, $(125.6875)_{10} = (1111101.1011)_2$

1.2.2. Binary to Decimal Conversion

- Binary numbers may be converted to their decimal equivalents by the positional weights method.
- In this method, each binary digit of the number is multiplied by its position weight (2^n, where n is the weight of the bit) and the product terms are added to obtain the decimal number.
- Example:- $(101011.11)_2 = (\ )_{10}$

  $= 1x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2}$

  $= 32 + 0 + 8 + 0 + 2 + 1 + 0.5 + 0.25$

  $= 43.75$

  Hence, $(101011.11)_2 = (43.75)_{10}$

1.2.3. Decimal to Octal conversion

- The decimal integer is converted to the octal integer number by successive division by 8, and the decimal fraction is converted to the octal fraction number by successive multiplication by 8.
- In the successive division-by-8 method, the given decimal integer number is successively divided by 8 till the quotient is 0.
- The remainders read from bottom to top give the equivalent octal integer number.
- In the successive multiplication-by-8 method, the given decimal fraction and the subsequent fractions are successively multiplied by 8, till the fraction part of the product is 0 or till the desired accuracy is obtained.
- The integers read from top to bottom give the equivalent octal integer number.
- To convert a mixed number to octal, convert the integer and fraction parts separately to octal and then combine them.
- Example:- $(125.6875)_{10} = (\ )_8$

8	125	5
8	15	7
8	1	1
	0	

$0.6875 \times 8 = 5.5000 \quad 5 + 0.5000$
$0.5000 \times 8 = 4.0000 \quad 4 + 0.0000$

Hence, $(125.6875)_{10} = (175.54)_8$

1.2.4. Octal to Decimal conversion

- Octal numbers may be converted to their decimal equivalents by the positional weights method.
- In this method, each octal digit of the number is multiplied by its position weight (8^n, where n is the weight of the bit) and the product terms are added to obtain the decimal number.
- Example:- $(724.25)_8 = (\ )_{10}$

$= 7x8^2 + 2x8^1 + 4x8^0 + 2x8^{-1} + 5x8^{-2}$
$= 448 + 16 + 4 + 0.25 + 0.0781$
$= 468.3281$

Hence, $(724.25)_8 = (468.3281)_{10}$

1.2.5. Octal to Decimal conversion

- The decimal integer is converted to the hexadecimal integer number by successive division by 16, and the decimal fraction is converted to the hexadecimal fraction number by successive multiplication by 16.
- In the successive division-by-16 method, the given decimal integer number is successively divided by 16 till the quotient is 0.
- The remainders read from bottom to top give the equivalent hexadecimal integer number.

- In the successive multiplication-by-16 method, the given decimal fraction and the subsequent fractions are successively multiplied by 16, till the fraction part of the product is 0 or till the desired accuracy is obtained.
- The integers read from top to bottom give the equivalent hexadecimal integer number.
- To convert a mixed number to hexadecimal, convert the integer and fraction parts separately to hexadecimal and then combine them.
- Example:- $(2598.675)_{10} = (\ )_{16}$

16	2598	6
16	162	2
16	10	10(A)
	0	

$0.6750 \times 16 = 10.8000 \rightarrow 10(A) + 0.8000$
$0.8000 \times 16 = 12.8000 \rightarrow 12(C) + 0.8000$
$0.8000 \times 16 = 12.8000 \rightarrow 12(C) + 0.8000$
$0.8000 \times 16 = 12.8000 \rightarrow 12(C) + 0.8000$

Hence, $(2598.675)_{10} = (A26.ACCC)_{16}$

1.2.6. Hexadecimal to Decimal conversion

- Hexadecimal numbers may be converted to their decimal equivalents by the positional weights method.
- In this method, each hexadecimal digit of the number is multiplied by its position weight (16^n, where n is the weight of the bit) and the product terms are added to obtain the decimal number.
- Example:- $(A0F9.0EB)_{16} = (\ )_{10}$
  $= 10x16^3 + 0x16^2 + 15x16^1 + 9x16^0 + 0x16^{-1} + 14x16^{-2} + 11x16^{-3}$
  $= 40960 + 0 + 240 + 9 + 0 + 0.0546 + 0.0026$
  $= 41209.0572$
- Hence, $(A0F9.0EB)_{16} = (41209.0572)_{10}$

1.2.7. Octal to Binary Conversion

- To convert a given octal number to a binary, just replace each octal digit by its 3-bit binary equivalent as per below table.

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

- Example:- $(367.52)_8 = (\ )_2$
  $= 011\ \ 110\ \ 111\ .\ 101\ \ 010$
  $= 11110111.10101$
  Hence, $(367.52)_8 = (11110111.10101)_2$

1.2.8. Binary to Octal Conversion

- To convert a binary number to an octal number, starting from the binary point make groups of 3 bits each (i.e. from point (".") in binary number, group of 3 bits in left side and group of 3 bits in right side), if there are not 3 bits available at last, just stuff "0" to make 3 bits group.
- Replace each 3-bit binary group by the equivalent octal digit.
- Example:- $(110101.101010)_2 = (\ )_8$
  $= 110\ \ 101\ .\ 101\ \ 010$
  $= 65.52$
- Hence, $(110101.101010)_2 = (65.52)_8$

1.2.9. Hexadecimal to Binary Conversion

- To convert a given hexadecimal number to a binary, just replace each hexadecimal digit by its 4-bit binary equivalent as per below table.

Hexadecimal Number	Binary Number
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

- Example:- $(3A9E.B0D)_{16} = (\ )_2$
  $= 0011\ \ 1010\ \ 1001\ \ 1110\ .\ 1011\ \ 0000\ \ 1101$
  $= 11101010011110.101100001101$
  Hence, $(3A9E.B0D)_{16} = (11101010011110.101100001101)_2$

1.2.10. Binary to Hexadecimal Conversion

- To convert a binary number to a hexadecimal number, starting from the binary point make groups of 4 bits each (i.e. from point (".") in binary number, group of 4 bits in left side and group of 4 bits in right side), if there are not 4 bits available at last, just stuff "0" to make 4 bits group.
- Replace each 4-bit binary group by the equivalent hexadecimal digit.
- Example:- $(01011111011.011111)_2 = (\ )_{16}$
  $= 0010\ 1111\ 1011.\ 0111\ \ 1100$
  $= 2FB.7C$
  Hence, $(01011111011.011111)_2 = (2FB.7C)_{16}$

1.2.11. Octal to Hexadecimal Conversion

- To convert an octal number to hexadecimal, the simplest way is to first convert the given octal number to binary and then the binary number to hexadecimal. (i.e. first from table of 3-bit and then table of 4-bit)
- Example :- $(756.603)_8 = (\ )_{16}$
  $=\ 7\ \ 5\ \ 6\ .\ 6\ \ 0\ \ 3$
  $\ \ 111\ \ 101\ \ 110\ .\ 110\ \ 000\ \ 01$
  $= \underline{0001}\ \underline{1110}\ \underline{1110}\ .\ \underline{1100}\ \underline{0001}\ \underline{1000}$
  $\ \ 1\ \ E\ \ E\ \ .\ \ C\ \ 1\ \ 8$
- Hence, $(756.603)_8 = (1EE.C18)_{16}$

1.2.12. Hexadecimal to Octal Conversion

- To convert a hexadecimal number to octal, the simplest way is to first convert the given hexadecimal number to binary and then the binary number to octal. (i.e. first from table of 4-bit and then table of 3-bit)
- Example :- $(B9F.AE)_{16} = (\ )_8$
  $=\ B\ \ 9\ \ F\ \ .\ A\ \ E$
  $\ \ 1011\ \ 1001\ \ 1111\ .\ 1010\ \ 1110$

  $= \underline{101}\ \underline{110}\ \underline{011}\ \underline{111}\ .\ \underline{101}\ \underline{011}\ \underline{100}$
  $\ \ 5\ \ 6\ \ 3\ \ 7\ \ .\ \ 5\ \ 3\ \ 4$
  Hence, $(B9F.AE)_{16} = (5637.534)_8$

1.2.13. Accuracy in Binary Number conversion

- Example:- Convert $(0.252)_{10}$ to binary with an error less than 1%
- Absolute value of allowable error is found by calculating 1% of the number.
$$E_{allow} = 0.01\ x\ 0.252\ =\ 0.00252_{10}$$
- Maximum error due to truncation is set to be less than allowable error by solving from $E_{10} = 2^{-n}$
- This equation is written as
$$2^{-n} < 0.00252$$
- Inverting both sides of the inequality
$$2^n > 397$$

- Taking log of both sides and solving for n

$$n \log 2 = \log 397$$

$$n = \frac{\log 397}{\log 2} = 8.63 \approx 9 \text{ (next largest integer)}$$

- This indicates that the use of 9 bits in the binary number will guarantee an error less than 1%.
- So, the conversion is carried out to 9 places.

0.252 x 2 = 0.504 0
0.504 x 2 = 1.008 1
0.008 x 2 = 0.016 0
0.016 x 2 = 0.032 0
0.032 x 2 = 0.064 0
0.064 x 2 = 0.128 0
0.128 x 2 = 0.256 0
0.256 x 2 = 0.512 0
0.512 x 2 = 1.024 1

- Therefore, $(0.252)_{10} = (0.010000001)_2$

1.2.14. Complement forms

- 9's Complement
  - The 9's complement of a decimal number is obtained by subtracting each digit of that decimal number from 9.
  - Example:- 782.54

```
  9 9 9 . 9 9
- 7 8 2 . 5 4
-------------
  2 1 7 . 4 5 (9's complement of 782.54)
```

- 10's Complement
  - The 10's complement of a decimal number is obtained by adding a 1 to its 9's complement.
  - *Shortcut:- Subtract LSB(Least Significant Bit) from 10 and rest of the digits from 9.*
  - Example:- 1056.074

```
  9 9 9 9 . 9 9 9
-1 0 5 6 . 0 7 4
---------------
  8 9 4 3 . 9 2 5
                +1
---------------
  8 9 4 3 . 9 2 6 (10's complement of 1056.074)
```

- 1's Complement
  - The 1's complement of a binary number is obtained by subtracting each digit of that binary number from 1.

- *Shortcut: Change 1's to 0's and 0's to 1's in binary number.*
- Example:- 101101.1001

```
  1 1 1 1 1 1 . 1 1 1 1
-1 0 1 1 0 1 . 1 0 0 1
---------------------
  0 1 0 0 1 0 . 0 1 1 0
```

- 1's complement of 101101.1001 is 010010.0110

- 2's Complement
  - The 2's complement of a binary number is obtained by adding a 1 to its 1's complement.
  - *Shortcut: A shortcut to manually convert a binary number into its 2's complement is to start at the least significant bit (LSB), and copy all the zeros, working from LSB toward the most significant bit (MSB) until the first 1 is reached; then copy that 1, and flip all the remaining bits.*
  - Example:- 110101.10100

```
    1 1 1 1 1 1 . 1 1 1 1 1
-   1 1 0 1 0 1 . 1 0 1 0 0
---------------------------
    0 0 1 0 1 0 . 0 1 0 1 1
                        +1
---------------------------
    0 0 1 0 1 0 . 0 1 1 0 0
```

  - 2's complement of 110101.10100 is 0 0 1 0 1 0 . 0 1 1 0 0

1.2.15. Signed Number Representation

- The 2's complement system for representing signed numbers works like this:
  1) If the number is positive, the magnitude is represented in its true binary form and a sign bit 0 is placed in front of the MSB.
  2) If the number is negative, the magnitude is represented in its 2's complement form and a sign bit 1 is placed in front of the MSB.
- Example:-

(1) Express -45 in 8-bit 2's complement form.

Ans. +45 in 8-bit form is 00101101 (By taking decimal to binary conversion).

take 2's complement of it, 11010011

Hence, -45 in 2's complement form is 11010011

(2) Express -73.25 in 12-bit 2's complement form.

Ans. +73.25 in 12-bit form is 01001001.1100 (By taking decimal to binary conversion).

take 2's complement of it, 10110110.0100

Hence, -73.25 in 2's complement form is 10110110.0100

1.2.16. Subtraction using complements form

- Subtraction using 9's complement form
  - To perform decimal subtraction using the 9's complement method, obtain 9's complement of the subtrahend and add it to the minuend.
  - Call this number the intermediate result.
  - If there is a carry, it indicates that the answer is positive.

- Add the carry to the LSD of this result to get the answer.
- If there is no carry, it indicates that the answer is negative and the intermediate result is its 9's complement.
- Take the 9's complement of this result and place a negative sign in front to get the answer.
- Example:-

(1) 745.81 – 436.62

Ans. 7 4 5 . 8 1 → 7 4 5 . 8 1
4 3 6 . 6 2 + 5 6 3 . 3 7 (9's complement of 436.62)
3 0 9 . 1 9 (1)3 0 9 . 1 8 (Intermediate result)
→ + 1
3 0 9 . 1 9 (Answer)

(2) 436.62 – 745.81

Ans. 4 3 6 . 6 2 → 4 3 6 . 6 2
- 7 4 5 . 8 1 + 2 5 4 . 1 8(9' complement of 745.81)
-3 0 9 . 1 9 6 9 0 . 8 0 (Intermediate result)

There is no carry indicating that the answer is negative. So, take the 9's complement of the intermediate result and put a minus sign. Therefore, the answer is -309.19

- Subtraction using 10's complement form
  - To perform decimal subtraction using the 10's complement method, obtain the 10's complement of the subtrahend and add it to the minuend.
  - If there is a carry, ignore it.
  - The presence of the carry indicates that the answer is positive; the result obtained is itself the answer.
  - If there is no carry, it indicates that the answer is negative and the result obtained is its 10's complement.
  - Obtain the 10's complement of the result and place a negative sign in front to get the answer.
  - Example:-

(1) 2928.54 – 416.73

Ans. 2928.54 2928.54
-0416.73 → +9583.27 (10's complement of 416.73)
12511.81 (1)2511.81 (Ignore the carry)

(2) 416.73 – 2928.54

Ans. 0416.73 0416.73
-2928.54 → +7071.46 (10's complement of 2928.54)
-2511.81 7488.19 (No carry)

  - There is no carry indicating that the answer is negative. So, take the 10's complement of the intermediate result and put a minus sign. Therefore, the answer is -2511.81
- Subtraction using 1's complement form
  - To perform binary subtraction using the 1's complement method, obtain 1's

complement of the subtrahend and add it to the minuend.

- Call this number the intermediate result.
- If there is a carry, it indicates that the answer is positive.
- Add the carry to the LSB of this result to get the answer.
- If there is no carry, it indicates that the answer is negative and the intermediate result is its 1's complement.
- Take the 1's complement of this result.
- Example:-

(1) 11010 – 1101

Ans. 11010 11010

\- 01101 → +10010 (1's complement)

101100 (Intermediate result)

→ +1

01101 (Answer)

(2) 100 – 110000

Ans. 000100 0000100

-110000 + 1001111 (1's complement)

1010011 (Intermediate result)

There is no carry. The MSB is 1. Hence, the answer is negative. Take the 1's complement of the remaining bits. So, it is 101100 (negative).

- Subtraction using 2's complement form
  - To perform binary subtraction using the 2's complement method, obtain the 2's complement of the subtrahend and add it to the minuend.
  - If there is a carry, ignore it.
  - The presence of the carry indicates that the answer is positive; the result obtained is itself the answer.
  - If there is no carry, it indicates that the answer is negative and the result obtained is its 2's complement.
  - Obtain the 2's complement of the result.
  - Example:-

(1) 11011 – 1101

Ans. 11011 11011

-01101 → + 10011 (2's complement)

101101 (Ignore the carry, result is positive)

(2) 100 – 110000

Ans. 000100 0000100

\- 110000 → +1010000 (2's complement)

1010100 (Result is negative)

There is no carry. The MSB is 1. Hence, the answer is negative. Take its 2's complement and put a minus sign. So it is, 101100 (negative).

1.2.17. Binary Operation

- Binary Addition

  $0 + 0 = 0$; $0 + 1 = 1$; $1 + 0 = 1$; $1 + 1 = 10$ (i.e. 0 with a carry of 1); $1 + 1 + 1 = 11$

  Example:- Add the binary numbers 1101.101 and 111.011.

  ```
  1  1  1  1  1   1  1 ←Carry
     1  1  0  1 . 1  0  1
  +  0  1  1  1 . 0  1  1
  ---------------------------
   1  0  1  0  1 . 0  0  0
  ```

- Binary Subtraction

  $0 - 0 = 0$; $1 - 1 = 0$; $1 - 0 = 1$; $0 - 1 = 1$ with a borrow of 1.

  Example:- Subtract 111.111 from 1010.01

  ```
  0  1  10 1     1 10  10   ← Borrow
  1  0  1  0  .  0  1  0
  0  1  1  1  .  1  1  1
  ---------------------------
  0  0  1  0  .  0  1  1
  ```

- Binary Multiplication

  Example:- Multiply 1011.101 by 101.01.

  ```
    1 0 1 1 . 1 0 1
  X   1 0 1 . 0 1 0
  -------------------
        1 0 1 1 1 0 1
      0 0 0 0 0 0 0 0
    1 0 1 1 1 0 1 0 0
  0 0 0 0 0 0 0 0 0 0
  -------------------------
  1 0 1 1 1 0 1 0 0 0 0
  -------------------------
  1 1 1 1 0 1.0 0 0 0 1
  ```

  Therefore, 1011.101 x 101.010 = 111101.00001

- Binary Division

  Example: - Divide 101101 by 110.

  ```
  1 1 0) 1 0 1 1 0 1 (1 1 1 . 1
         1 1 0
         -------
         1 0 1 0
           1 1 0
           -------
           1 0 0 1
             1 1 0
             -------
               1 1 0
               1 1 0
               -------
               0 0 0
  ```

  Therefore, 101101 / 110 = 111.1

1.3. Codes

1.3.1. BCD Code

- In this code, each decimal digit, 0 through 9, is coded by 4-bit binary number.
- It is a weighted code and is also sequential. Therefore, it is useful for mathematical operations.
- The main advantage of this code is its ease of conversion to and from decimal.
- It is less efficient than the pure binary, in the sense that it requires more bits.
- For example, the decimal number 14 can be represented as 1110 in pure binary but as 0001 0100 in 8421 BCD code.
- Another disadvantage of the BCD code is that, arithmetic operations are more complex than they are in pure binary.
- There are six illegal combinations 1010, 1011, 1100, 1101, 1110 and 1111 in this code.

BCD Addition

- If there is no carry and the sum term is not an illegal code, no correction is needed.
- If there is a carry out of one group to the next group, or if the sum term is illegal code, then 6 (0110) is added to the sum term of that group and the resulting carry is added to the next group.
- Example:-

(1) 25 + 13

Ans. 25 ⇨ 0010 0101 (25 in BCD)
+ 13 + 0001 0011 (13 in BCD)
38 0011 1000 (No carry, no illegal code. So, this is the correct sum)

(2) 679.6 + 536.8

Ans. 679.6 ⇨ 0110 0111 1001. 0110 (679.6 in BCD)
+ 536.8 + 0101 0011 0110. 1000 (536.8 in BCD)
1216.4 1011 1010 1111. 1110 (All are illegal codes)
+0110 +0110 +0110.+0110(Add 0110 to each)
0001 0010 0001 0110. 0100 (Corrected sum = 1216.4)

BCD Subtraction

- If there is no borrow from the next higher group then no correction is required.
- If there is a borrow from the next group, then 6_{10} (0110) is subtracted from the difference term of this group.
- Example:-

(1) 38 – 15

Ans. 38 ⇨ 0011 1000 (38 in BCD)
-15 0001 0101 (15 in BCD)
23 0010 0011(No borrow.So, this is the correct difference)

(2) 206.7 – 147.8

Ans. 206.7 ⇨ 0010 0000 0110. 0111 (206.7 in BCD)

-147.8 0001 0100 0111. 1000 (147.8 in BCD)

58.9 0000 1011 1110. 1111(Borrows are present, subtract 0110)

- 0110 -0110. -0110

0101 1000. 1001(Corrected difference = 58.9)

1.3.2. Excess 3 Code

- The Excess-3 code, also called XS-3, is a non-weighted BCD code.
- This code derives its name from the fact that each binary code word is the corresponding 8421 code word plus 0011(3).
- It is sequential code and, therefore, can be used for arithmetic operations.
- It is a self-complementing code.
- The XS-3 code has six invalid states 0000, 0001, 0010, 1101, 1110 and 1111.

XS-3 Addition

- If there is no carry out from the addition of any of the 4-bit groups, subtract 0011 from the sum term of those groups.
- If there is a carry out, add 0011 to the sum term of those groups.
- Example:-

(1) 247.6 + 359.4

Ans. 247.6 ⇨ 0101 0111 1010. 1001 (247.6 in XS-3)

+359.4 + 0110 1000 1100. 0111 (359.4 in XS-3)

607.0 1100 0000 0111. 0000 (Add 0011 to 0000,0111,0000

-0011+ 0011+0011.+ 0011 and subtract 0011 from 1100)

1001 0011 1010 0011(Corrected sum in XS-3=607.0)

XS-3 Subtraction

- If there is no borrow from the next 4-bit group, add 0011 to the difference term of such groups.
- If there is a borrow, subtract 0011 from the difference term.
- Example:-

(1) 267 – 175

Ans. 267 ⇨ 0101 1001 1010 (267 in XS-3)

-175 0100 1010 1000 (175 in XS-3)

092 0000 1111 0010 (Subtract 0011 from 1111 and

+0011-0011+0011 add 0011 to 0000 & 0010)

0011 1100 0101 (Corrected difference in XS-3=92)

1.3.3. Gray Code

- The gray code is a non-weighted code.
- It is a cyclic code because successive code words in this code differ in one bit position only, i.e. it is a unit distance code.
- It is also a reflective code.
- The n least significant bits for 2^n through 2^{n+1}-1 are the mirror images of those for 0 through 2^n-1.
- An N-bit gray code can be obtained by reflecting an N-1 bit code about an axis at the end of the code, and putting the MSB of 0 above the axis and the MSB of 1 below the axis.
- One reason for the popularity of the gray code is its ease of conversion to and from binary.
- Reflection of gray code is shown in table.

Gray Code				Decimal	4-bit binary
1-bit	2-bit	3-bit	4-bit		
0	00	000	0000	0	0000
1	01	001	0001	1	0001
	11	011	0011	2	0010
	10	010	0010	3	0011
		110	0110	4	0100
		111	0111	5	0101
		101	0101	6	0110
		100	0100	7	0111
			1100	8	1000
			1101	9	1001
			1111	10	1010
			1110	11	1011
			1010	12	1100
			1011	13	1101
			1001	14	1110
			1000	15	1111

Binary to Gray Conversion

- If an n-bit binary number is represented by $B_n\ B_{n-1}$ … B_1 and its gray code equivalent $G_n\ G_{n-1}$ … G_1, where B_n and G_n are the MSBs, then the gray code bits are obtained from the binary code as follows:

$G_n = B_n$	$G_{n-1} = B_n \oplus B_{n-1}$	$G_{n-2} = B_{n-1} \oplus B_{n-2}$	……	$G_1 = B_2 \oplus B_1$

- The conversion procedure is as follows:
  1. Record the MSB of the binary as the MSB of the gray code.
  2. Perform X-ORing between the MSB of the binary and the next bit in binary. This answer is the next bit of the gray code.

3. Perform X-ORing between 2nd bit of the binary and 3rd bit of the binary, the 3rd bit with the 4th bit, and so on.
4. Record the successive answer bits as the successive bits of the gray code until all the bits of the binary number are exhausted.

- Example: - Convert the binary 1001 to Gray code.

Ans. 1 ⊕ 0 ⊕ 0 ⊕ 1

1 1 0 1

Gray to Binary Conversion

- If an n-bit gray number is represented by G_n G_{n-1} … G_1 and its binary code equivalent B_n B_{n-1} … B_1, where G_n and B_n are the MSBs, then the binary bits are obtained from the gray bits as follows:

$B_n = G_n$	$B_{n-1} = B_n \oplus G_{n-1}$	$B_{n-2} = B_{n-1} \oplus G_{n-2}$	……	$B_1 = B_2 \oplus G_1$

- The conversion procedure is as follows:
  1. The MSB of the binary number is the same as the MSB of the gray code number.
  2. Perform X-ORing between the MSB of the binary and next significant bit of gray code. This answer is the next bit of binary.
  3. Perform X-ORing between the 2nd bit of the binary and 3rd bit of the gray code, the 3rd bit of the binary with the 4th bit of gray code, and so on.
  4. Record the successive answers as the successive bits of the binary until all the bits of the gray code are exhausted.

- Example:- Convert the gray code 1101 to binary.

Ans.

1 1 0 1

⊕ ⊕ ⊕

1 0 0 1

1.3.4. Error Detecting Code

- When binary data is transmitted and processed, it is susceptible to noise that can alter or distort its contents.
- The 1s may get changed to 0s and 0s to 1s.
- Because digital systems must be accurate to the digit, errors can pose a serious problem.
- Several schemes have been devised to detect the occurrence of a single-bit error in a binary word, so that whenever such an error occurs the concerned binary word can be corrected and retransmitted.

Parity

- The simplest technique for detecting errors is that of adding an extra bit, known as the *parity* bit, to each word being transmitted.
- There are two types of parity – odd parity and even parity.
- For odd parity, the parity bit is set to a 0 or a 1 at the transmitter such that the total number of 1 bits in the word including the parity bit is an odd number.
- For even parity, the parity bit is set to a 0 or a 1 at the transmitter such that the total number of 1 bits in the word including the parity bit is an even number.
- When the digital data is received, a parity checking circuit generates an error signal if the total number of 1s is even in an odd-parity system or odd in an even-parity system.
- This parity check can always detect a single-bit error but cannot detect two or more errors within the same word.
- In any practical system, there is always a finite probability of the occurrence of single error.
- Odd parity is used more often than even parity because even parity does not detect the situation where all 0s are created by a short-circuit or some other fault condition.
- Example:- If Data is 1011010, then even parity must be 0 and odd parity must be 1.

Block Parity

- When several binary words are transmitted or stored in succession, the resulting collection of bits can be regarded as a block of data, having rows and columns.
- Parity bits can then be assigned to both rows and columns.
- This scheme makes it possible to *correct* any single error occurring in a data word and to *detect* any two errors in a word.
- This technique also called word parity, is widely used for data stored on magnetic tapes.
- For example, six 8-bit words in succession can be formed into a 6x8 block for transmission.
- Parity bits are added so that odd parity is maintained both row-wise and column-wise and the block is transmitted as a 7x9 block as shown in Figure 1.
- At the receiving end, parity is checked both row-wise and column-wise and suppose errors are detected as shown in Figure 2.
- These single-bit errors detected can be corrected by complementing the error bit.
- In Figure 2, parity errors in the 3rd row and 5th column mean that the 5th bit in 3rd row is in error.
- It can be corrected by complementing it.
- Two errors as shown in Figure 3 can only be detected but not corrected.
- In Figure 3, parity errors are observed in both columns 2 and 4.
- It indicated that in one row there two errors.

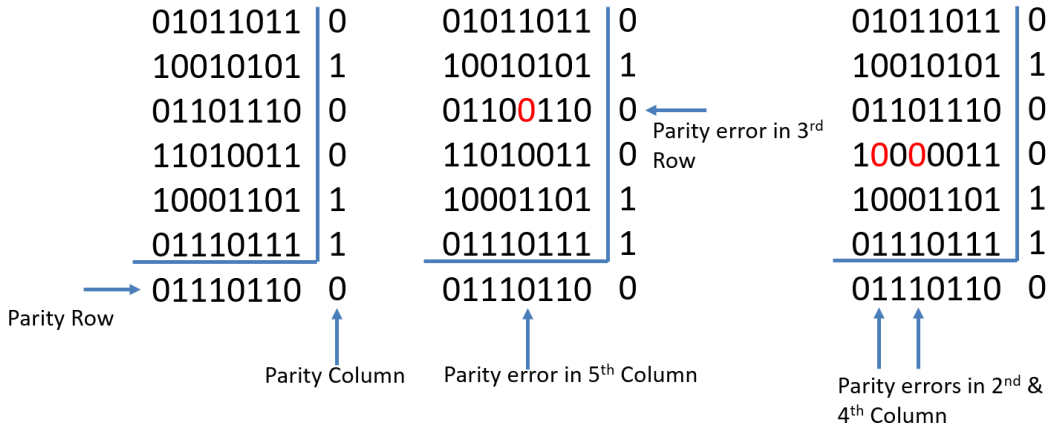


Figure 1 Figure 2 Figure 3

1.3.5. Error Correcting Code

- A code is said to be an error-correcting code, if the correct code word can always be deducted from an erroneous word.
- For a code to be a single-bit error-correcting code, the minimum distance of that code must be three.
- The minimum distance of a code is the smallest number of bits by which any two code words must differ.
- A code with minimum distance of three can not only correct single-bit errors, but also detect (but cannot correct) two-bit errors.
- The key to error correction is that it must be possible to detect and locate erroneous digits.
- If the location of an error correction is determined, then by complementing the erroneous digit, the message can be corrected.
- 7-bit hamming code is one type of error-correcting code.

The 7-bit hamming code

- To transmit four data bits, three parity bits located at positions 2^0, 2^1, and 2^2 from left are added to make a 7-bit code word which is then transmitted.
- The word format would be as shown below:

  $$\mathbf{P_1\ P_2\ D_3\ P_4\ D_5\ D_6\ D_7}$$

  Where the D bits are the data bits and the P bits are the parity bits.
- P_1 is to be set a 0 or 1 so that it establishes even parity bits 1, 3, 5, and 7 (i.e. P_1 D_3 D_5 D_7).
- P_2 is to be set a 0 or 1 so that it establishes even parity bits 2, 3, 6, and 7 (i.e. P_2 D_3 D_6 D_7).
- P_4 is to be set a 0 or 1 so that it establishes even parity bits 4, 5, 6, and 7 (i.e. P_4 D_5 D_6 D_7).
- Example:-
  (1) Encode data bits 1101 into the 7-bit even-parity hamming code.
  Ans: The bit pattern is P_1 P_2 D_3 P_4 D_5 D_6 D_7 = P_1 P_2 1 P_4 1 0 1

Bits 1, 3, 5, 7 (i.e. P_1 1 1 1) must have even parity. So, P_1 must be a 1.
Bits 2, 3, 6, 7 (i.e. P_2 1 0 1) must have even parity. So, P_2 must be a 0.
Bits 4, 5, 6, 7 (i.e. P_4 1 0 1) must have even parity. So, P_4 must be a 0.

Therefore, the final code is 1 0 1 0 1 0 1.

(2) The message 1001001 in the 7-bit hamming code is transmitted through a noisy channel. Decode the message assuming that at most a single error occurred in each code word.

Ans. Message is 1001001

Bits 1, 3, 5, 7 (1001) $\longrightarrow$ no error $\longrightarrow$ $C_1 = 0$
Bits 2, 3, 6, 7 (0001) $\longrightarrow$ error $\longrightarrow$ $C_2 = 1$
Bits 4, 5, 6, 7 (1001) $\longrightarrow$ no error $\longrightarrow$ $C_3 = 0$
The error word is $C_3C_2C_1 = 010 = 2_{10}$. So, complement the 2^{nd} bit from left.
Therefore, the correct code is 1 1 0 1 0 0 1.

Unit-2
Simplification of Logic Function

2.1. Standard representation for logic functions

2.1.1. Sum-of-products (SOP) form:

- This form is also called the Disjunctive Normal Form (DNF).
- For example, $f(A,B,C) = \bar{A}B + \bar{B}C$

2.1.2. Product-of-sums (POS) form:

- This form is also called the Conjunctive Normal Form (CNF).
- The function of above equation may also be written in the form shown in equation below.
- By using multiplying it out and using the consensus theorem, we can see that it is the same as

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

2.1.3. Standard sum-of-products form:

- This form is also called Disjunctive Canonical Form (DCF).
- It is also called the Expanded Sum of Products Form or Canonical Sum-of-Products Form.
- In this form, the function is the sum of a number of product terms where each product term contains all the variables of the function either in complemented or uncomplemented form.
- This can be derived from the truth table by finding the sum of all the terms that correspond to those combinations (rows) for which 'f ' assumes the value 1.
- It can also be obtained from the SOP form algebraically as shown below.

$$f(A,B,C) = \bar{A}B + \bar{B}C = \bar{A}B\,(C + \bar{C}) + (A + \bar{A})\bar{B}$$
$$= \bar{A}BC + \bar{A}B\bar{C} + A\bar{B}C + \bar{A}\bar{B}C$$

- A product term which contains all the variables of the function either in complemented or uncomplemented form is called a minterm.
- A minterm assumes the value 1 only for one combination of the variables. An n variable function can have in all 2^n minterms.
- The sum of the minterms whose value is equal to 1 is the standard sum of products form of the function.
- The minterms are often denoted as m_0, m_1, m_2, ..., where the suffixes are the decimal codes of the combinations.
- For a 3-variable function $m_0 = \bar{A}\bar{B}\bar{C}$, $m_1 = \bar{A}\bar{B}C$, $m_2 = \bar{A}B\bar{C}$, $m_3 = \bar{A}BC$, $m_4 = A\bar{B}\bar{C}$, $m_5 = A\bar{B}C$, $m_6 = AB\bar{C}$, $m_7 = ABC$.
- Another way of representing the function in canonical SOP form is by showing the sum of minterms for which the function equals 1.

- Thus $f(A,B,C) = m_1 + m_2 + m_3 + m_5$
- Yet another way of representing the function in DCF is by listing the decimal codes of the minterms for which f = 1.
- Thus $f(A,B,C) = \sum_m(1,2,3,5)$

2.1.4. Standard product-of-sums form:

- This form is also called Conjunctive Canonical Form (CCF).
- It is also called Expanded Product-of-Sums Form or Canonical Product-of-Sums Form.
- This is derived by considering the combinations for which f = 0. Each term is a sum of all the variables.
- A variable appears in uncomplemented form if it has a value of 0 in the combination and appears in complemented form if it has a value of l in the combination.

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

- A sum term, which contains each of the n variables in either complemented or uncomplemented form, is called a maxterm.
- A maxterm assumes the value 0 only for one combination of the variables. For all other combinations it will be 1.
- There will be at the most 2^n maxterms. The product of maxterms corresponding to the rows for which f = 0, is the standard or canonical product of sums form of the function.
- Maxterms are often represented as M_0, M_1, M_2, ..., where the suffixes denote their decimal code.
- Thus, the CCF of function f may be written as $f(A,B,C) = M_0 \cdot M_4 \cdot M_6 \cdot M_7$
- Expression can also be expressed as $f(A,B,C) = \prod_M(0,4,6,7)$
- Where represents the product of all maxterms whose decimal code is given within the parenthesis.

2.2. Boolean Algebra Laws

- AND laws

1. $A \cdot 0 = 0\ (Null\ Law)$
2. $A \cdot 1 = A\ (Identity\ Law)$
3. $A \cdot A = A$
4. $A \cdot \bar{A} = 0$

- OR laws

1. $A + 0 = A\ (Null\ Law)$
2. $A + 1 = 1\ (Identity\ Law)$
3. $A + A = A$
4. $A + \bar{A} = 1$

- Commutative law

1. $A + B = B + A$
2. $A \cdot B = B \cdot A$

- Associative laws

1. $(A + B) + C = A + (B + C)$

2. $(A \cdot B)C = A(B \cdot C)$

- Distributive laws

1. $A(B + C) = AB + AC$
2. $A + BC = (A + B)(A + C)$

- Redundant Literal Rule

1. $A + \bar{A}B = A + B$
2. $A(\bar{A} + B) = AB$

- Idempotence laws

1. $A \cdot A = A$
2. $A + A = A$

- Absorption laws

1. $A + AB = A$
2. $A(A + B) = A$

2.2.1. De Morgan's Theorem

1. Law 1 : $\overline{A + B + C} = \bar{A}\,\bar{B}\,\bar{C}$
   - This law states that the complement of a sum of variables is equal to the product of their individual complements.

A	B	C	A+B+C	(A+B+C)'	A'	B'	C'	A'B'C'
0	0	0	0	**1**	1	1	1	**1**
0	0	1	1	**0**	1	1	0	**0**
0	1	0	1	**0**	1	0	1	**0**
0	1	1	1	**0**	1	0	0	**0**
1	0	0	1	**0**	0	1	1	**0**
1	0	1	1	**0**	0	1	0	**0**
1	1	0	1	**0**	0	0	1	**0**
1	1	1	1	**0**	0	0	0	**0**

   - Hence, Law 1 is proved from the above truth table.
2. Law 2 : $\overline{A\,B\,C} = \bar{A} + \bar{B} + \bar{C}$
   - This law states that the complement of a product of variables is equal to the sum of their individual complements.

A	B	C	A B C	(A B C)'	A'	B'	C'	A'+B'+C'
0	0	0	0	**1**	1	1	1	**1**
0	0	1	0	**1**	1	1	0	**1**
0	1	0	0	**1**	1	0	1	**1**
0	1	1	0	**1**	1	0	0	**1**
1	0	0	0	**1**	0	1	1	**1**
1	0	1	0	**1**	0	1	0	**1**
1	1	0	0	**1**	0	0	1	**1**
1	1	1	1	**0**	0	0	0	**0**

   Hence, Law 2 is proved from the above truth table.

2.2.2. Reduction of Boolean Expression

1. $f = A[B + \bar{C}\,(\overline{AB + A\bar{C}})]$

$= A[B + \bar{C}\,(\overline{AB}\;\overline{A\bar{C}})]$ (De Morgan's Theorem)
$= A[B + \bar{C}\,(\bar{A} + \bar{B})(\bar{A} + C)]$ (De Morgan's Theorem)
$= A[B + \bar{C}\,(\bar{A}\bar{A} + \bar{A}C + \bar{B}\bar{A} + \bar{B}C)]$ (Distributive Law)
$= A[B + \bar{C}\,(\bar{A} + \bar{A}C + \bar{A}\bar{B} + \bar{B}C)]$ (A' · A' = A')
$= A(B + \bar{C}\bar{A} + \bar{C}\bar{A}C + \bar{C}\bar{A}\bar{B} + \bar{C}\bar{B}C)$ (Distributive Law)
$= A(B + \bar{A}\bar{C} + 0 + \bar{A}\bar{B}\bar{C} + 0)$ (C · C' = 0)
$= AB + A\bar{A}\bar{C} + A\bar{A}\bar{B}\bar{C}$
$= AB$ (A · A' = 0)

2. $f = A + B[AC + (B + \bar{C})D]$
$= A + B[AC + BD + \bar{C}D]$ (Distributive Law)
$= A + BAC + BBD + B\bar{C}D$ (Distributive Law)
$= A + ABC + BD + B\bar{C}D$ (B · B = B)
$= A(1 + BC) + BD(1 + \bar{C})$
$= A \cdot 1 + BD \cdot 1$ (1 + A = A)
$= A + BD$

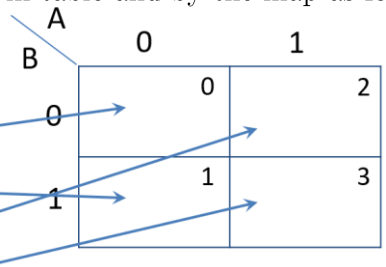
2.3. Karnaugh Map (K-Map)

- The Karnaugh map (K-map) method is a systematic method of simplifying the Boolean expression.
- The K-map is a chart or a graph, composed of an arrangement of adjacent cells, each representing a particular combination of variables in sum or product form.
- The output values placed in each cell are derived from the minterms of a Boolean function.
- A *minterm* is a product term that contains all of the function's variables exactly once, either complemented or not complemented.

Two-variable K-Map

- The two variable expression can have $2^2 = 4$ possible combinations of the input variables A and B.
- Each of these combinations, A'B', A'B, AB', and AB (in SOP form) is called minterm.
- These possible combinations can be represented in table and by the map as follows:

A	B	Minterm
0	0	$m_0 = A'B'$
0	1	$m_1 = A'B$
1	0	$m_2 = AB'$
1	1	$m_3 = AB$



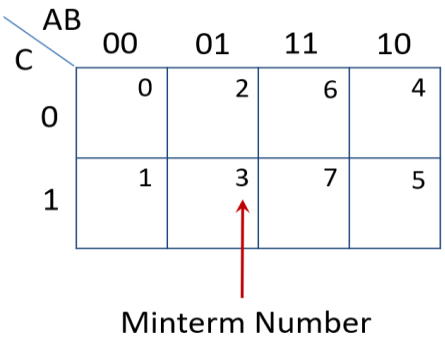
Three-variable K-Map

- A function in three variable (A, B, C) expressed in the standard SOP form can have eight

possible combinations: A'B'C', A'B'C, A'BC', A'BC, AB'C', AB'C, ABC' and ABC.

- Each one of these combinations designated by m_0, m_1, m_2, m_3, m_4, m_5, m_6, and m_7, respectively, is called minterm.
- In the standard POS form, the eight possible combinations are: A+B+C, A+B+C', A+B'+C, A+B'+C', A'+B+C, A'+B+C', A'+B'+C, and A'+B'+C'.
- Each one of these combinations designated by M_0, M_1, M_2, M_3, M_4, M_5, M_6, and M_7, respectively, is called maxterm.

A	B	C	Minterm
0	0	0	$m_0 = A'B'C'$
0	0	1	$m_1 = A'B'C$
0	1	0	$m_2 = A'BC'$
0	1	1	$m_3 = A'BC$
1	0	0	$m_4 = AB'C'$
1	0	1	$m_5 = AB'C$
1	1	0	$m_6 = ABC'$
1	1	1	$m_7 = ABC$

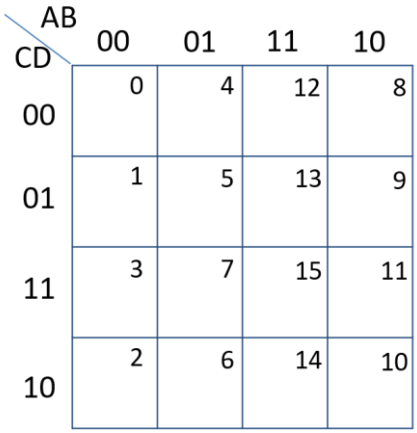


Four-variable K-Map

- The four variables A, B, C and D have sixteen possible combinations that can be represented by the map as follows

A	B	C	D	Minterm
0	0	0	0	$m_0 = A'B'C'D'$
0	0	0	1	$m_1 = A'B'C'D$
0	0	1	0	$m_2 = A'B'CD'$
0	0	1	1	$m_3 = A'B'CD$
0	1	0	0	$m_4 = A'BC'D'$
0	1	0	1	$m_5 = A'BC'D$
0	1	1	0	$m_6 = A'BCD'$
0	1	1	1	$m_7 = A'BCD$

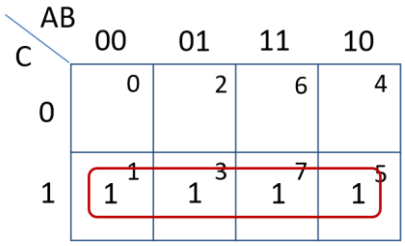
A	B	C	D	Minterm
1	0	0	0	$m_8 = AB'C'D'$
1	0	0	1	$m_9 = AB'C'D$
1	0	1	0	$m_{10} = AB'CD'$
1	0	1	1	$m_{11} = AB'CD$
1	1	0	0	$m_{12} = ABC'D'$
1	1	0	1	$m_{13} = ABC'D$
1	1	1	0	$m_{14} = ABCD'$
1	1	1	1	$m_{15} = ABCD$



2.4. Reduction using K-Map

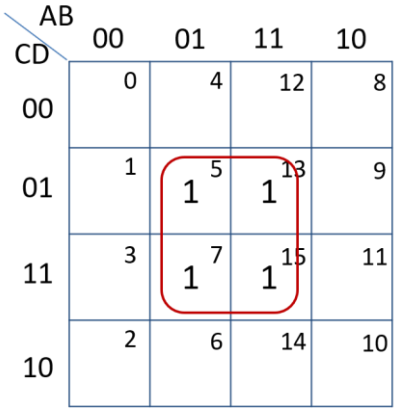
- Squares which are physically adjacent to each other or which can be made adjacent by wrapping the map around from left to right or top to bottom can be combined to form bigger squares.
- The bigger squares (2 squares, 4 squares, 8 squares, etc.) must form either a geometric square or rectangle.
- For the minterms or maxterms to be combinable into bigger squares, it is necessary but not sufficient that their binary designations differ by a power of 2.

Example - 1 Reduce f(A, B, C) = A'B'C + A'BC + AB'C + ABC



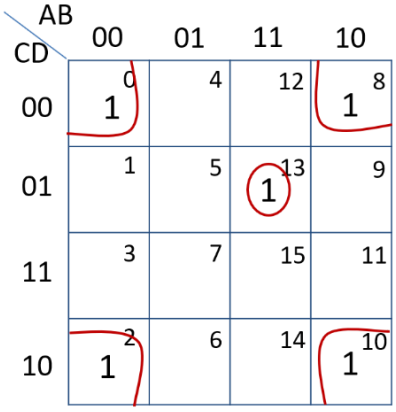
Answer: f = C

Example - 2 Reduce f (A, B, C, D) = ABC'D + ABCD + A'BC'D + A'BCD



Answer: f = BD

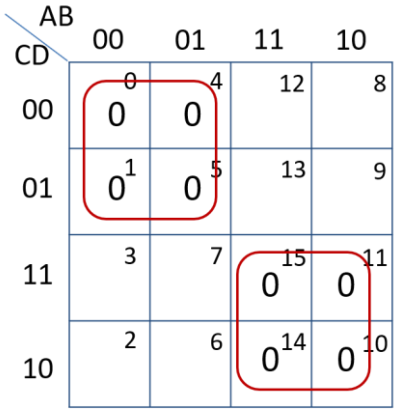
Example - 3 Reduce f(A, B, C, D) = $\sum_m(0, 2, 8, 10, 13)$



Answer: f = B'D' + ABC'D

Example – 4 Reduce f(A, B, C, D) = $\prod_M(0, 1, 4, 5, 10, 11, 14, 15)$

- In POS form, we have to put "0" in the given number boxes in K-Map.
- Make group of Os same as we make group of 1s in SOP form.
- For common "1" write complemented form and for common "0" write uncomplemented form of variable, e.g. 1 - A' and 0 - A.

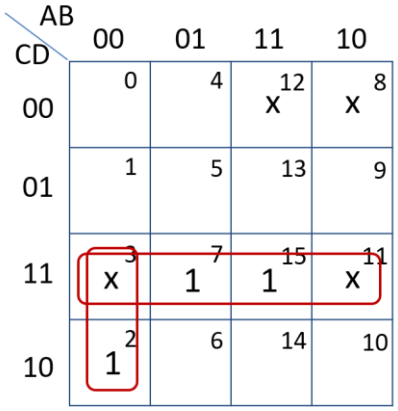


Answer: f = (A+C) (A'+C')

2.4.1. K-Map with don't care condition

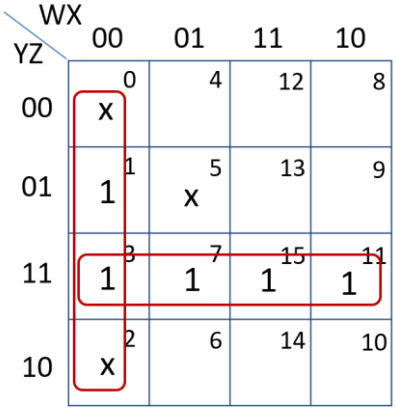
- Suppose we are given a problem of implementing a circuit to generate a logical 1 when a 2, 7, or 15 appears on a four-variable input.
- A logical 0 should be generated when 0, 1, 4, 5, 6, 9, 10, 13 or 14 appears.
- The input conditions for the numbers 3, 8, 11 and 12 never occur in the system. This means we don't care whether inputs generate logical 1 or logical 0.
- Don't care combinations are denoted by 'x' in K-Map which can be used for the making groups.
- The above example can be represented as

Example - 1 Reduce f(A, B, C, D) = $\sum_m(2, 7, 15) + d(3, 8, 11, 12)$



Answer: f = CD + A'B'C

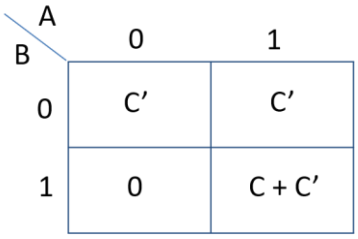
Example - 2 Reduce f(W, X, Y, Z) = $\sum_m(1,3,7,11,15) + d(0,2,5)$



Answer: f = W'X' + YZ

2.5. Variable-Entered Map (VEM)

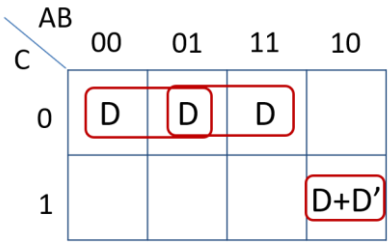
- Variable-entered map can be used to plot an n-variable problem on n - 1 variable map.
- Possible to reduce the map dimension by two or three in some cases.
- Advantage of using VEM occurs in design problems involving multiplexers.
- For example, Map-entered variable for 3 variable function
- Consider the function $X = A'B'C' + ABC' + AB'C' + ABC$
- Considering value of X to be function of map location and the variable C and plotting 2 variable K-Map.



Example - 1 Reduce f = AB'CD + A'BC'D + AB'CD' + ABC'D + A'B'C'D

= AB'CD + A'BC'D + AB'CD' + ABC'D + A'B'C'D

5 2 5 6 0

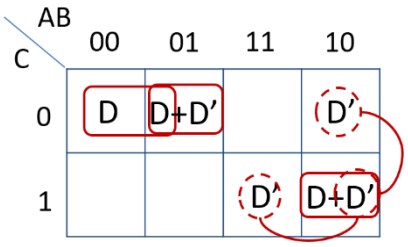


Answer: f = A'C'D + BC'D + AB'C

Example - 2 Reduce f = A'B'C'D + A'BC'D' + A'BC'D + AB'C'D' + AB'CD' + AB'CD + ABCD' using VEM method.

F = A'B'C'D + A'BC'D' + A'BC'D + AB'C'D' + AB'CD' + AB'CD + ABCD'

0 2 2 4 5 5 7



Answer: f = A'C'D + A'BC' + ACD' + AB'D' + AB'C

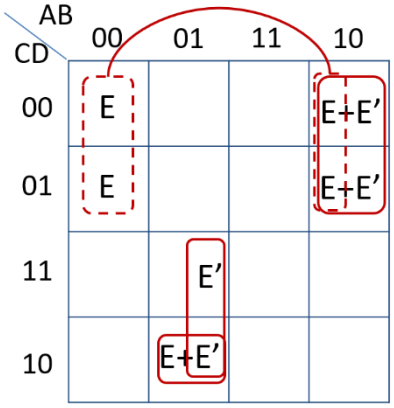
Example - 3 Reduce f = A'B'C'D'E + A'B'C'DE + A'BCD'E' + A'BCD'E + AB'C'D'E' + AB'C'D'E + AB'C'D + A'BCDE'

= A'B'C'D'E + A'B'C'DE + A'BCD'E' + A'BCD'E + AB'C'D'E' + AB'C'D'E +

0 1 6 6 8 8

AB'C'D + A'BCDE'

9 7



Answer: f = B'C'E + AB'C' + A'BCD' + A'BCE'

2.6. Quine McCluskey Method (Tabulation Method)

Procedure for minimization using tabulation method

1. List all the minterms.
2. Arrange all minterms in groups of the same number of 1s in their binary representation in column 1. Start with the least number of 1s group and continue with groups of increasing number of 1s.
3. Compare each term of the lowest index group with every term in the succeeding group. Whenever possible, combine the two terms being compared by means of the combining theorem. Two terms from adjacent groups are combinable, if their binary representations differ by just a single digit in the same position; the combined terms consist of the original fixed representation with the differing one replaced by a dash (-). Place a check mark (√) next to every term, which has been combined with at least one term and write the combined terms in column 2. Repeat this by comparing each term in a group of index i with every term in the group of index i + 1, until all possible applications of the combining theorem have been exhausted.
4. Compare the terms generated in step 3 in the same fashion; combine two terms which differ by only a single 1 and whose dashes are in the same position to generate a new term. Two terms with dashes in different positions cannot be combined. Write the new terms in

column 3 and put a check mark next to each term which has been combined in column 2. Continue the process with terms in columns 3, 4 etc. until no further combinations are possible. The remaining *unchecked terms* constitute the *set of prime implicants* of the expression.

5. List all the prime implicants and draw the *prime implicant chart*. (The don't cares if any should not appear in the prime implicant chart).
6. Obtain the *essential prime implicants* and write the minimal expression.

Example - 1 Simplify f(A, B, C, D) = $\sum_m(1, 2, 3, 5, 6, 7, 8, 9, 12, 13, 15)$ using tabulation method.

Step: -1 List all minterm

Minterms	Binary Designation
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
12	1 1 0 0
13	1 1 0 1
15	1 1 1 1

Step: -2 Arrange all minterms in groups of same number of 1s

Column-1		
Index	**Minterms**	**Binary Designation**
Index 1	1	0 0 0 1✓
	2	0 0 1 0✓
	8	1 0 0 0✓
Index 2	3	0 0 1 1✓
	5	0 1 0 1✓
	6	0 1 1 0✓
	9	1 0 0 1✓
	12	1 1 0 0✓
Index 3	7	0 1 1 1✓
	13	1 1 0 1✓
Index 4	15	1 1 1 1✓

Step: -3 Compare each term of the lowest index group with every term in the succeeding group till no change.

Column-2	
Pairs	**A B C D**
1,3	0 0 – 1 ✓
1,5	0 – 0 1 ✓
1,9	– 0 0 1 ✓
2,3	0 0 1 – ✓
2,6	0 – 1 0 ✓
8,9	1 0 0 – ✓
8,12	1 – 0 0 ✓
3,7	0 – 1 1 ✓
5,7	0 1 – 1 ✓
5,13	– 1 0 1 ✓
6,7	0 1 1 – ✓
9,13	1 – 0 1 ✓
12,13	1 1 0 – ✓
7,15	– 1 1 1 ✓
13,15	1 1 – 1 ✓

Step: -4 Compare the terms generated in step 3 in the same fashion until no further combinations are possible.

Column-3	
Quads	**A B C D**
1,3,5,7	0 – – 1 **T**
1,5,9,13	– – 0 1 **S**
2,3,6,7	0 – 1 – **R**
8,9,12,13	1 – 0 – **Q**
5,7,13,15	– 1 – 1 **P**

Step: -5 List all prime implicants and draw prime implicants chart.

Prime Implicants: P(BD), Q(AC'), R(A'C), S(C'D), T(A'D)

Minterms	**1**	**2** ✓	**3** ✓	**5** ✓	**6** ✓	**7** ✓	**8** ✓	**9** ✓	**12** ✓	**13** ✓	**15** ✓
T(A'D)	X		X	X		X					
S(C'D)	X			X				X		X	
R(A'C) ✓		X	X		X	X					
Q(AC') ✓							X	X	X	X	
P(BD) ✓				X		X				X	X

Step: -6 Obtain essential prime implicants and minimal expression.

Essential Prime Implicants: P(BD), Q(AC'), R(A'C)

Minimal Expression: P+Q+R+S = BD + A'C + AC' + C'D

P+Q+R+T = BD + A'C + AC' + A'D

As minterm 1 is covered by S and T.

Example - 2 Simplify f(A, B, C, D) = $\sum_m(0, 1, 3, 7, 8, 9, 11, 15)$ using tabulation method.

Step: -1 List all minterm

Minterms	**Binary Designation**
0	0 0 0 0
1	0 0 0 1
3	0 0 1 1
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1
11	1 0 1 1
15	1 1 1 1

Step: -2 Arrange all minterms in groups of same number of 1s

Column-1		
Index	**Minterms**	**Binary Designation**
Index 0	0	0 0 0 0 ✓
Index 1	1	0 0 0 1 ✓
	8	1 0 0 0 ✓
Index 2	3	0 0 1 1 ✓
	9	1 0 0 1 ✓
Index 3	7	0 1 1 1 ✓
	11	1 0 1 1 ✓
Index 4	15	1 1 1 1 ✓

Step: -3 Compare each term of the lowest index group with every term in the succeeding group till no change.

Column-2	
Pairs	**A B C D**
0,1	0 0 0 – ✓
0,8	– 0 0 0 ✓
1,3	0 0 – 1 ✓
1,9	– 0 0 1 ✓
8,9	1 0 0 – ✓
3,7	0 – 1 1 ✓
3,11	– 0 1 1✓
9,11	1 0 – 1 ✓
7,15	– 1 1 1 ✓
11,15	1 – 1 1 ✓

Step: -4 Compare the terms generated in step 3 in the same fashion until no further combinations are possible.

Column-3	
Quads	**A B C D**
0,1,8,9	– 0 0 – **R**
1,3,9,11	– 0 – 1 **Q**
3,7,11,15	– – 1 1 **P**

Step: -5 List all prime implicants and draw prime implicants chart.
Prime Implicants: P(CD), Q(B'D), R(B'C')

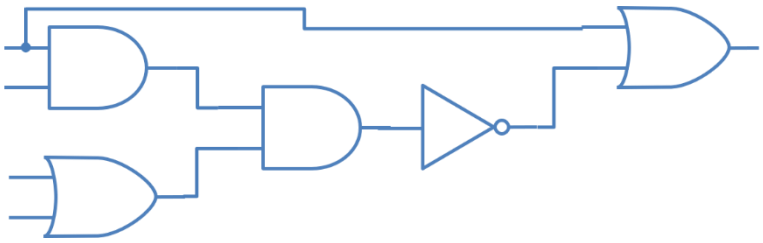
Minterms	**0 ✓**	**1 ✓**	**3 ✓**	**7 ✓**	**8 ✓**	**9 ✓**	**11 ✓**	**15 ✓**
P(CD)✓			X	X			X	X
Q(B'D)		X	X			X	X	
R(B'C')✓	X	X			X	X		

Step: -6 Obtain essential prime implicants and minimal expression.
Essential Prime Implicants: P(CD), R(B'C')
Minimal Expression: P+R = CD + B'C'

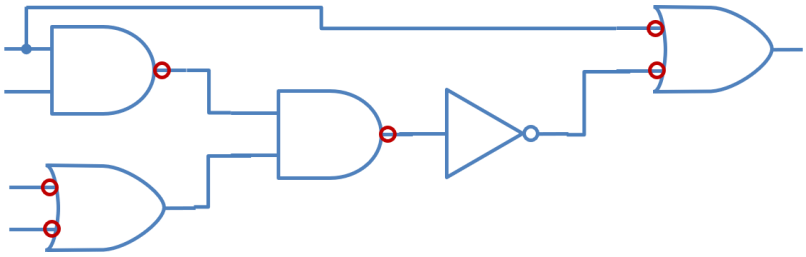
2.7. Realization using universal gates

1. Draw the circuit in AOI logic.
2. If NAND hardware is chosen, add a circle at the output of each AND gate and at the inputs to all the OR gates.
3. If NOR hardware is chosen, add a circle at the output of each OR gate and at the inputs to all the AND gates.
4. Add or subtract an inverter on each line that received a circle in steps 2 or 3 so that the polarity of signals on those lines remains unchanged from that of the original diagram.
5. Replace bubbled OR by NAND and bubbled AND by NOR.
6. Eliminate double inversions.

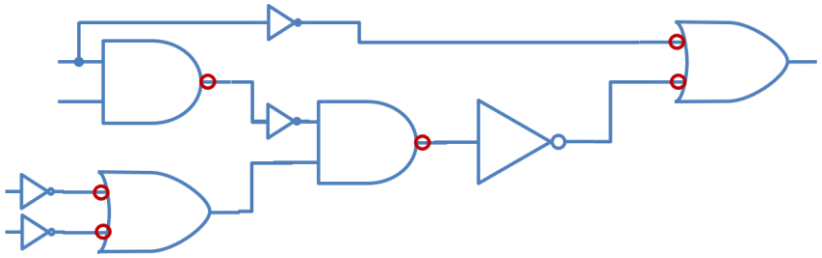
Example - 1 Implement the following AOI logic using NAND.



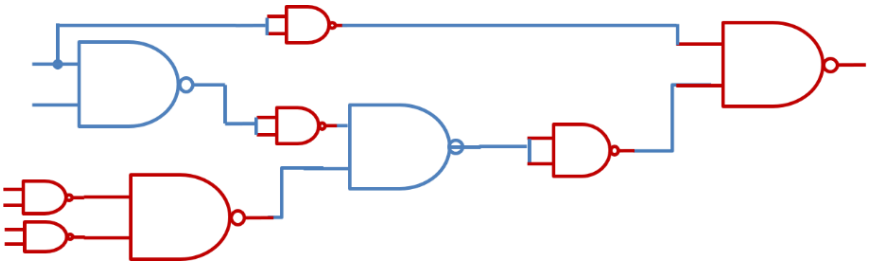
- Put a circle at the output of each AND gate and at the inputs to all OR gates



- Add an inverter to each of the lines that received only one circle at input so that polarity remains unchanged.



- Replace bubbled OR gates and NOT gates by NAND gates.



Unit-3
Combinational Digital Circuits

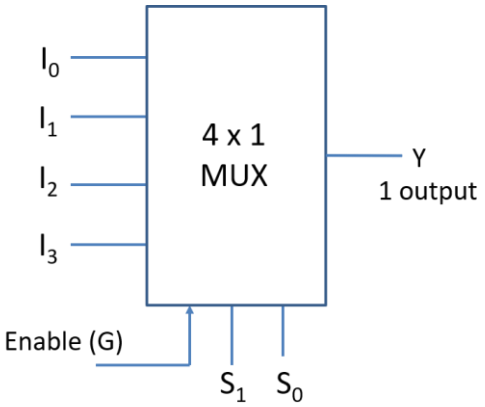
3.1. Multiplexer & De-Multiplexer

3.1.1. Multiplexer

- A multiplexer (MUX) is a device that allows digital information from several sources to be routed onto a single line for transmission over that line to a common destination.
- Consider an integer 'm', which is constrained by the following relation:

  $m = 2^n$, where m and n are both integers.
- A **m-to-1** Multiplexer has
  - m Inputs: $I_0, I_1, I_2, \ldots\ldots\ldots\ldots I_{(m-1)}$
  - One Output: Y
  - n Control inputs: $S_0, S_1, S_2, \ldots\ldots S_{(n-1)}$
  - One (or more) Enable input(s)

  such that Y may be equal to one of the inputs, depending upon the control inputs.
- The block diagram of 4 x 1 multiplexer is as follows.



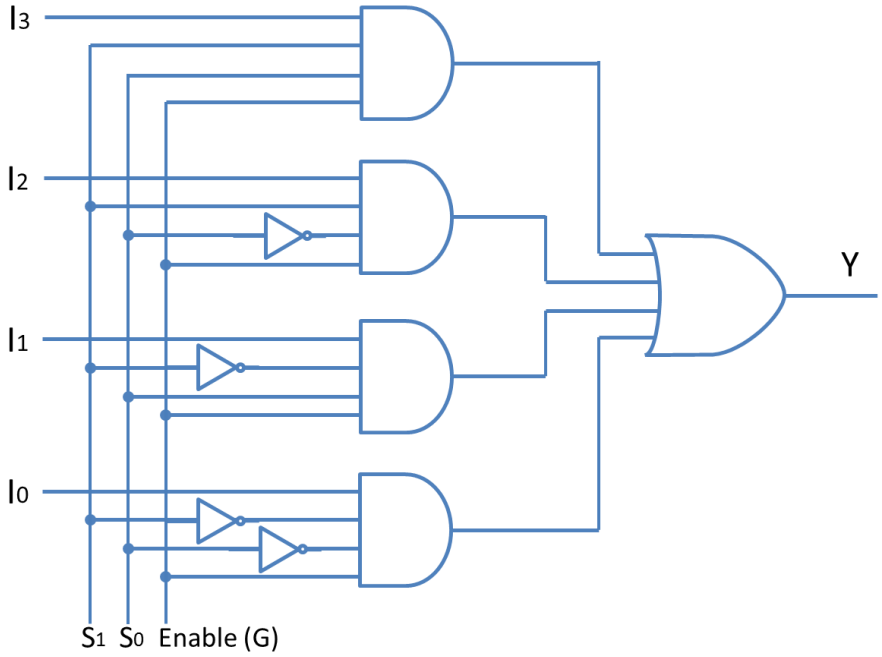
- The function table for the 4 x 1 multiplexer can be stated as below.

Select Inputs		Output
S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

- The following logic function describes the above function table.

$$Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$$

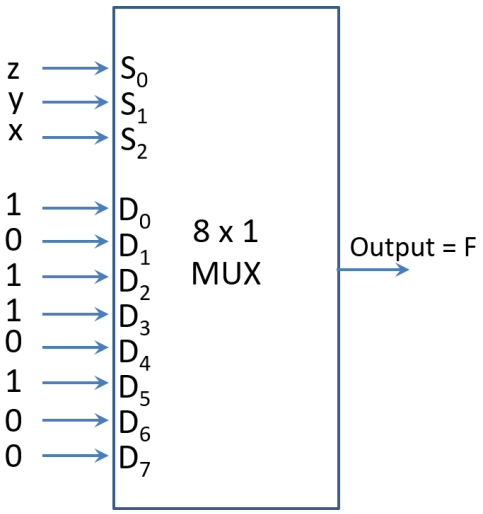
- The following figure describes the logic circuit for 4 x 1 multiplexer.



- Applications of Multiplexer is as follows:
  1. Logic function generation
  2. Data selection
  3. Data routing
  4. Operation sequencing
  5. Parallel-to-serial conversion
  6. Waveform generation

Example - 1 Implement the following function using 8 to 1 mux: f(X, Y, Z) = $\sum_m(0, 2, 3, 5)$

S_2	S_1	S_0	F
x	y	z	
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

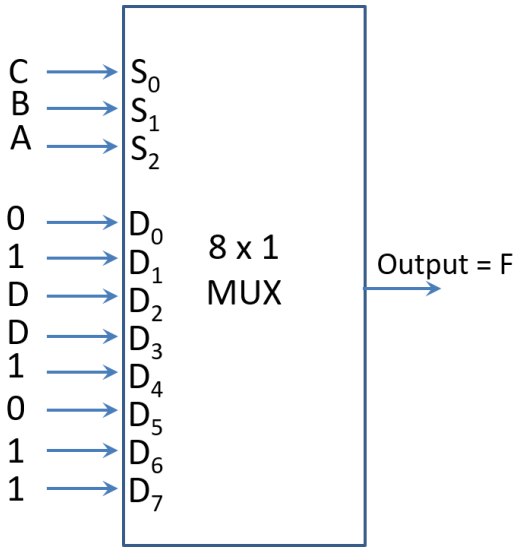


✓ **Logic Function Generator Using Multiplexer**

- Multiplexer with n-data select inputs can implement any function of n + 1 variables.
- The first n variables of the function as the select inputs and to use the least significant input variable and its complement to drive some of the data inputs.
- If the single variable is denoted by D, each data output of the multiplexer will be D, D', 1, or 0.
- Suppose, we wish to implement a 4-variable logic function using a multiplexer with three data select inputs.
- Let the input variables be A, B, C, and D; D is the LSB.
- A truth table for the function F(A, B, C, D) is constructed with ABC has the same value twice once with D = 0 and again with D = 1.
- The following rules are used to determine the connections that should be made to the data inputs of the multiplexer.
    1. If F = 0 both times when the same combination of ABC occurs, connect logic 0 to the data input selected by that combination.
    2. If F = 1 both times when the same combination of ABC occurs, connect logic 1 to the data input selected by that combination.
    3. If F is different for the two occurrences of a combination of ABC, and if F = D in each case, connect D to the data input selected by that combination.
    4. If F is different for the two occurrences of a combination of ABC, and if F = D' in each case, connect D' to the data input selected by that combination.

Example - 2 Implement the following using 8 to 1 mux: f(A, B, C, D) = $\sum_m(2,3,5,7,8,9,12,13,14,15)$

S_2	S_1	S_0	D	F	
A	B	C			
0	0	0	0	0	F = 0
0	0	0	1	0	
0	0	1	0	1	F = 1
0	0	1	1	1	
0	1	0	0	0	F = D
0	1	0	1	1	
0	1	1	0	0	F = D
0	1	1	1	1	
1	0	0	0	1	F = 1
1	0	0	1	1	
1	0	1	0	0	F = 0
1	0	1	1	0	
1	1	0	0	1	F = 1
1	1	0	1	1	
1	1	1	0	1	F = 1
1	1	1	1	1	



3.1.2. De-Multiplexer

- A multiplexer takes several inputs and transmits one of them to the output.
- A demultiplexer performs the reverse operation; it takes a single input and distributes it over several outputs.
- So, a demultiplexer can be thought of as a 'distributor', since it transmits the same data to different destinations.
- Thus, whereas a multiplexer is an N-to-1 device, demultiplexer is a 1-to-N device.
- Consider an integer 'm', which is constrained by the following relation:

$m = 2^n$, where m and n are both integers.

A **1-to-m** Demultiplexer has

One Input: D

m Outputs: O_0, O_1, O_2, ... $O_{(m-1)}$

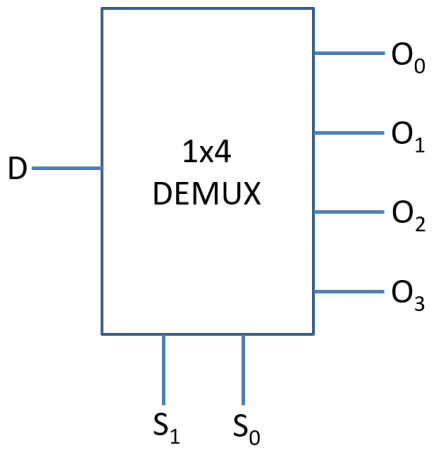
n Control inputs: S_0, S_1, S_2, ... $S_{(n-1)}$

One (or more) Enable input(s)

- Such that D may be transfer to one of the outputs, depending upon the control inputs.

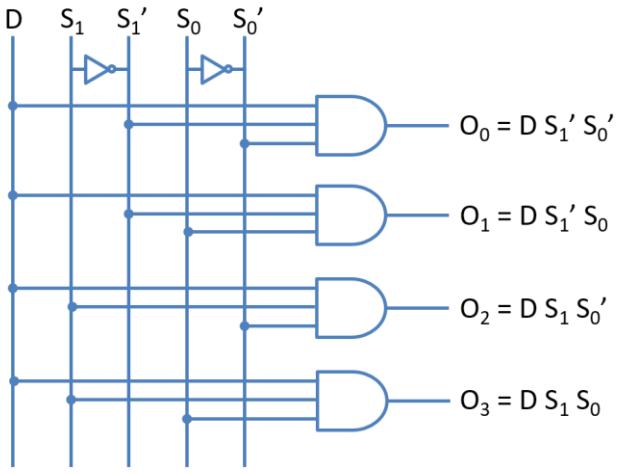
1x4 Demultiplexer

- The block diagram and truth table of 1 x 4 demultiplexer is as follows.



Select code		Outputs			
S_1	S_0	O_3	O_2	O_1	O_0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

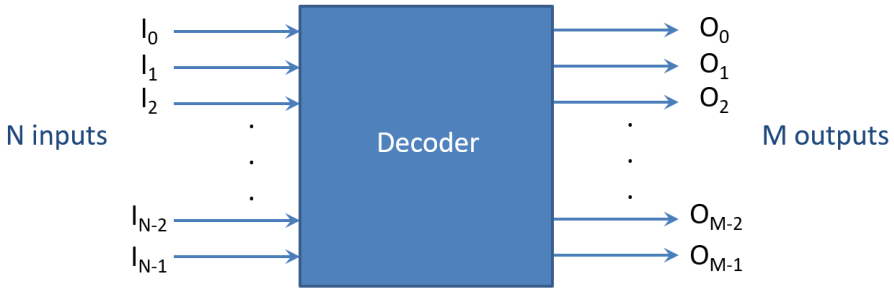
- The following figure describes the logic circuit for 1 x 4 demultiplexer.



3.2. Decoders, Encoders and Priority Encoders

3.2.1. Decoder

- A decoder is a logic circuit that converts an N-bit binary input code into M output lines such that only one output line is activated for each one of the possible combinations of inputs.
- In other words, we can say that a decoder identifies or recognizes or detects a particular code.



- Figure shows the general decoder diagram with N inputs and M outputs.
- Since each of the N inputs can be a 0 or a 1, there 2^N possible input combinations or codes.
- For each of these input combinations, only one of the M outputs will be active, all the other outputs will remain inactive.

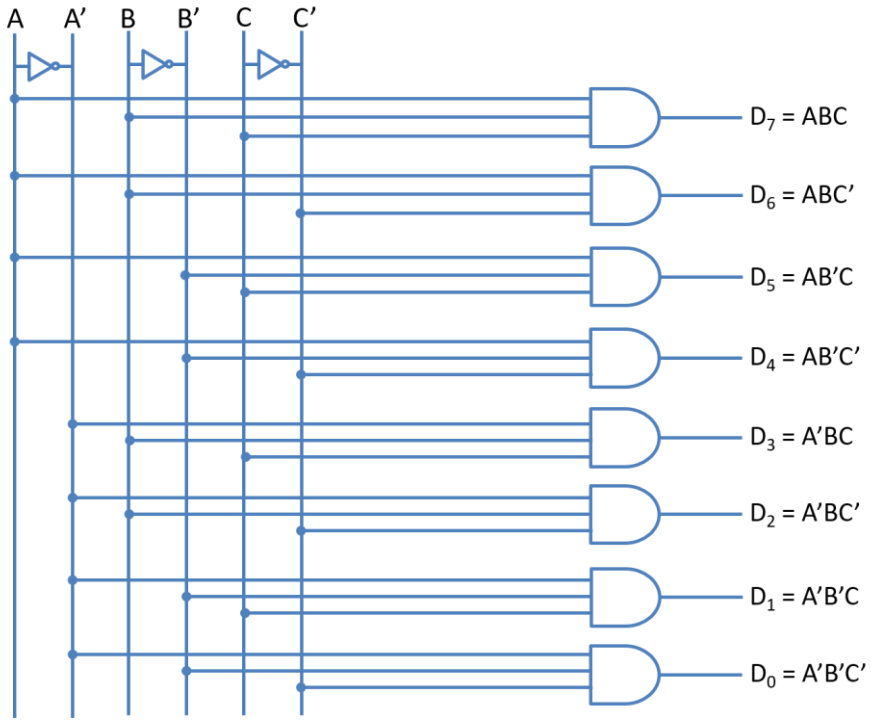
3 to 8 Decoder

- Figure shows the circuitry for a decoder with three inputs and eight outputs.
- It uses all AND gates, and therefore, the outputs are active-HIGH. For active-LOW outputs, NAND gates are used
- The truth table of the decoder is shown below.

Inputs			Outputs							
A	B	C	D_0 A'B'C'	D_1 A'B'C	D_2 A'BC'	D_3 A'BC	D_4 AB'C'	D_5 AB'C	D_6 ABC'	D_7 ABC
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

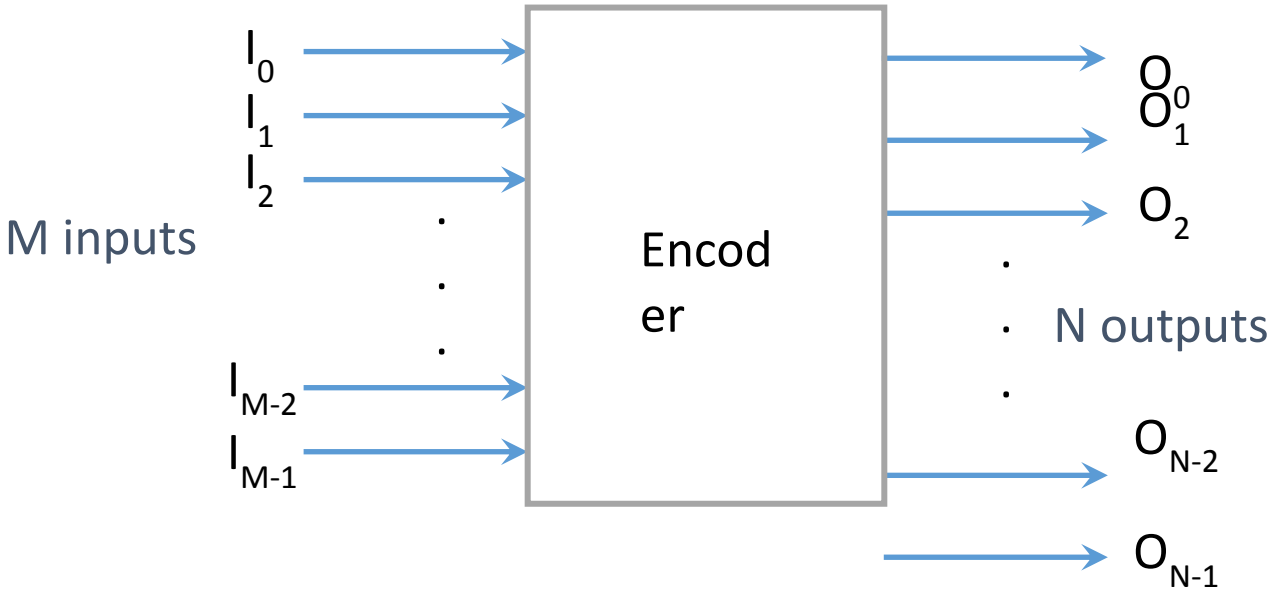
- This decoder can be referred to in several ways.
- It can be called a 3-line to 8-line decoder because it has three lines and eight output lines.

- It is also called a binary-to-octal decoder because it takes a 3-bit binary input code and activates one of the eight (octal) outputs corresponding to that code.
- It is also referred to as a 1-of-8 decoder because only one of the eight outputs is activated at one time.



3.2.1. Encoder

- Device to convert familiar numbers or symbols into coded format.
- It has a number of input lines, only one of which is activated at a given time, and produces an N-bit output code depending on which input is activated.
- Figure shows the block diagram of an encoder with M inputs and N outputs.



3.2.2. Priority Encoder

- A priority encoder is a logic circuit that responds to just one input in accordance with some priority system, among all those that may be simultaneously HIGH.
- The most common priority system is based on the relative magnitudes of the inputs; whichever decimal input is the largest, is the one that is encoded.
- The truth table of 4-input priority encoder and expression are given below:

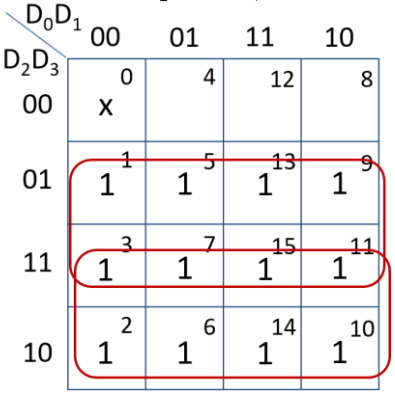
Inputs				Outputs		
D_0	D_1	D_2	D_3	A	B	V
0	0	0	0	x	x	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1

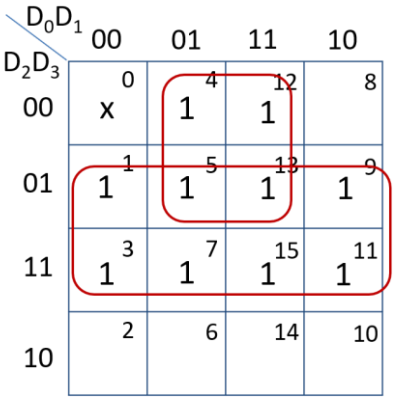
$$A = \sum_m(1,2,3,5,6,7,9,10,11,13,14,15)$$

$$B = \sum_m(1,3,4,5,7,9,11,12,13,15)$$

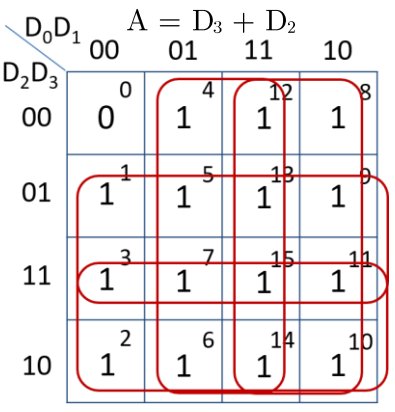
$$V = \sum_m(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)$$

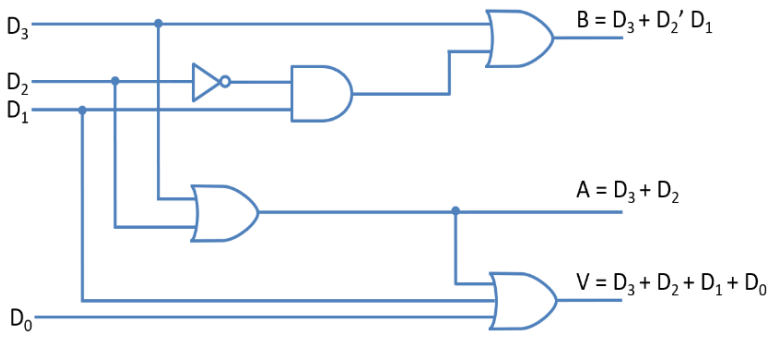
- In addition to the outputs A and B, the circuit has a third output designated by V.
- This is a valid bit indicator that is set to 1 when one or more inputs are equal to 1.
- If all inputs are 0, there is no valid input and V is equal to 0, the two other outputs are not inspected when V equals 0 and are specified as don't care conditions.
- According to the truth table, the higher the subscript number, the higher the priority of the input.
- The K-map for A, B and V with minimized expression are shown below:





$B = D_3 + D_2'D_1$





$V = D_3 + D_2 + D_1 + D_0$

3.3. Adders & Subtractors

3.3.1. Half Adder

- A half-adder is a combinational circuit with two binary inputs (augend and addend bits) and two binary outputs (sum and carry bits).
- It adds the two inputs (single bit words A and B) and produces the sum (S) and the carry (C) bits.
- The truth table of a half-adder are shown below:

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

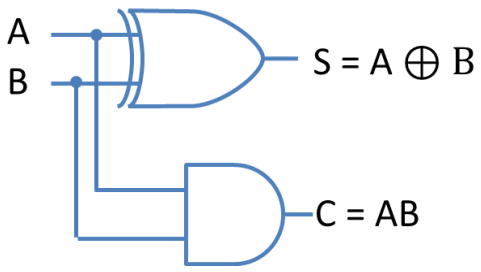
- The Sum (S) is the X-OR of A and B (It represent the LSB of the sum). Therefore,

$$S = AB' + BA' = A \oplus B$$

- The carry (C) is the AND of A and B (It is 0 unless both the inputs are 1). Therefore,

$$C = AB$$

- A half-adder can, therefore, be realized by using one X-OR gate and one AND gate as shown in figure below.



3.3.2. Full Adder

- A full-adder is a combinational circuit that adds two bits and a carry and outputs a sum bit and a carry bit.
- When we want to add two binary numbers, each having two or more bits, the LSBs can be added by using a half-adder.
- The carry resulted from the addition of the LSBs is carried over to the next significant column and added to the two bits in that column.
- The full-adder adds the bits A and B and the carry from the previous column called the carry-in C_{in} and outputs the sum bit S and the carry bit called the carry-out C_{out}.
- The variable S gives the value of the least significant bit of the sum.
- The variable C_{out} gives the output carry.

- The truth table of a full-adder are shown in figure below.

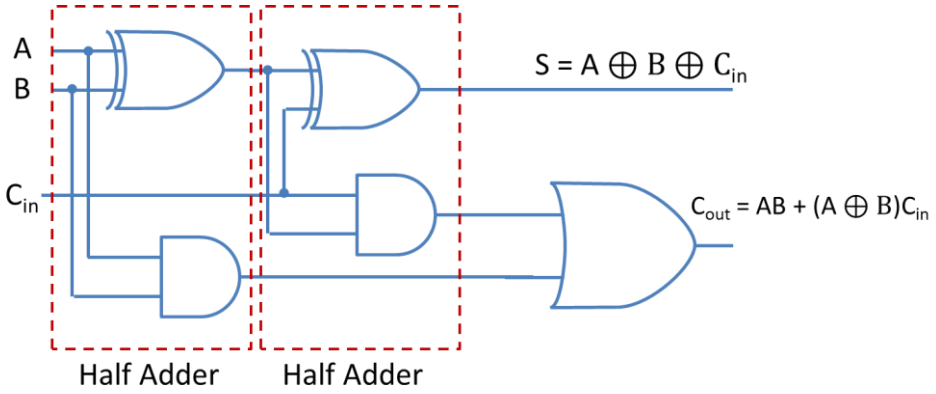
Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- The eight rows under the input variables designate all possible combinations of 1s and 0s that these variables may have.
- When all the bits are 0s, the output is 0.
- The S output is equal to 1 when only 1 input is equal to 1 or when all the inputs are equal to 1.
- The C_{out} has a carry of 1 if two or three inputs are equal to 1.
- From the truth table, a circuit that will produce the correct sum and carry bits in response to every possible combination of A, B, and C_{in} is described by

$$\begin{aligned} S &= A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in} \\ &= (AB' + A'B)C_{in}' + (AB + A'B')C_{in} \\ &= (A \oplus B)C_{in}' + (A \oplus B)'C_{in} \\ &= A \oplus B \oplus C_{in} \end{aligned}$$

$$\begin{aligned} C_{out} &= A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in} \\ &= AB + (A \oplus B)C_{in} \end{aligned}$$

- The sum term of the full-adder is the X-OR of A, B and C_{in}, i.e., the sum bit is the modulo sum of the data bits in that column and the carry from the previous column.
- The logic diagram of the full-adder using two X-OR gates and two AND gates (i.e., two half-adders) and one OR gate is shown in figure below.



3.3.3. Half Subtractor

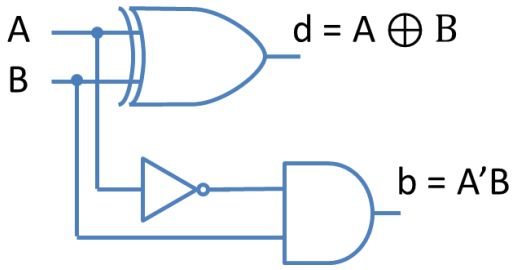
- A half-subtractor is a combinational circuit that subtracts one bit from the other and produces the difference.
- It also has an output to specify if a 1 has been borrowed.
- It is used to subtract the LSB of the subtrahend from the LSB of the minuend when one binary number is subtracted from the other.
- A half-subtractor is a combinational circuit with two inputs A and B and two outputs *d* and *b*.
- *d* indicates the difference and *b* is the output signal generated that informs the next stage that a 1 has been borrowed.
- We know that, when a bit B is subtracted from another bit A, a difference bit (d) and a borrow bit (b) result according to the rules given as follows.

Inputs		Outputs	
A	B	d	b
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

- A circuit that produces the correct difference and borrow bits in response to every possible combination of the two 1-bit numbers is, therefore, described by

$$d = AB' + BA' = A \oplus B \text{ and } d = A'B$$

- That is, the difference bit is obtained by X-ORing the two inputs, and the borrow bit is obtained by ANDing the complement of the minuend with the subtrahend.
- Figure below shows logic diagrams of a half-subtractor.



3.3.4. Full Subtractor

- The half-subtractor can be used only for LSB subtraction.
- If there is a borrow during the subtraction of the LSBs, it affects the subtraction in the next higher column; the subtrahend bit is subtracted from the minuend bit, considering the borrow from that column used for the subtraction in the preceding column.
- Such a subtraction is performed by a full-subtractor.
- It subtracts one bit (B) from another bit (A), when already there is a borrow b_i from this column for the subtraction in the preceding column, and outputs the difference bit (d) and the borrow bit (b) required from the next column.
- So a full-subtractor is a combinational circuit with three inputs (A, B, bi) and two outputs d and b.
- The 1s and 0s for the output variables are determined from the subtraction of A - B - b_i.
- The truth table of a full-subtractor are shown in figure.

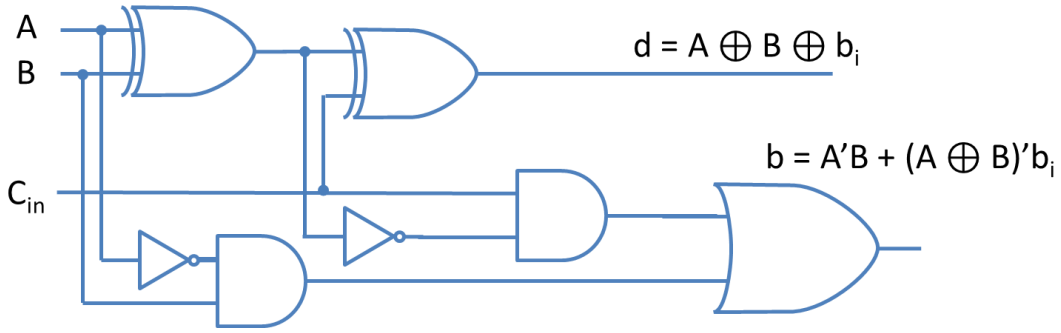
Inputs			Outputs	
A	**B**	**b_i**	**d**	**b**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

From the truth table, a circuit that will produce the correct difference and borrow bits in response to every possible combination of A, B, and b_i is described by

$$d = A'B'b_i + A'Bb_i' + AB'b_i' + ABb_i$$
$$= (AB' + A'B)b_i' + (AB + A'B')b_i$$
$$= (A \oplus B)b_i' + (A \oplus B)'b_i$$
$$= A \oplus B \oplus b_i$$

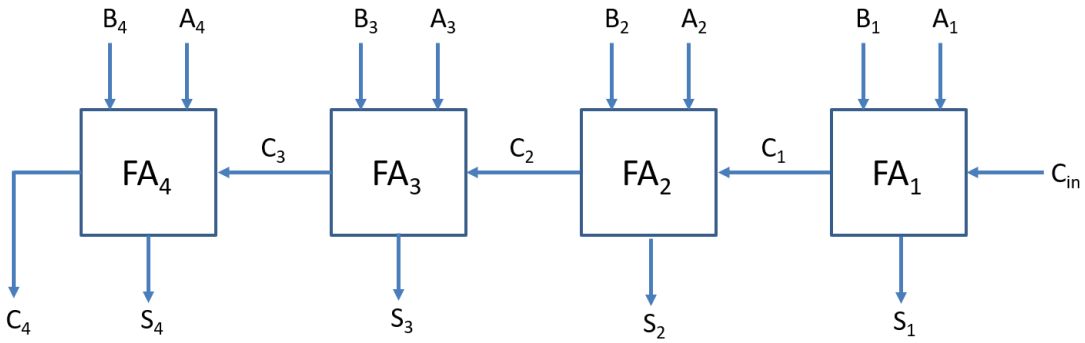
$$b = A'B'b_i + A'Bb_i' + A'Bb_i + ABb_i$$
$$= A'B(b_i + b_i') + (AB + A'B')b_i$$
$$= A'B + (A \oplus B)'b_i$$

- A full-subtractor can, therefore, be realized using X-OR gates as shown below.



3.3.5. Binary Parallel Adder

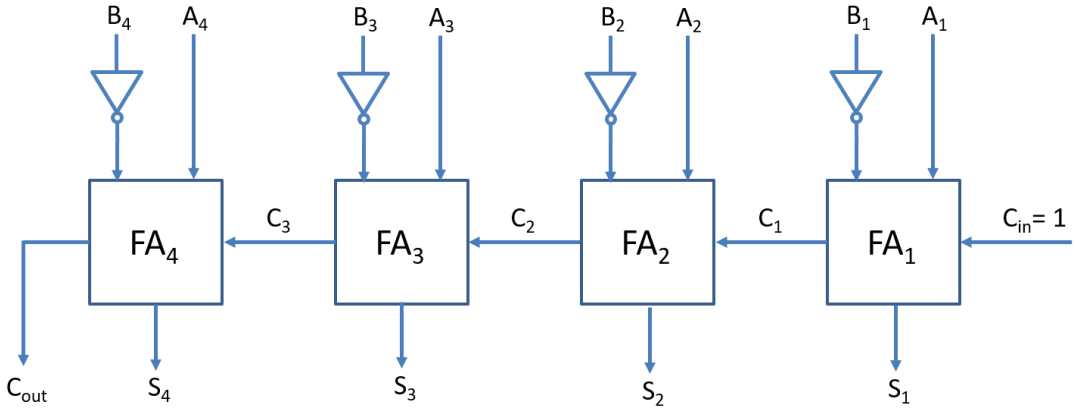
- A binary parallel adder is a digital circuit that adds two binary numbers in parallel form and produces the arithmetic sum of those numbers in parallel form.
- It consists of full adders connected in a chain with the output carry from each full-adder connected to the input carry of the next full-adder in the chain.



- Figure shows the interconnection of four full-adder (FA) circuits to provide a 4-bit parallel adder.
- The augend bits of A and addend bits of B are designated by subscript numbers from right to left, with subscript 1 denoting the lower-order bit.
- The carries are connected in 3 chain through the full-adders.
- The input carry to the adder is C_{in} and the output carry is C_4.
- The S outputs generate the required sum bits.
- When the 4-bit full-adder circuit is enclosed within an IC package, it has four terminals for the augend bits, four terminals for the addend bits, four terminals for the sum bits, and two terminals for the input and output carries.
- An n-bit parallel adder requires n-full adders.
- It can be constructed from 4-bit, 2-bit, and 1-bit full adder ICs by cascading several packages.
- The output carry from one package must be connected to the input carry of the one with the next higher-order bits.
- The 4-bit full adder is a typical example of an MSI function.

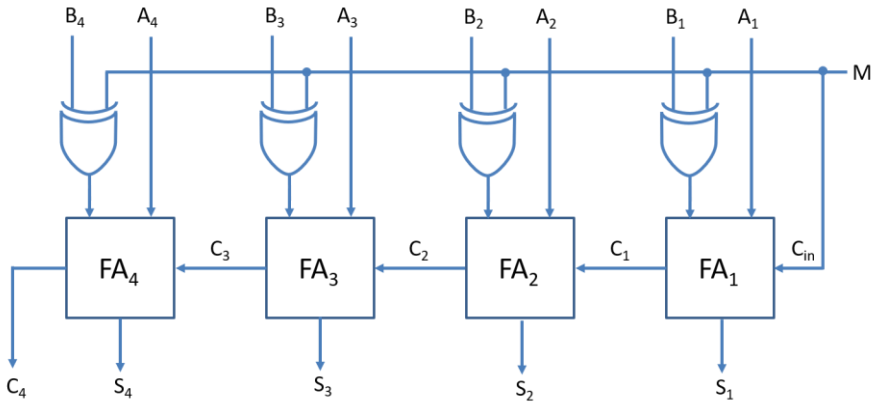
3.3.6. Binary Parallel Subtractor

- The subtraction of binary numbers can be carried out most conveniently by means of complement.
- Remember that the subtraction A – B can be done by taking the 2's complement of B and adding it to A.
- The 2's complement can be obtained by taking the 1's complement and adding 1 to the least significant pair of bits.
- The 1's complement can be implemented with inverters as shown in below figure.



3.3.7. Binary Adder Subtractor

- Figure shows a 4-bit adder-subtractor circuit.
- Here the addition and subtraction operations are combined into one circuit with one common binary adder.
- This is done by including an X-OR gate with each full-adder.
- The mode input M controls the operation.
- When M = 0, the circuit is an adder, and when M = 1, the circuit becomes a subtractor.
- Each X-OR gate receives input M and one of the inputs of B.
- When M = 0, we have B ⊕ 0 = B. The full-adder receives the value of B, the input carry is 0 and the circuit performs A + B.
- When M = 1, we have B ⊕ 1 = B' and C_1 = 1. The B inputs are complemented and a 1 is added through the input carry.
- The circuit performs the operation A + B' + 1 (i.e. A – B).



3.4. Digital Comparator

- A comparator is a logic circuit used to compare the magnitudes of two binary numbers.
- Comparator circuit provides 3 outputs

  $A = B$

  $A > B$

  $A < B$
- X-NOR gate is a basic comparator (the output is 1 if and only if the input bits coincide).
- 2 binary numbers are equal if and only if all their corresponding bits coincide.
- E.g. $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are equal if and only if $A_3=B_3$, $A_2=B_2$, $A_1=B_1$ and $A_0=B_0$

$$\text{Equality} = (A_3 \odot B_3)\ (A_2 \odot B_2)\ (A_1 \odot B_1)\ (A_0 \odot B_0)$$

3.4.1. 1 - Bit Magnitude Comparator

- Let the 1-bit numbers be $A = A_0$ and $B = B_0$
- If $A_0 = 1$ and $B_0 = 0$ then $A > B$

$$A > B : G = A_0B_0'$$

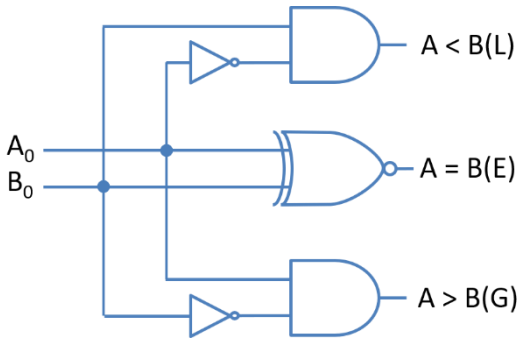
- If $A_0 = 0$ and $B_0 = 1$ then $A < B$

$$A < B : L = A_0'B_0$$

- If $A_0 = 1$ and $B_0 = 1$ (coincides) then $A = B$

$$A = B : E = A_0 \odot B_0$$

A_0	B_0	L	E	G
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0



3.4.2. 2- bit Magnitude Comparator

- The logic for a 2-bit magnitude comparator: Let the two 2-bit numbers be $A = A_1A_0$ and $B = B_1B_0$.
  1. If $A_1 = 1$ and $B_1 = 0$, then $A > B$ or
  2. If A_1 and B_1 coincide and $A_0 = 1$ and $B_0 = 0$, then $A > B$. So the logic expression for $A > B$ is

$$A > B : G = A_1B_1' + (A1 \odot B1)\ A_0B_0'$$

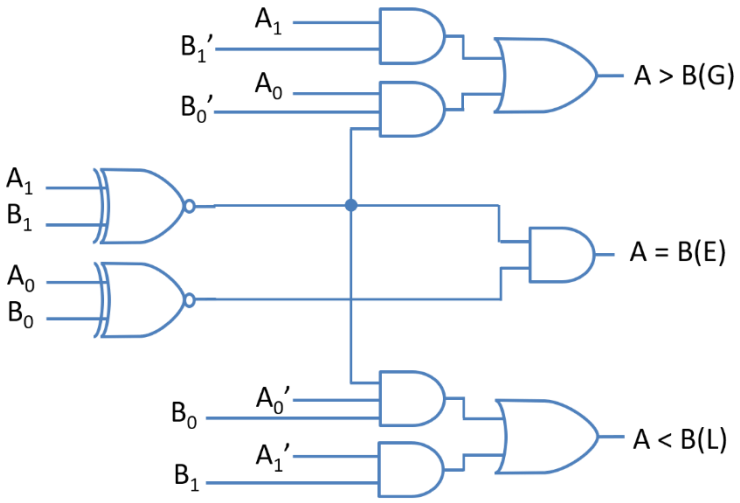
  1. If $A_1 = 0$ and $B_1 = 1$, then $A < B$ or
  2. If A_1 and B_1 coincide and $A_0 = 0$ and $B_0 = 1$, then $A < B$. So the expression for $A < B$ is

$$A < B : L = A_1'B_1 + (A_1 \odot B_1)\ A_0'B_0$$

  If A_1 and B_1 coincide and if A_0 and B_0 coincide then $A = B$. So the expression for $A = B$ is

$$A = B : E = (A_1 \odot B_1)(A_0 \odot B_0)$$

- The logic diagram for a 2-bit comparator is as shown below:



3.5. Parity Generator

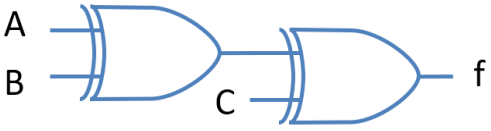
- Exclusive-OR functions are very useful in systems requiring error detection and correction codes.
- Binary data, when transmitted and processed, is susceptible to noise that can alter its 1s to Os and 0s to 1s.
- To detect such errors, an additional bit called the *parity bit* is added to the data bits and the word containing the data bits and the parity bit is transmitted.
- At the receiving end the number of 1s in the word received are counted and the error, if any, is detected.
- This parity checks, however, detects only single bit errors.
- The circuit that generates the parity bit in the transmitter is called a parity generator.
- The circuit that checks the parity in the receiver is called a parity checker.
- A parity bit, a 0 or a 1 is attached to the data bits such that the total number of 1s in the word is even for even parity and odd for odd parity.
- The parity bit can be attached to the code group either at the beginning or at the end depending on system design.
- A given system operates with either even or odd parity but not both.
- So, a word always contains either an even or an odd number of 1s.
- At the receiving end, if the word received has an even number of 1s in the odd parity system or an odd number of 1s in the even parity system, it implies that an error has occurred.
- In order to check or generate the proper parity bit in a given code word, the basic principle used is, “the modulo sum of an even number of 1s is always a 0 and the modulo sum of an odd number of 1s is always a 1".

- Therefore, in order to check for an error, all the bits in the received word are added.
- If the modulo sum is a 0 for an odd parity system or a 1 for an even parity system, an error is detected.

Example - 1 Design 3-bit parity generator using even parity bit.

Inputs			Outputs parity bit (f)
A	B	C	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$f = A'B'C + A'BC' + ABC + AB'C'$$
$$f = A'(B'C + BC') + A(BC + B'C')$$
$$f = A'(B \oplus C) + A(B \oplus C)'$$
$$f = A \oplus B \oplus C$$



3.6. Code Converters

3.6.1. Binary to Gray Code Converter

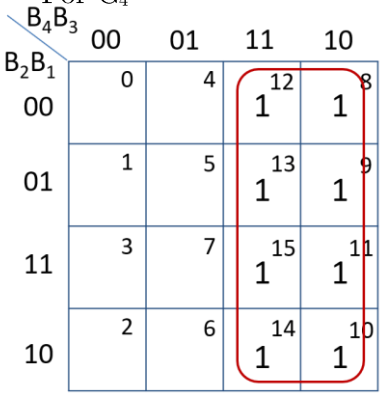
- The input to the 4-bit binary to gray code converter circuit is a 4-bit binary and the output is a 4-bit gray code.
- There are 16 possible combinations of 4-bit binary input and all of them are valid.
- The 4-bit binary and the corresponding gray code are shown in below table.
- From the conversion table, we observe that the expressions for the outputs G_4, G_3, G_2, and G_1 are as follows:

$$G_4 = \sum_m(8, 9, 10, 11, 12, 13, 14, 15)$$
$$G_3 = \sum_m(4, 5, 6, 7, 8, 9, 10, 11)$$
$$G_2 = \sum_m(2, 3, 4, 5, 10, 11, 12, 13)$$
$$G_1 = \sum_m(1, 2, 5, 6, 9, 10, 13, 14)$$

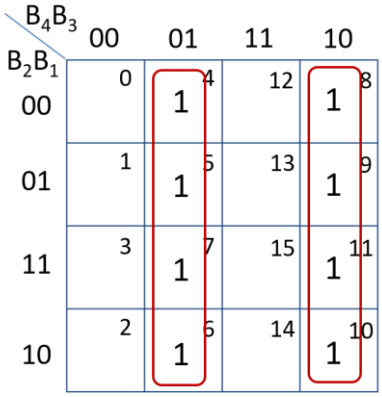
4-bit Binary				4-bit Gray			
B_4	B_3	B_2	B_1	G_4	G_3	G_2	G_1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

- The K-maps for G_4, G_3, G_2, and G_1 and their minimization are shown below:

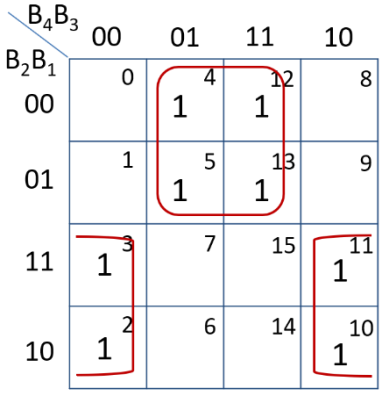
For G_4



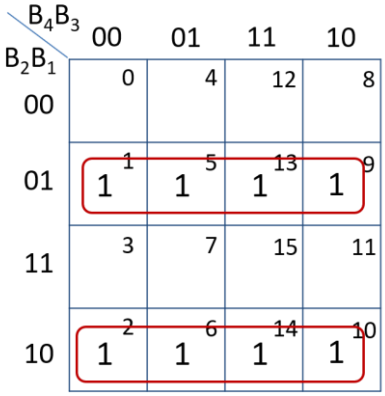
For G_3



ForG_2



For G_1



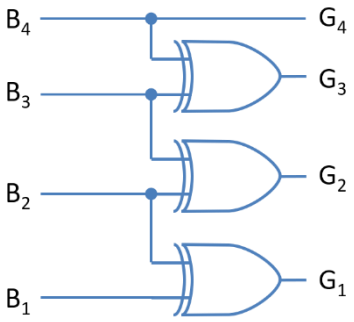
- The minimal expressions for the outputs obtained from the K-map are:

$G_4 = B_4$

$G_3 = B_4' B_3 + B_4 B_3' = B_4 \oplus B_3$

$G_2 = B_3' B_2 + B_3 B_2' = B_3 \oplus B_2$

$G_1 = B_2' B_1 + B_2 B_1' = B_2 \oplus B_1$



3.6.2. BCD to XS-3 Code Converter

- BCD means 8421 BCD.
- The 4-bit input BCD code (B4 B3 B2 B1) and the corresponding output XS-3 code (X4 X3 X2 X1) numbers are shown in the conversion table in figure.

8421 code				XS-3 code			
B_4	B_3	B_2	B_1	X_4	X_3	X_2	X_1
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

- The input combinations 1010, 1011, 1100, 1101, 1110, and 1111 are invalid in BCD. So they are treated as don't cares.
- From the above truth table, function can be realized as follows:

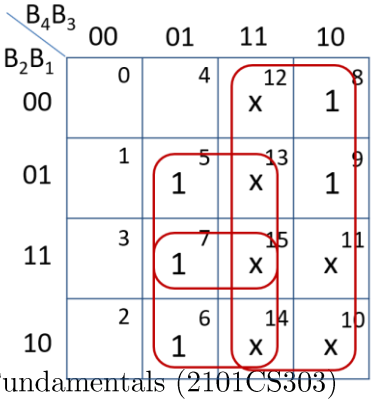
$X_4 = \mathbf{\Sigma}\, m(5, 6, 7, 8, 9) + d(10, 11, 12, 13, 14, 15)$

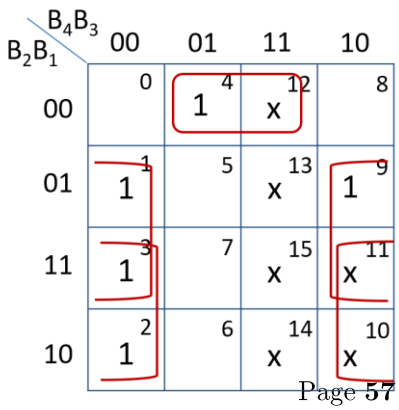
$X_3 = \mathbf{\Sigma}\, m(1, 2, 3, 4, 9) + d(10, 11, 12, 13, 14, 15)$

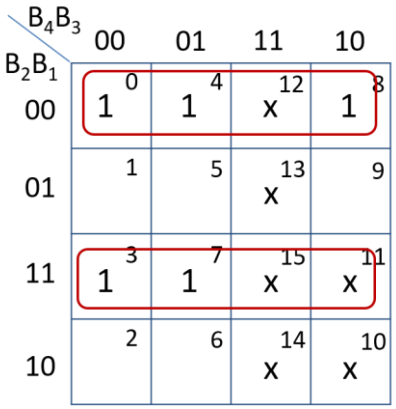
$X_2 = \mathbf{\Sigma}\, m(0, 3, 4, 7, 8) + d(10, 11, 12, 13, 14, 15)$

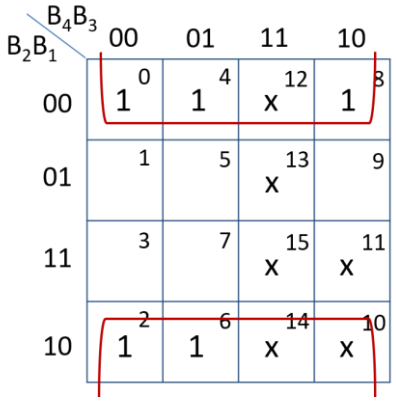
$X_1 = \mathbf{\Sigma}\, m(0, 2, 4, 6, 8) + d(10, 11, 12, 13, 14, 15)$

- Drawing K-maps for the outputs X_4, X_3, X_2, and X_1 in terms of the inputs B_4, B_3, B_2, and B_1 and simplifying them, as shown.







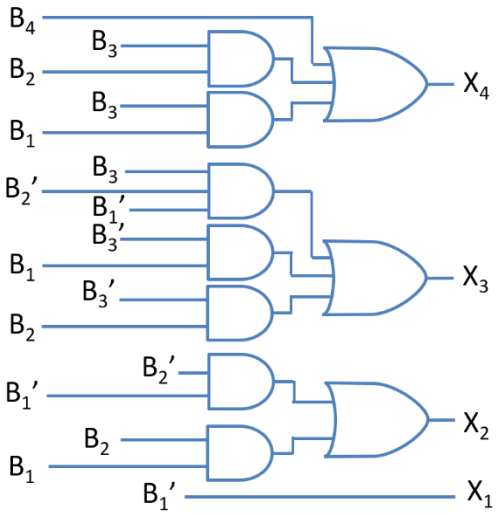


- The minimal expressions are

  $X_4 = B_4 + B_3B_2 + B_3B_1$

  $X_3 = B_3B_2'B_1' + B_3'B_1 + B_3'B_2$

  $X_2 = B_2'B_1' + B_2B_1$



Unit-4
Sequential Digital Circuits

4.1. SR Latch & Flip-flop: SR, D, JK, T

4.1.1. Sequential switching Circuits

- Sequential switching circuits are circuits whose output levels at any instant of time are dependent on the levels present at the inputs at that time and on the state of the circuit, i.e., on the prior input level conditions (i.e. on its past inputs)
- The past history is provided by feedback from the output back to the input.
- Made up of combinational circuits and memory elements, e.g. Counters, shift registers, serial adder, etc.

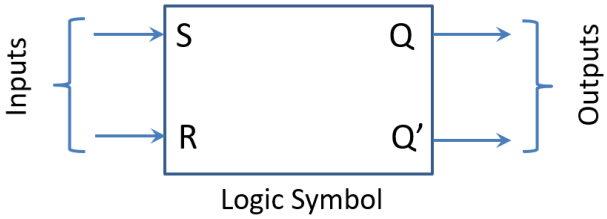
Combinational circuits	Sequential circuits
1. In combinational circuits, the output variables at any instant of time are dependent only on the present input variables.	1. In sequential circuits, the output variables at any instant of time are dependent not only on the present input variables, but also on the present state, i.e. on the past history of the system.
2. Memory unit is not required in combinational circuits.	2. Memory unit is required to store the past history of the input variables in sequential circuits.
3. Combinational circuits are faster because the delay between the input and the output is due to propagation delay of gates only.	3. Sequential circuits are slower than combinational circuits.
4. Combinational circuits are easy to design.	4. Sequential circuits are comparatively harder to design.

4.1.2. Latch

- A flip-flop, known formally as bistable multivibrator, has two stable states.
- It can remain in either of the states indefinitely.
- Its state can be changed by applying the proper triggering signal.
- Latch is used for certain flip-flop which are *non-clocked.*These flip-flops 'latch on' to a 1 or a 0 immediately upon receiving the input pulse called SET or RESET.
- The latch is sequential device that checks all its inputs continuously and changes its outputs accordingly at any time independent of a clock signal.

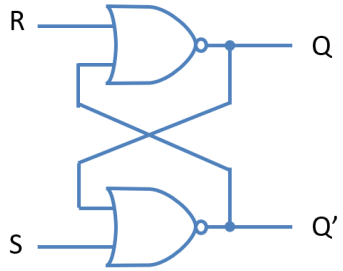
4.1.2.1. S-R Latch

- The simplest type of flip-flop is called an S-R latch.
- It has two outputs labelled Q and Q' and two inputs labelled S and R. The state of the latch corresponds to the level of Q (HIGH or LOW, 1 or 0) and Q' is the complement of that state.
- It can be constructed using either two cross-coupled NAND gates or two-cross coupled NOR gates.
- Using two NOR gates, an active-HIGH S-R latch can be constructed and using two NAND gates an active-LOW S-R latch can be constructed.
- The name of the latch, S-R or SET-RESET, is derived from the names of its inputs.
- Below is the common logic symbol for both latches.



Logic Symbol

NOR gate S-R latch (Active HIGH)

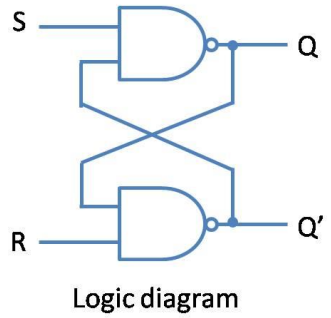


Logic diagram

S	R	Q_n	Q_{n+1}	State
0	0	0	0	No Change (NC)
0	0	1	1	
0	1	0	0	Reset
0	1	1	0	
1	0	0	1	Set
1	0	1	1	
1	1	0	X	Indeterminate (Invalid)
1	1	1	X	

- When the SET input is made HIGH, Q becomes 1.
- When the RESET input is made HIGH, Q becomes 0.
- If both the inputs S and R made LOW, there is no change in the state of the latch.
- If both the inputs are made HIGH, the output is unpredictable i.e. invalid.
- Figure shows the logic diagram of an active HIGH S-R latch composed of two cross-coupled NOR gates.
- Note that the output of each gate is connected to one of the inputs of the other gate.
- The latch works as per the truth table of figure, where Q_n represents the state of the flip-flop before applying inputs and Q_{n+1} represents the state of the flip-flop after applying inputs.

NAND gate S-R latch (Active LOW)

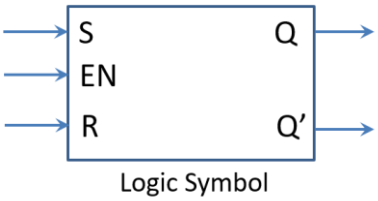


Logic diagram

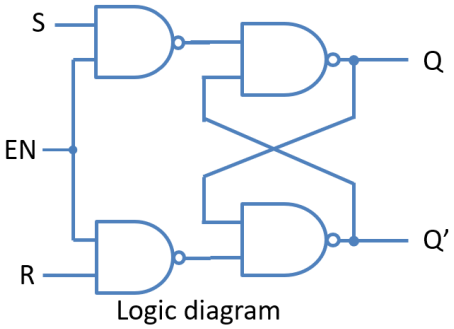
S	R	Q_n	Q_{n+1}	State
0	0	0	X	Indeterminate (Invalid)
0	0	1	X	
0	1	0	1	Set
0	1	1	1	
1	0	0	0	Reset
1	0	1	0	
1	1	0	0	No Change (NC)
1	1	1	1	

- The NAND gate is equivalent to an active LOW OR gate, am active LOW S-R latch using OR gates may also be represented.
- The operation of this latch is the reverse of the operation of the NOR gate latch.
- If the 0s are replaced by 1s and 1s by 0s, we get the same truth table as that of NOR gate latch.
- The SET and RESET inputs are normally resting in the HIGH state and one of them will be pulsed LOW, whenever we want to change the latch outputs.

4.1.2.2. S-R Flip-flop (Gated S-R latch)



Logic Symbol

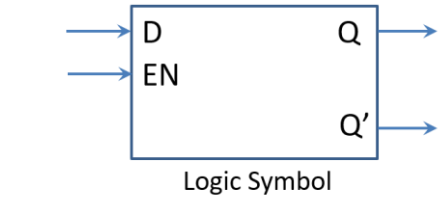


Logic diagram

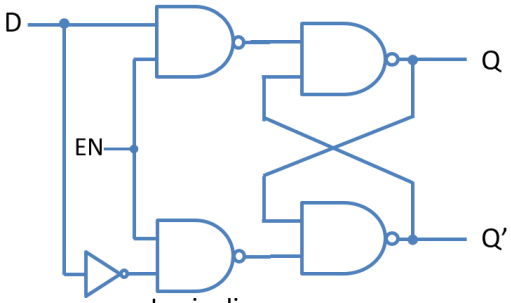
En	S	R	Q_n	Q_{n+1}	State
1	0	0	0	0	No Change (NC)
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	
1	1	0	0	1	Set
1	1	0	1	1	
1	1	1	0	X	Indeterminate (Invalid)
1	1	1	1	X	
0	X	X	0	0	No Change (NC)
0	X	X	1	1	

- A gated S-R larch requires an ENABLE (EN) input.
- Its S and R inputs will control the state of the flip-flop only when the EN is HIGH.
- When EN is LOW, the inputs become ineffective and no change of state can take place.
- The EN input may be a clock. So, a gated S-R latch is also called a *clocked S-R latch.*
- Since this type of flip-flop responds to the changes in inputs only as long as the clock is HIGH, these types of flip-flops are also called *level triggered flip-flops.*

4.1.2.3. D Flip-flop (Gated D latch)



Logic Symbol

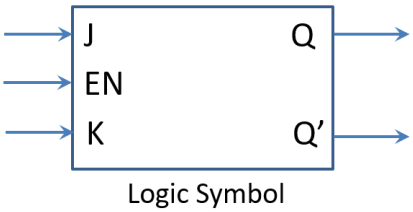


Logic diagram

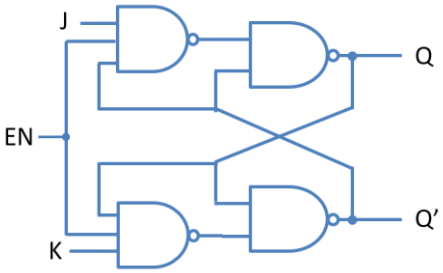
En	D	Q_n	Q_{n+1}	State
1	0	0	0	Reset
1	0	1	0	
1	1	0	1	Set
1	1	1	1	
0	X	0	0	No Change (NC)
0	X	1	1	

- It differs from the S-R latch in that it has only one input in addition to EN.
- When D=1, we have S=1 and R=0, causing the latch to SET when ENABLED.
- When D=0, we have S=0 and R=1, causing the latch to RESET when ENABLED.
- When EN is LOW, the latch is ineffective, and any change in the value of D input does not affect the output at all.
- When EN is HIGH, a LOW D input makes Q LOW, i.e. resets the flip-flop and a HIGH D input makes Q HIGH, i.e. sets the flip-flop.
- In other words, we can say that the output Q follows the D input when EN is HIGH.

4.1.2.4. J-K Flip-flop (Gated J-K latch)



Logic Symbol

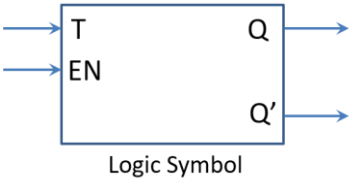


En	J	K	Q_n	Q_{n+1}	State
1	0	0	0	0	No Change (NC)
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	
1	1	0	0	1	Set
1	1	0	1	1	
1	1	1	0	1	Toggle
1	1	1	1	0	
0	X	X	0	0	No Change (NC)
0	X	X	1	1	

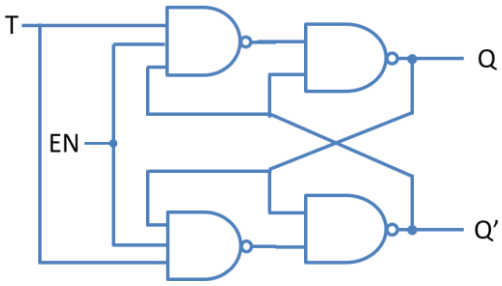
- The J-K flip-flop is very versatile and also the most widely used.
- The functioning of the J-K flip-flop is identical to that of the S-R flip-flop, except that it has no invalid state like that of S-R flip-flop.
- When J=0 and K=0, no change of state place even if a clock pulse is applied.

- When J=0 and K=1, the flip-flop resets at the HIGH level of the clock pulse.
- When J=1 and K=0, the flip-flop sets at the HIGH level of the clock pulse.
- When J=1 and K=1, the flip-flop toggles, i.e. goes to the opposite state at HIGH level of clock pulse.
- We can make edge triggered flip-flop as well by using positive and negative edges of clock instead of HIGH and LOW level.

4.1.2.5. T Flip-flop



Logic Symbol



En	T	Q_n	Q_{n+1}	State
1	0	0	0	No Change (NC)
1	0	1	1	
1	1	0	1	Toggle
1	1	1	0	
0	X	0	0	No Change (NC)
0	X	1	1	

- A T flip-flop has a single control input, labeled T for toggle.
- When T is HIGH, the flip-flop toggles on every new clock pulse.
- When T is LOW, the flip-flop remains in whatever state it was before.
- Although T flip-flops are not widely available commercially, it is easy to convert a J-K flip-flop to the functional equivalent of a T flip-flop by just connecting J and K together and labeling the common connection as T.
- Thus, when T = 1, we have J = K = 1, and the flip-flop toggles.
- When T = 0, we have J =K = 0, and so there is no change of state.

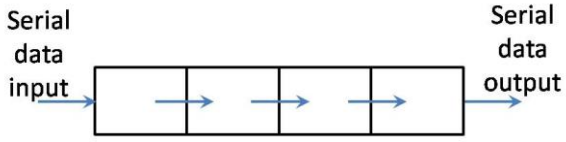
4.2. Register

- As a flip-flop (FF) can store only one bit of data, a 0 or a 1, it is referred to as a single-bit register.
- A register is a set of FFs used to store binary data.
- The storage capacity of a register is the number of bits (1s and 0s) of digital data it can retain.
- Loading a register means setting or resetting the individual FFs, i.e. inputting data into the register so that their states correspond to the bits of data to be stored
- Loading may be serial or parallel.
- In serial loading, data is transferred into the register in serial form i.e. one bit at a time.
- In parallel loading, the data is transferred into the register in parallel form meaning that all the FFs are triggered into their new states at the same time.

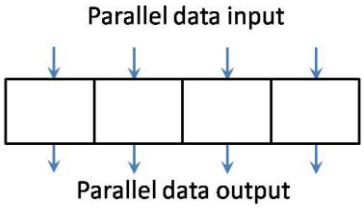
- Types of Registers
  1. Buffer register
  2. Shift register
  3. Bidirectional shift register
  4. Universal shift register

4.2.1. Shift Register

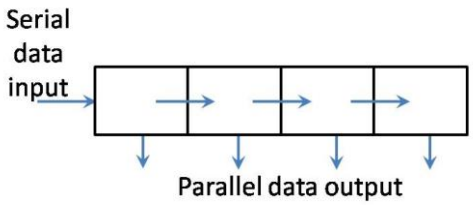
- A number of FFs connected together such that data may be shifted into and shifted out of them is called a shift register.
- Data may be shifted into or out of the register either in serial form or in parallel form.
- So, there are four basic types of shift registers:
  1. serial-in, serial-out
  2. serial-in, parallel out
  3. parallel-in, serial-out
  4. parallel-in, parallel-out
- Data may be rotated left or right. Data may be shifted from left to right or right to left at will, i.e. in a bidirectional way.
- Also, data may be shifted in serially (in either way) or in parallel and shifted out serially (in either way) or in parallel.



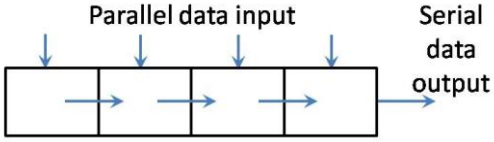
Serial-in, serial-out shift-right, shift register



Parallel-in, parallel-out, shift register



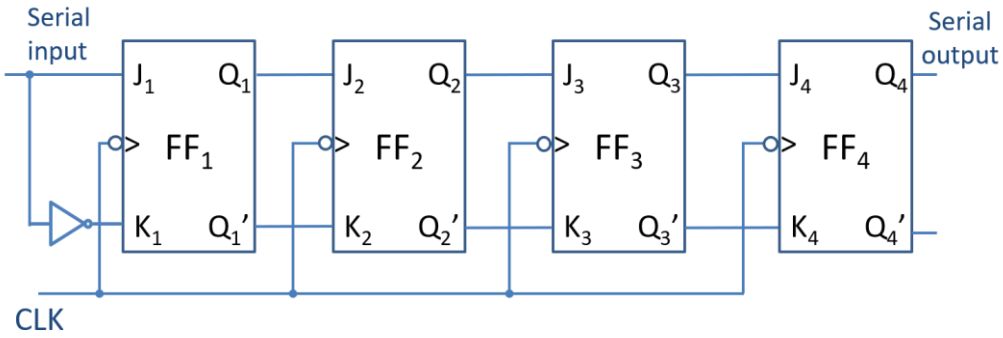
Serial-in, parallel-out, shift register



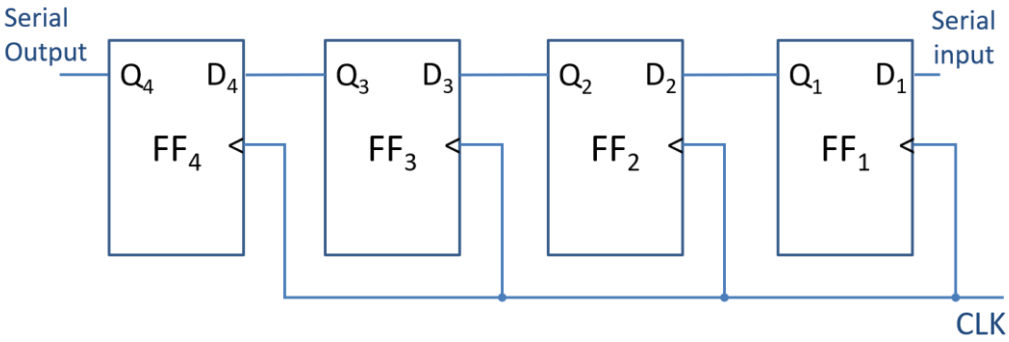
Parallel-in, serial-out, shift register

4.2.1.1. Serial-in, Serial-out, Shift register

- With four stages, i.e. four FFs, the register can store up to four bits.
- Serial data is applied at the D input of the first FF. The Q output of the first FF is connected to the D input of the second FF, the Q output of the second FF is connected to the D input of the third FF and the Q output of the third FF is connected to the D input of fourth FF.
- When serial data is transferred into a register, each new bit is clocked into the first FF at the positive edge of each clock pulse.
- The bit that was previously stored by the first FF is transferred to the second FF. The bit that was stored by the second FF is transferred to the third FF, and so on.



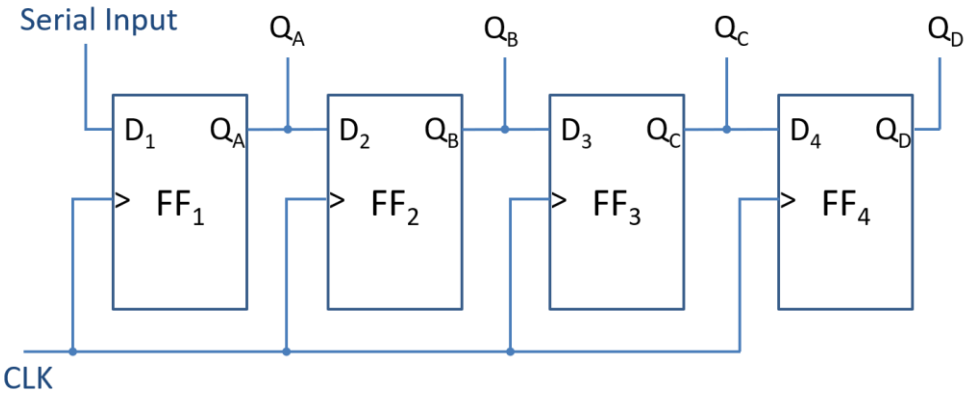
- A shift register can also be constructed using J-K FFs as shown in above figure.
- The data is applied at the J input of the first FF, the complement of this is fed to the K input of FF.
- The Q output of the first FF is connected to J input of the second FF, the Q output of the second FF to J input of the third FF, and so on.
- Also, Q_1' is connected K_2, Q_2' is connected to K_3, and so on.



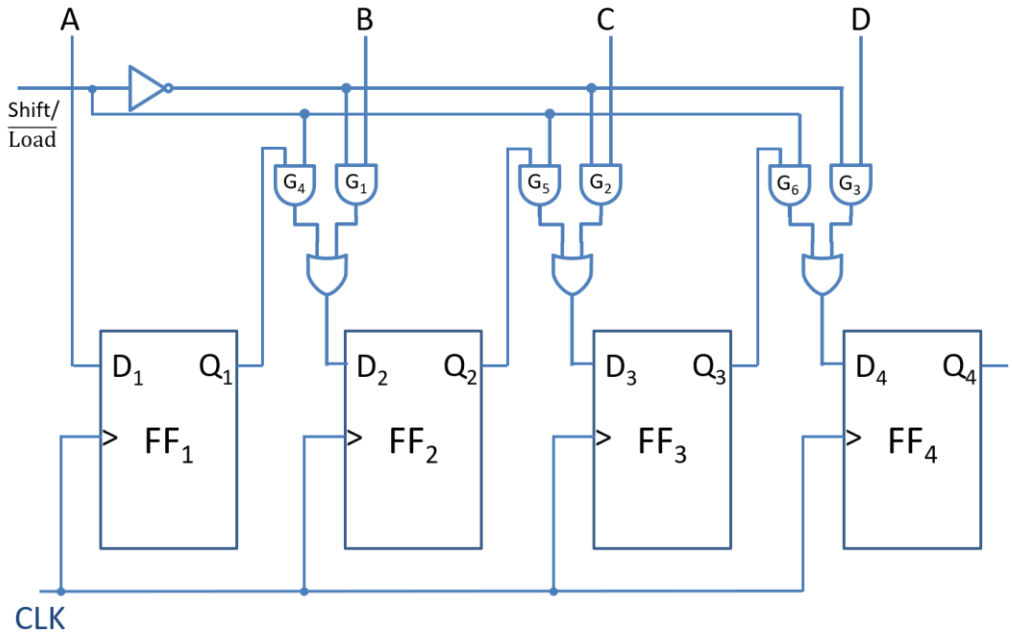
4.2.1.2. Serial-in, Parallel-out, Shift register

- In this type of register, the data bits are entered into the register serially, but the data stored in the register is shifted out in parallel form.
- Once the data bits are stored, each bit appears on its respective output line and all bits are available simultaneously, rather than on a bit-by-bit basis as with the serial output.

- The serial-in, parallel-out, shift register can be used as a serial-in, serial-out, shift register if the output is taken from Q terminal of the last FF.

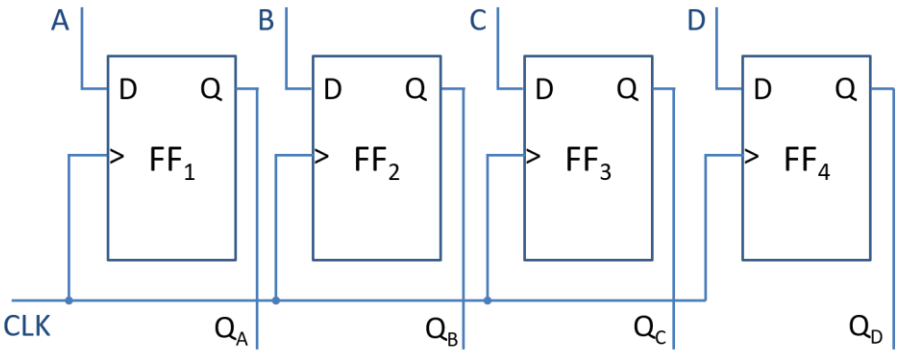


4.2.1.3. Parallel-in, Serial-out, Shift register



- There are four data lines A, B, C, and D through which the data is entered into the register in parallel form.
- The signal Shift/$\overline{LOAD}$ allows (a) the data to be entered in parallel form into the register and (b) the data to be shifted out serially from terminal Q_4.
- When Shift/$\overline{LOAD}$ line is HIGH, gates G_1, G_2, and G_3 are disabled, but gates G_4, G_5, and G_6 are enabled allowing the data bits to shift right from one stage to the next.
- When Shift/$\overline{LOAD}$ line is LOW, gates G_4, G_5, and G_6 are disabled, whereas gates G_1, G_2, and G_3 are enabled allowing the data input to appear at the D inputs of the respective FFs.
- When a clock pulse is applied, these data bits are shifted t the Q output terminals of the FFs and, therefore, data is inputted in one step.
- The OR gate allows wither the normal shifting operation or the parallel data entry depending on which NAD gates are enabled by the level on the Shift/$\overline{LOAD}$ input.

4.2.1.4. Parallel-in, Parallel-out, Shift register



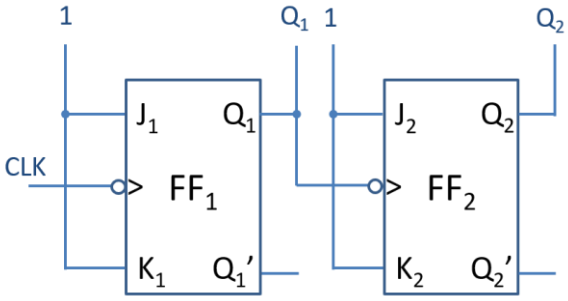
- In a parallel-in, parallel-out, shift register the data is entered into the register in parallel form, and also the data is taken out of the register in parallel form.
- Data is applied to the D input terminals of the FFs.
- When a clock pulse is applied, at the positive-going edge of that pulse, the D inputs are shifted into the Q outputs of the FFs.
- The register now stores the data. The stored data is available instantaneously for shifting out in parallel form.

4.3. Asynchronous Counter: Ripple UP/DOWN Counter, Modulo Counter

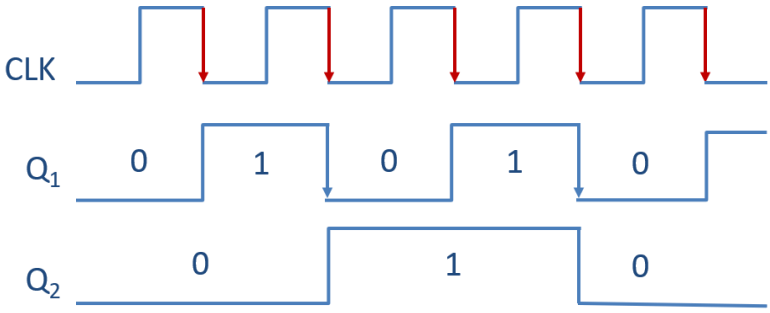
4.3.1. Difference between Asynchronous counter & Synchronous Counter

Asynchronous counter	Synchronous counter
1. In this type of counter FFs are connected in such a way that the output of first FF drives the clock for the second FF; the output of the second drives the clock of the third and so on.	1. In this type of counter there is no connection between the output of first FF and clock input of next FF and so on.
2. All the FFs are not clocked simultaneously.	2. All the FFs are clocked simultaneously.
3. Design and implementation is very simple even for more number of states.	3. Design and implementation becomes tedious and complex as the number of states increases.
4. Main drawback of these counters is their low speed as the clock is propagated through a number of FFs before it reaches the last FF.	4. Since clock is applied to all the FFs simultaneously, the total propagation delay is equal to the propagation delay of only one FF. Hence, they are faster.
5. In this type of counter FFs are connected in such a way that the output of first FF drives the clock for the second FF; the output of the second drives the clock of the third and so on.	5. In this type of counter there is no connection between the output of first FF and clock input of next FF and so on.

4.3.2. 2-bit ripple up-counter using negative edge triggered FF

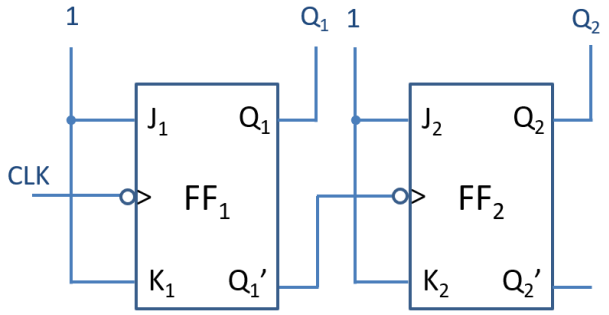


CLK	Present State		Next State	
	Q_1	Q_0	Q_1	Q_0
↓	0	0	0	1
↓	0	1	1	0
↓	1	0	1	1
↓	1	1	0	0

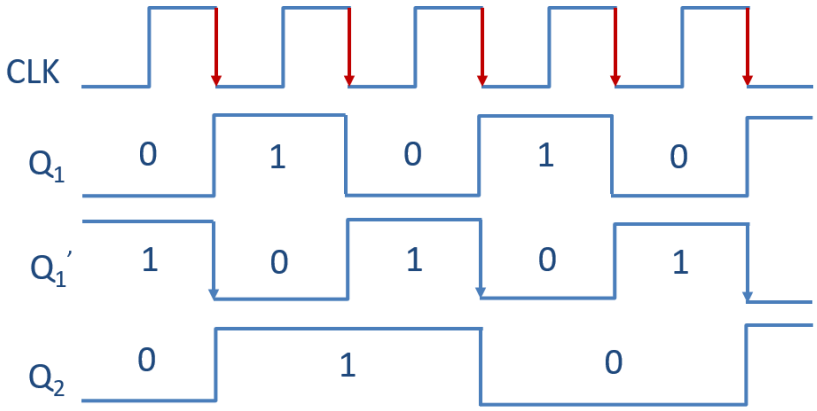


- The 2-bit up-counter counts in the order 0, 1, 2, 3, 0, … etc..
- The counter is initially reset to 00.
- When the first clock pulse is applied, FF_1 toggles at the negative edge of this pulse, therefore, Q_1 goes from LOW to HIGH.
- This becomes a positive edge at the clock input of FF_2. So FF_2 is not affected, and hence the state of the counter after one clock pulse is $Q_1 = 1$ and $Q_2 = 0$, i.e. 01.
- At the negative edge of the second clock pulse, FF_1 toggles.
- So Q_1 changes from HIGH to LOW and this negative edge clock applied to CLK of FF_2 activates FF_2, and hence, Q_2 goes from LOW to HIGH. Therefore, $Q_1 = 0$ and $Q_2 = 1$, i.e. 10 is the state of the counter after the second clock pulse.
- At the negative edge of the third clock pulse, FF_1 toggles.
- So Q_1 changes from a 0 to a 1. This becomes a positive edge to FF_2, hence FF_2 is not affected. Therefore, $Q_2 = 1$ and $Q_1 = 1$, i.e. 11 is the state of the counter after the third clock pulse.
- At the negative edge of the fourth clock pulse, FF_1 toggles.
- So, Q_1 changes from a 1 to a 0. This negative edge at Q_1 toggles FF_2, hence Q_2 also changes from a 1 to a 0. Therefore, $Q_2 = 0$ and $Q_1 = 0$, i.e. 00 is the state of the counter after the fourth clock pulse.
- So, it acts as a mod-4 counter with Q_1 as the LSB and Q_2 as the MSB.

4.3.3. 2-bit ripple down-counter using negative edge triggered FF



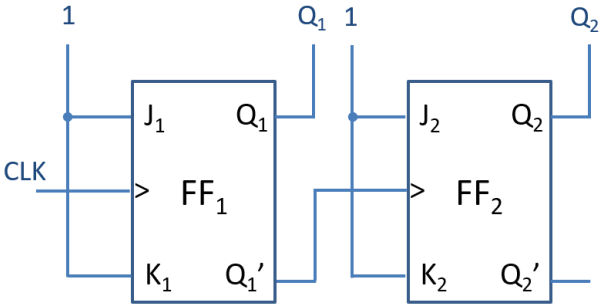
CLK	Present State		Next State	
	Q_1	Q_0	Q_1	Q_0
↓	0	0	1	1
↓	1	1	1	0
↓	1	0	0	1
↓	0	1	0	0



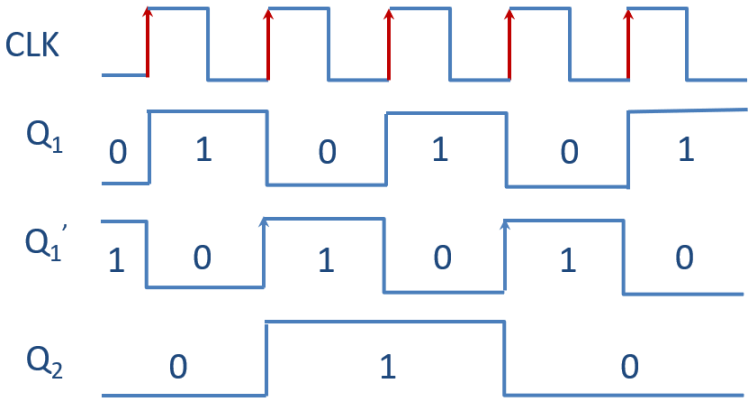
- A 2-bit down-counter counts in the order 0, 3, 2, 1, 0, …etc.
- For down counting, Q_1' of FF_1 is connected to the clock of FF_2. Let initially all the FFs be reset, i.e. 00.
- At the negative edge of the first clock pulse, FF_1 toggles. So, Q_1 goes from a 0 to a 1 and Q_1' goes from a 1 to a 0.
- This negative edge at Q_1' applied to the clock input of FF_2, toggles FF_2 and, therefore, Q_2 goes from a 0 to a 1. So, after one clock pulse $Q_2 = 1$ and $Q_1 = 1$, i.e. the state of the counter is 11.
- At the negative edge of the second clock pulse, Q_1 changes from a 1 to a 0 and Q_1' from a 0 to a 1.
- This positive edge at Q_1' does not affect FF_2 and, therefore Q_2 remains at a 1. Hence, the state of the counter after second clock pulse is 10.
- At the negative edge of third clock pulse, FF_1 toggles. So, Q_1 goes from a 0 to a 1 and Q_1' goes from a 1 to a 0.
- This negative edge at Q_1' applied to the clock input of FF_2, toggles FF_2 and, therefore, Q_2 goes from a 1 to a 0. Hence, the state of the counter is 01.
- At the negative edge of fourth clock pulse, FF_1 toggles. So, Q_1 goes from a 1 to a 0 and Q_1' goes from a 0 to a 1.

- This positive edge at Q_1' does not affect FF_2 and, therefore Q_2 remains at a 0. Hence, the state of the counter after fourth clock pulse is 00.

4.3.4. 2-bit ripple up-counter using positive edge triggered FF



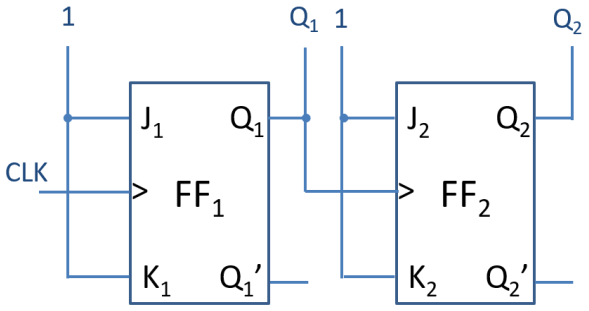
CLK	Present State		Next State	
	Q_1	Q_0	Q_1	Q_0
↑	0	0	0	1
↑	0	1	1	0
↑	1	0	1	1
↑	1	1	0	0



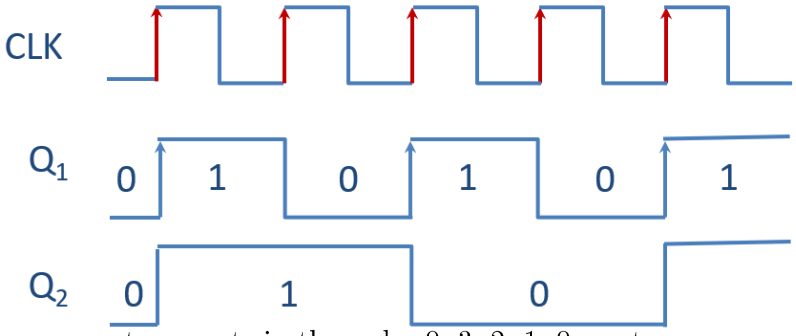
- A 2-bit up-counter counts in the order 0, 1, 2, 3, 0, …etc.
- For up counting in positive edge triggering, Q_1' of FF_1 is connected to the clock of FF_2. Let initially all the FFs be reset, i.e. 00.
- At the positive edge of the first clock pulse, FF_1 toggles. So, Q_1 goes from a 0 to a 1 and Q_1' goes from a 1 to a 0.
- This negative edge at Q_1' does not affect FF_2 and, therefore Q_2 remains at a 0. Hence, the state of the counter after first clock pulse is 01.
- At the positive edge of the second clock pulse, Q_1 changes from a 1 to a 0 and Q_1' from a 0 to a 1.
- This positive edge at Q_1' applied to the clock input of FF_2, toggles FF_2 and, therefore, Q_2 goes from a 0 to a 1. Hence, the state of the counter is 10.
- At the positive edge of third clock pulse, FF_1 toggles. So, Q_1 goes from a 0 to a 1 and Q_1' goes from a 1 to a 0.
- This negative edge at Q_1' does not affect FF_2 and, therefore Q_2 remains at a 1. Hence, the state of the counter after third clock pulse is 11.
- At the positive edge of fourth clock pulse, FF_1 toggles. So, Q_1 goes from a 1 to a 0 and Q_1' goes from a 0 to a 1.

- This positive edge at Q_1' applied to the clock input of FF_2, toggles FF_2 and, therefore, Q_2 goes from a 1 to a 0. Hence, the state of the counter is 00.

4.3.5. 2-bit ripple down-counter using positive edge triggered FF



CLK	Present State		Next State	
	Q_1	Q_0	Q_1	Q_0
↑	0	0	1	1
↑	1	1	1	0
↑	1	0	0	1
↑	0	1	0	0

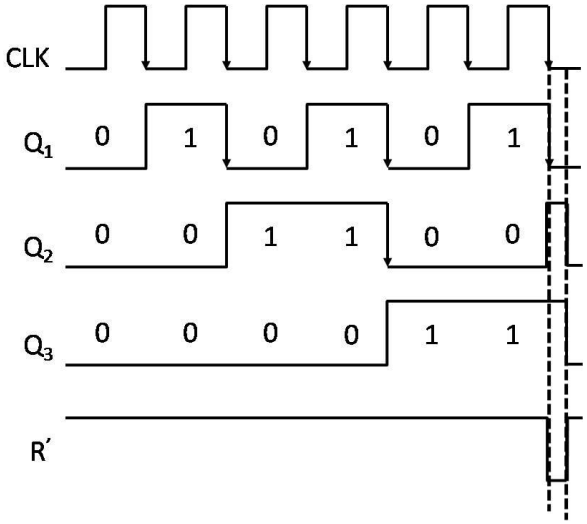


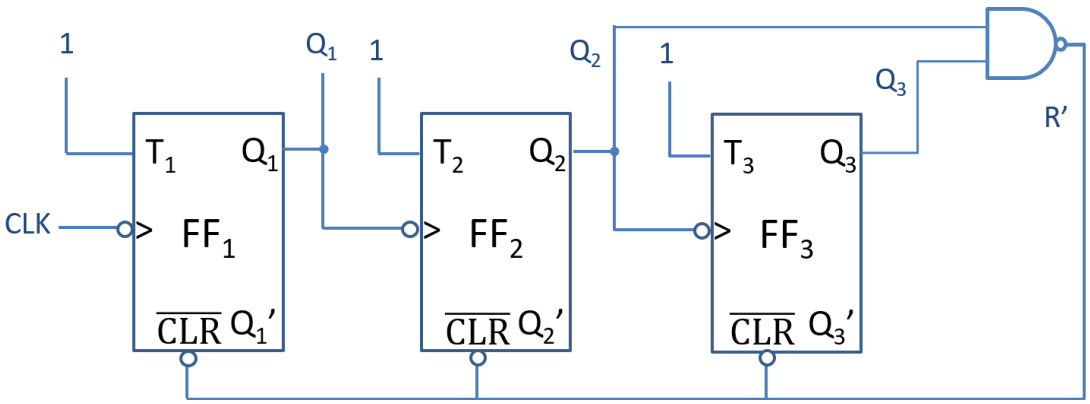
- The 2-bit down-counter counts in the order 0, 3, 2, 1, 0, … etc..
- The counter is initially reset to 00.
- When the first clock pulse is applied, FF_1 toggles at the positive edge of this pulse, therefore, Q_1 goes from LOW to HIGH.
- This positive edge at Q_1 toggles FF_2, hence Q_2 also changes from a 0 to a 1. Therefore, $Q_2 = 1$ and $Q_1 = 1$, i.e. 11 is the state of the counter after the first clock pulse.
- At the positive edge of the second clock pulse, FF_1 toggles.
- So Q_1 changes from a 1 to a 0. This becomes a negative edge to FF_2, hence FF_2 is not affected. Therefore, $Q_2 = 1$ and $Q_1 = 0$, i.e. 10 is the state of the counter after the second clock pulse.
- At the positive edge of the third clock pulse, FF_1 toggles.
- So Q_1 changes from a 0 to a 1. This positive edge at Q_1 toggles FF_2, hence Q_2 also changes from a 1 to a 0. Therefore, $Q_2 = 0$ and $Q_1 = 1$, i.e. 01 is the state of the counter after the third clock pulse.
- At the positive edge of the fourth clock pulse, FF_1 toggles.
- So Q_1 changes from a 1 to a 0. This becomes a negative edge to FF_2, hence FF_2 is not affected. Therefore, $Q_2 = 0$ and $Q_1 = 0$, i.e. 00 is the state of the counter after the fourth clock pulse.

4.3.6. Modulo-6 asynchronous counter

- A mod-6 counter has six stable states 000, 001, 010, 011, 100, 101.
- It is also known as "Divide by 6" counter and it requires 3 Flip-flops for designing.
- At the 6th clock pulse, it will again reset to 000 via feedback circuit.
- Reset signal R = 1 at time of 110, R = 0 for 000 to 101 and R = X for invalid states i.e. 111.
- Therefore, R = $Q_3Q_2Q_1' + Q_3Q_2Q_1 = Q_3Q_2$.

After Pulses	State Q_3	Q_2	Q_1	R
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
	↓	↓	↓	
	0	0	0	





4.4. Synchronous Counter

4.4.1. Synchronous counter designing

- Step 1. Number of flip-flops:
  Based on the description of the problem, determine the required number n of the FFs - the smallest value of n is such that the number of states N ≤ 2^n and the desired counting sequence. Step 2. State diagram: Draw the state diagram showing all the possible states.
- Step 3. Choice of flip-flops and excitation table:
  Select the type of flip-flops to be used and write the excitation table.

An excitation table is a table that lists the present state (PS), the next state (NS) and the required excitations.

- Step 4. Minimal expressions for excitations:
  Obtain the minimal expressions for the excitations of the FFs using K-maps for the excitations of the flip-flops in terms of the present states and inputs.
- Step 5. Logic Diagram:

4.4.2. Flip-Flop Excitation Tables

(1) S-R Flip-flop excitation table

PS	NS	Required inputs	
Q_n	Q_{n+1}	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

(2) J-K Flip-flop excitation table

PS	NS	Required inputs	
Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

(3) D Flip-flop excitation table

PS	NS	Required inputs
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

(4) T Flip-flop excitation table

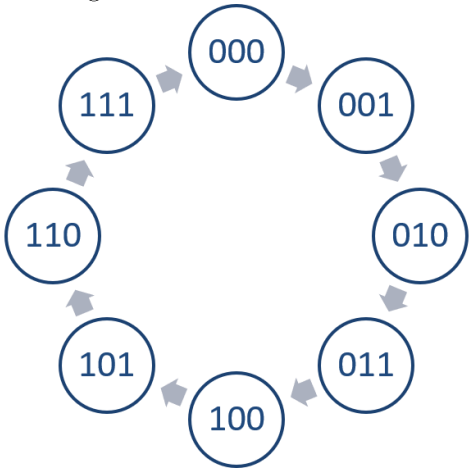
PS	NS	Required inputs
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

4.4.3. Synchronous 3-bit up counter

- Step 1. Number of flip-flops:

A 3-bit up-counter requires 3 flip-flops. The counting sequence is 000, 001, 010, 011, 100, 101, 110, 111, 000 …

- Step 2. Draw the state diagram:

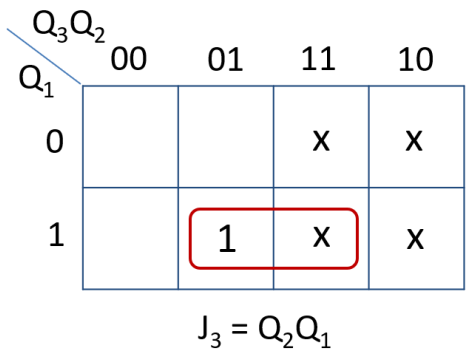


- Step 3. Select the type of flip-flops and draw the excitation table:
  JK flip-flops are selected and the excitation table of a 3-bit up-counter using J-K flip-flops is drawn as shown below.

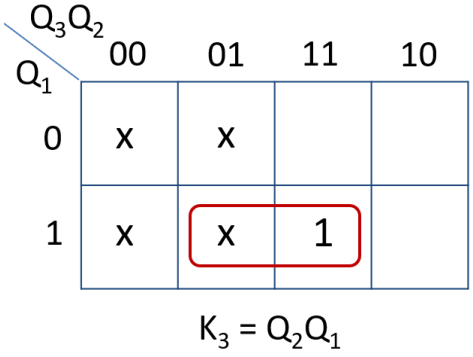
Present State			Next State			Required Excitation					
Q_3	Q_2	Q_1	Q_3	Q_2	Q_1	J_3	K_3	J_2	K_2	J_1	K_1
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	0	1	X	X	1	X	1
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	1	1	1	X	0	X	0	1	X
1	1	1	0	0	0	X	1	X	1	X	1

- Step 4. Obtain the minimal expressions
  From excitation table, $J_1 = K_1 = 1$.

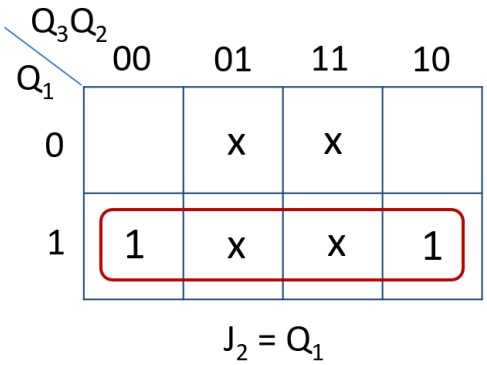
K – Maps for excitations J_3, K_3, J_2 and K_2 and their minimized form are as follows:



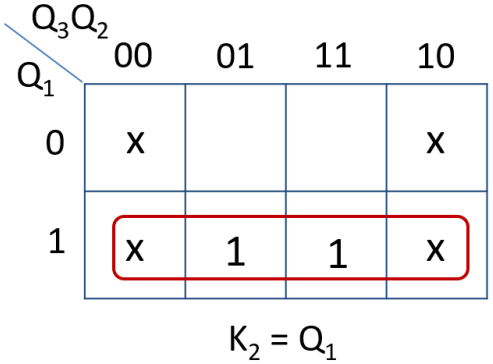
$J_3 = Q_2Q_1$



$K_3 = Q_2Q_1$

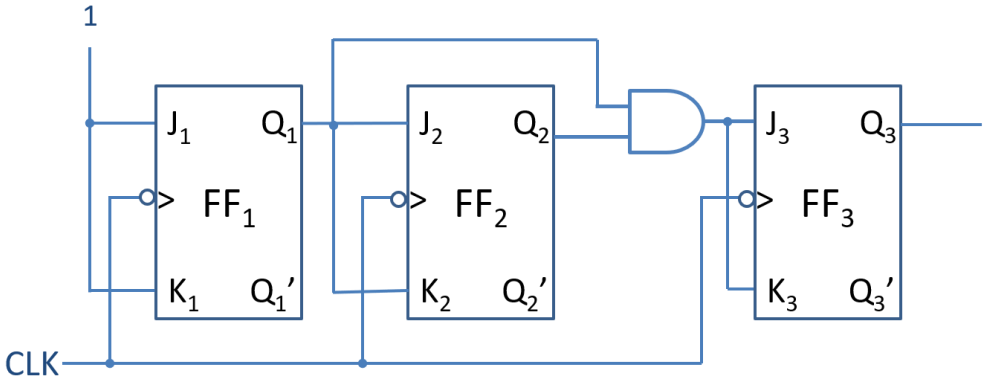


$J_2 = Q_1$



$K_2 = Q_1$

- Step 5. Draw the logic diagram



4.5. Sequential Generator: Direct & Indirect

4.5.1. Sequence generator using direct logic

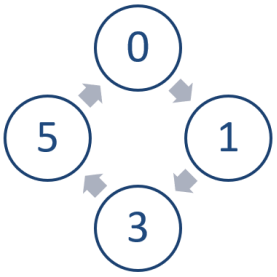
- Step 1. Inspect given pule train

- Step 2. Decide the number of unique states and minimum number of FFs required. If unique states are not possible with the least number of FFs n, then increase the number of FFs by one or more to get the unique states.

FF States	
	LSB
0	0
0	1
1	1
?	1

FF States			Decimal equivalent
		LSB	
0	0	0	0
0	0	1	1
0	1	1	3
1	0	1	5

State assignment



State diagram

- Step 3. Select the type of flip-flops and draw the excitation table:
  JK flip-flops are selected and the excitation table of a given sequence using J-K flip-flops is drawn as shown below.

PS			NS			Required excitations					
Q_3	Q_2	Q_1	Q_3	Q_2	Q_1	J_3	K_3	J_2	K_2	J_1	K_1
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	1	0	X	1	X	X	0
0	1	1	1	0	1	1	X	X	1	X	0
1	0	1	0	0	0	X	1	0	X	X	1

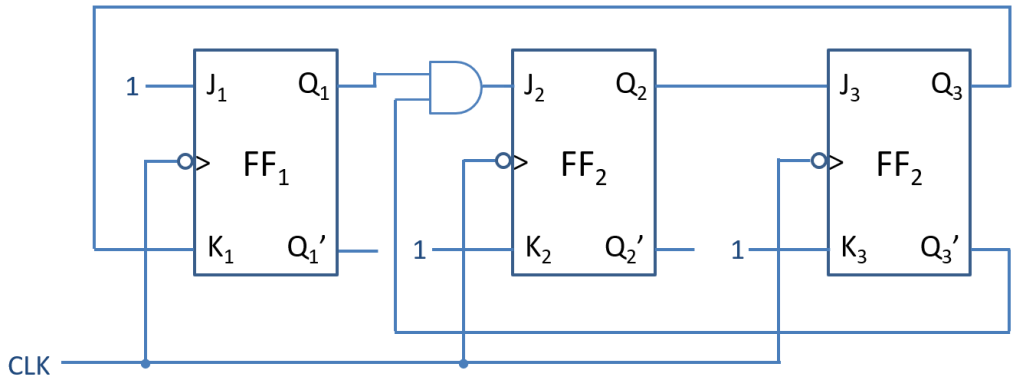
- Step 4. Obtain the minimal expressions

  Using K – Maps, we can obtain minimal expressions as shown in below.

  $J_3 = Q_2,\ K_3 = 1$ $\qquad$ $J_2 = Q_3'Q_1,\ K_2 = 1$

  $J_1 = 1,\ K_1 = Q_3$

- Step 5. Draw the logic diagram



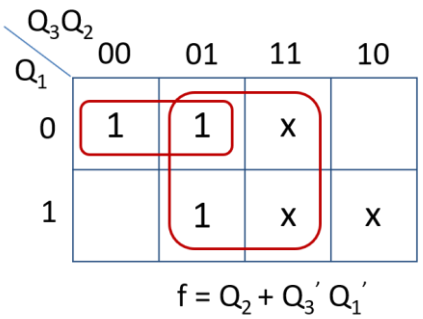
4.5.2. Sequence generator using indirect logic

- Step 1. Inspect given pule train

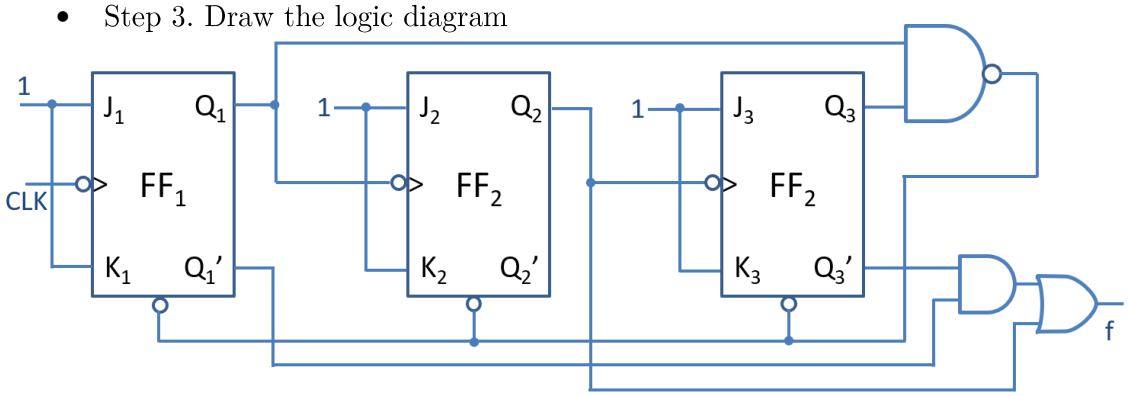
- Step 2. Obtain output function f to generate the inspected sequence and also obtain minimal expression of function f using K – Map.

We need to reset counter to 000 at the state 101, so expression of R should be Q_3Q_1.

Q_3	Q_2	Q_1	Output(f)	States
0	0	0	1	0
0	0	1	0	1
0	1	0	1	2
0	1	1	1	3
1	0	0	0	4
1	0	1	X	5
1	1	0	X	6
1	1	1	X	7



- Step 3. Draw the logic diagram

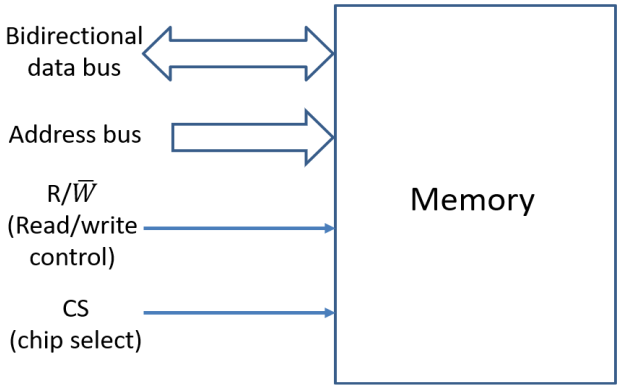


Unit – 5

Memories and Programmable Logic Devices

5.1. Memory organization and operation

- The number of inputs required to store the data into or read the data from any memory location is N.
- One set of N lines is required for storing the data into the memory, referred to as data inputs and another set of N lines is required for reading the data already stored in the memory, which is referred to as data outputs.
- The concept of bus is used to refer to a group of conductors carrying related set of signals.
- Therefore, the set of lines meant for data inputs is input data bus and for data outputs is output data bus.
- Input and output data buses are unidirectional, i.e. the data can flow in one direction only.
- In most of the memory chips available, the same set of lines is used for data input as well as data output and is referred to as bidirectional bus.



- This means that the data bus is time multiplexed.
- It is used as input bus for some specific time and as output bus for some other time depending upon a Read/Write control input as shown in figure.
- A number of control inputs are required to give commands to the device to perform the desired operation.
- For example, a command signal is required to tell the memory whether a read or a write (R/$\overline{W}$ in figure) operation is desired.
- When R/$\overline{W}$ is HIGH, the data bus will be used for reading the memory (output bus) whereas when R/$\overline{W}$ is LOW, the bus will be acting in the input direction and the data on the bus will go into the memory.
- Other command includes inputs chip enable (CE), chip select (CS), etc.
- There are mainly two types of operations performed by memory unit.

1. Write operation

- The chip select signal is applied to the CS terminal.

- The word to be stored is applied to the data-input terminal.
- The address of the desired memory location is applied to the address-input terminals.
- A write command signal is applied to the write control input terminal with $R/\overline{W} = 0$.

2. Read operation

- The chip select signal is applied to the CS terminal.
- The address of the desired memory location is applied to the address-input terminals.
- A read command signal is applied to the read control input terminal with $R/\overline{W} = 1$.

5.1.1. Memory expansion

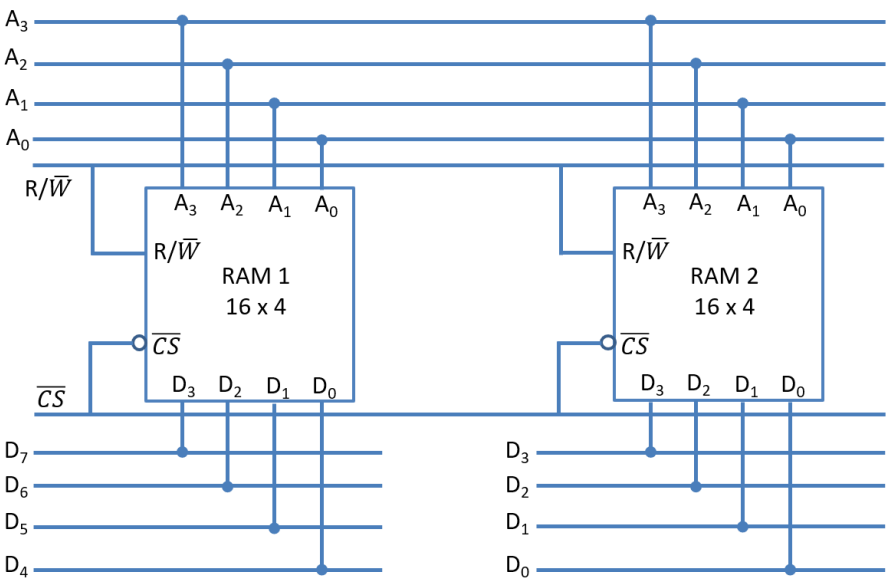
- In many applications, the required memory capacity, i.e. the number of words and/or word size, can not be satisfied by a single available memory IC chips.
- Therefore, several chips have to be combined suitably to provide the desired number of words and/or word size.

Expanding word size

- If it is required to have a memory of word size n and the word size of the available memory ICs is N (n > N), then a number of similar ICs can be combined together to achieve the desired word size.
- The number of IC chips required is an integer, next higher to the value n/N.
- These chips are to be connected in the following way:
  1. Connect the corresponding address lines of each chip individually, i.e. A_0 of each chip is connected together and it becomes A_0 of the overall memory. Similarly, connect other address lines together.
  2. Connect the RD input of each IC together and it becomes the read input for the overall memory. Similarly, connect the WR and CS inputs.

Example: Show how to combine several 16 x 4 RAM to form a 16 x 8 RAM.

Solution: n = 8 and N = 4, so we require n/N = 2 chips to obtain desired memory.



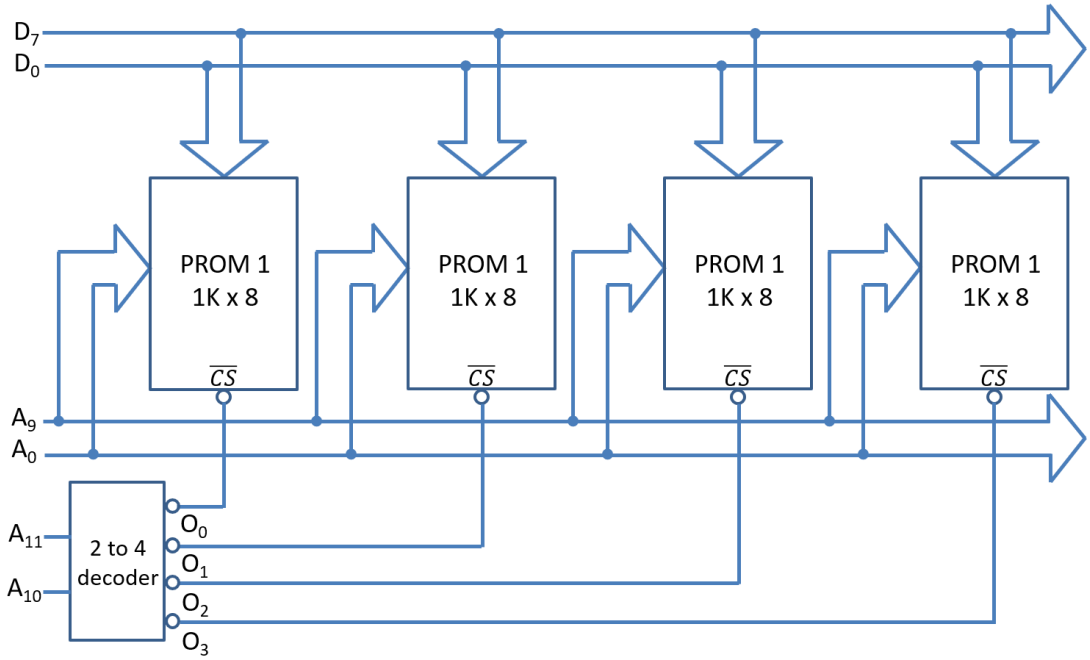
Expanding word capacity

- To obtain a memory of capacity m words, using the memory chips with M words each, the number of chips required is an integer next higher to the value m/M.
- These chips are to be connected in the following way:
  1. Connect the corresponding address lines of each chip individually.
  2. Connect the RD input of each chip together. Similarly, connect the WR inputs.
  3. Use a decoder of proper size and connect each of its outputs to one of the CS terminals of memory chips.

Example: Show how to combine several 1K x 8 PROMs to produce 4K x 8 PROM.

Solution: 1K x 8 PROM has 10 number of address lines because $2^{10} = 1024$ (1K).

We need total 4 number of 1K x 8 PROM chips to make one 4K x 8 PROM chip.



A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1

Memory chip	Starting address	Ending address
PROM 1	000 H	3FF H
PROM 2	400 H	7FF H
PROM 3	800 H	BFF H
PROM 4	C00 H	FFF H

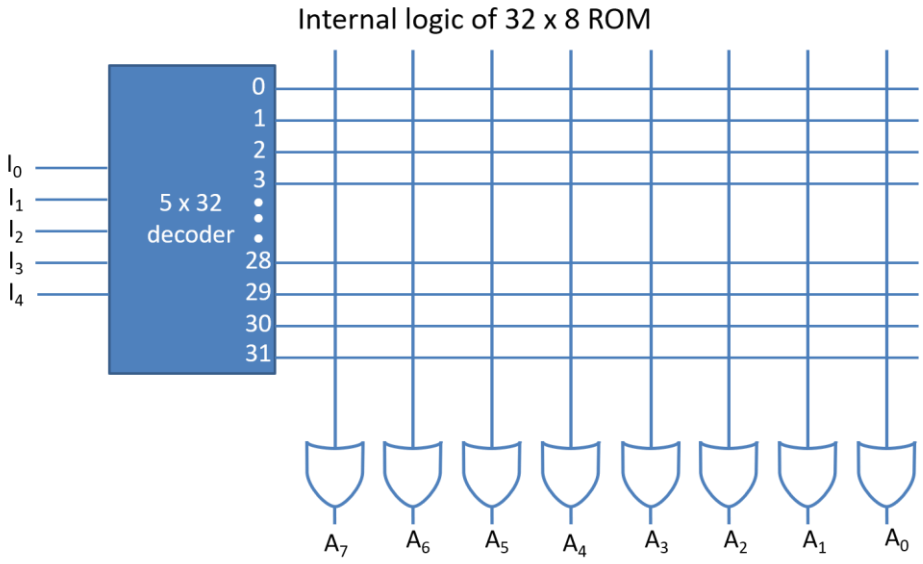
5.2. Classification of memory: ROM, RAM, ROM as PLD

5.2.1. Read Only Memory (ROM)

- A read-only memory (ROM) is a semiconductor memory device used to store information which is permanent in nature.
- It has become an important part of many digital systems because of its low cost, high speed, system-design flexibility and data non-volatility.
- The read-only memory has a variety of applications in digital systems, such as implementation of combinational logic and sequential logic, character generation, look-up table, microprocessor program storage, etc.
- ROMS are well-suited for LSI manufacturing processes and are available in many forms.
- Two major semiconductor technologies are used for the manufacturing of ROM integrated circuits, viz. bipolar technology and MOS technology, which differ primarily in access time.
- In general, bipolar devices are faster and have higher drive capability, whereas MOS devices require less silicon area and consume less power.
- With improvements in MOS technology, it is now possible to make MOS memories with speeds comparable to those of bipolar memories.
- The process of entering information into a ROM is referred to as programming the ROM.
- Depending on the programming process employed, the ROMS are categorized as:
  1. *Mask programmable read-only memories*, which are referred to as ROMS. In these memories, the data pattern must be programmed as part of the fabrication process. Once programmed, the data pattern can never be changed. These are highly suited for very high-volume usage due to their low cost.
  2. *Programmable read-only memories*, which are referred to as PROMs. A PROM is electrically programmable, i.e. the data pattern is defined after final packaging rather than when the device is fabricated. The programming is done with an equipment referred to as PROM programmer. The programming techniques used will be discussed later.
  3. *Erasable programmable read-only memories*, which are referred to as EPROMs. As the name suggests, in these memories, data can be written any number of times, i.e. they are reprogrammable. Reprogrammable ROMs are possible only in MOS technology. For erasing the contents of the memory, one of the following two methods are employed:
     (a) Exposing the chip to ultraviolet radiation for about 30 minutes.
     (b) Erasing electrically by applying voltage of proper polarity and amplitude. Electrically erasable PROM is also referred to as EIPROM or EAROM (Electrically alterable ROM).

5.2.1.1. ROM Organization

- k inputs – provide address for the memory
- n outputs – data bits of the stored word selected by address
- k address input lines specify 2^k words
- ROM does not have data inputs because it does not have write operation.
- Consider, for example, a 32 x 8 ROM.
- The unit consists of 32 words of 8 bits each.
- There are five input lines that form the binary numbers from 0 through 31 for the address.
- The five inputs are decoded into 32 distinct outputs by means of a 5 x 32 decoder.
- Each output of the decoder represents a memory address.
- The 32 outputs of the decoder are connected to each of the 8 OR gates.
- Each OR gate must be considered as having 32 inputs.
- Since each OR gate has 32 input connections and there are 8 OR gates, the ROM contains 32 x 8 = 256 internal connections.
- In general, a 2^k x n ROM will have an internal k x 2^k decoder and n OR gates.
- Each OR gate has 2^k inputs, which are connected to each of the outputs of the decoder.



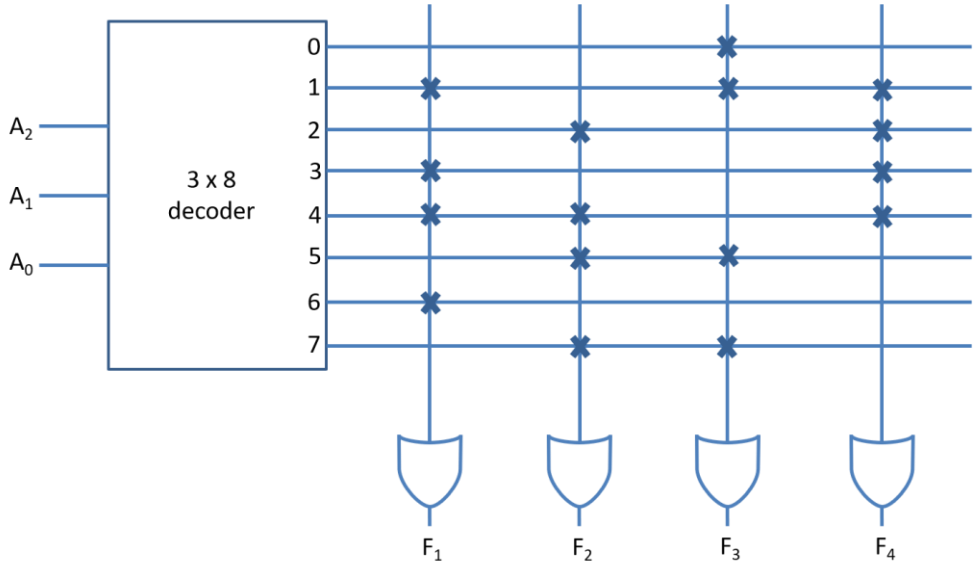
Example: Implement following functions using ROM.

$F_1 = \sum_m(1,3,4,6)$ $F_2 = \sum_m(2,4,5,7)$

$F_3 = \sum_m(0,1,5,7)$ $F_4 = \sum_m(1,2,3,4)$

Solution: First we have to decide that which decoder has been used to implement given example. We have 3 variable functions, so 3-to-8 decoder will be used

A_2	A_1	A_0	F_1	F_2	F_3	F_4
0	0	0	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	1	0	1
0	1	1	1	0	0	1
1	0	0	1	1	0	1
1	0	1	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	1	1	0



5.2.2. Random Access Memory (RAM)

- Many digital systems require memories in which it should be possible to write into, or read from, any memory location with the same speed.
- In such memories, the data stored at any location can be changed during the operation of the system.
- This type of memory is known as a read/write memory and is usually referred to as RAM (random-access memory).
- The read-write memories (RWM)/random-access memories (RAM) are fabricated using bipolar devices or unipolar (MOS) devices.
- There are two types of RAMs. These are static RAM (SRAM) and dynamic RAM (DRAM).
- Bipolar RAMs are static, whereas the MOS RAMs can be static or dynamic.
- The basic storage cell of a static RAM is a bistable circuit, i.e., a latch, which simply consists of two cross-coupled inverters.
- A RAM is an array of these storage cells requiring as many FLIP-FLOPs as the bit storage capacity of the RAM, which is usually a large number.

- The storage cell of a DRAM is simply a capacitor, therefore, only MOS devices can be used for dynamic random-access memories.
- Since capacitors leak charge, therefore, the voltage stored in it gets reduced with time which requires periodic refreshing.
- In general, bipolar RAMs are faster than the MOS RAMs.
- With improvements in the MOS technology, it has become possible to make MOS RAMS with speeds (access time) comparable to those of bipolar RAMs.

Static RAM v/s Dynamic RAM

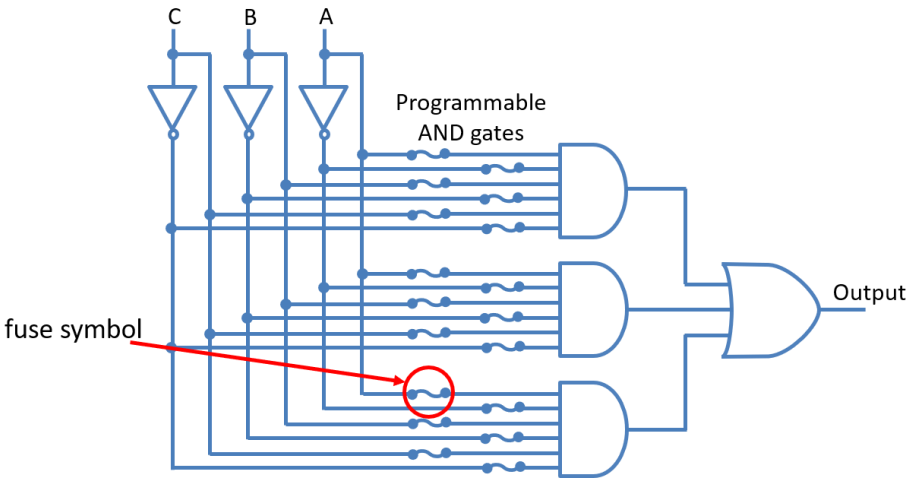
Static RAM	Dynamic RAM
1. SRAM has lower access time, so it is faster compared to DRAM.	1. DRAM has higher access time, so it is slower than SRAM.
2. SRAM is costlier than DRAM.	2. DRAM costs less compared to SRAM.
3. SRAM requires constant power supply, which means this type of memory consumes more power.	3. DRAM offers reduced power consumption, due to the fact that the information is stored in the capacitor.
4. Due to complex internal circuitry, less storage capacity is available compared to the same physical size of DRAM memory chip.	4. Due to the small internal circuitry in the one-bit memory cell of DRAM, the large storage capacity is available.
5. SRAM has low packaging density.	5. DRAM has high packaging density.
6. No need to refresh periodically.	6. Due to capacitor used as storage element, information may lose over period of time. So, need to refresh periodically.
7. Uses an array of 6 transistors for each memory cell.	7. Uses a single transistor and capacitor for each memory cell.

RAM v/s ROM

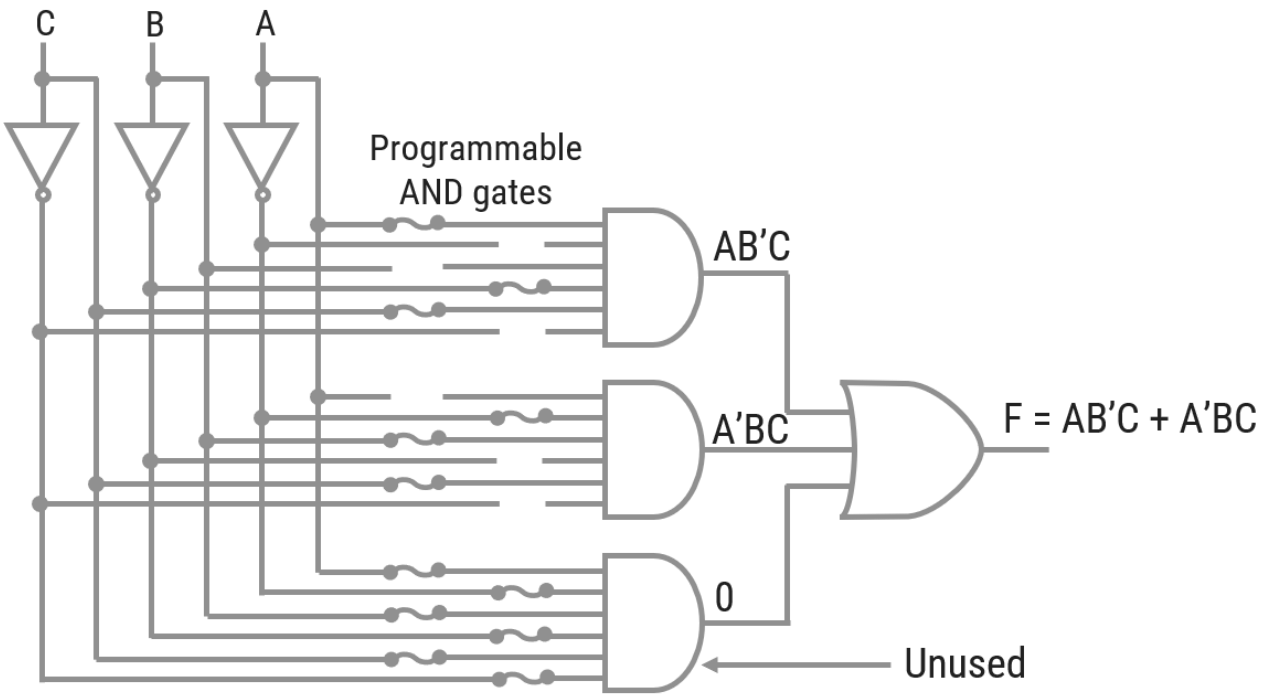
Parameter	RAM	ROM
Data	The data is not permanent but it can be altered any number of times.	The data is permanent. It can be altered but only a limited number of times that too at slow speed.
Speed	It is a high-speed memory.	It is much slower than the RAM.
CPU Interaction	The CPU can access the data stored on it.	The CPU cannot access the data stored on it. In order to do so, the data is first copied to the RAM.
Size and Capacity	Large size with higher capacity.	Small size with less capacity.
Usage	Primary memory (DRAM DIMM modules), CPU Cache (SRAM).	Firmware like BIOS or UEFI. RFID tags, microcontrollers, medical devices, and at places where a small and permanent memory solution is required.
Cost	It doesn't come cheap.	Way cheaper than RAM.

5.3. Programmable Array Logic (PAL)

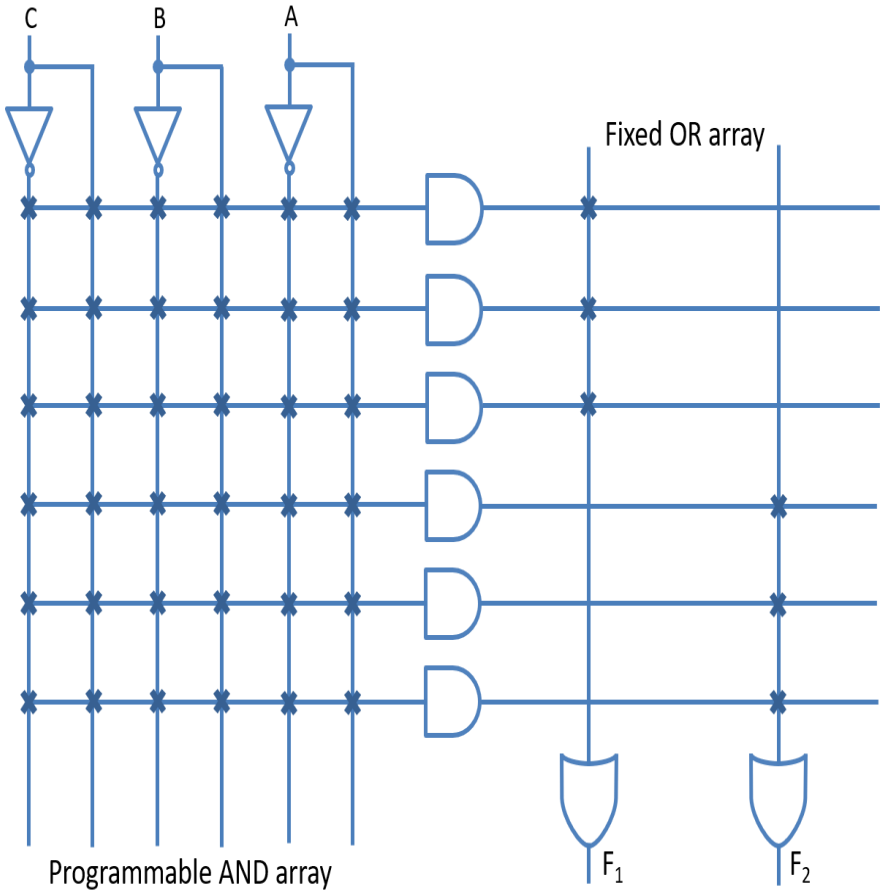
- Programmable array logic (a registered trade mark of Monolithic Memories) is a particular family of programmable logic devices (PLDs) that is widely used and available from a number of manufacturers.
- The PAL circuits consist of a set of AND gates whose inputs can be programmed and whose outputs are connected to an OR gate, i.e. the inputs to the OR gate are hard-wired, i.e. PAL is a PLD with a fixed OR array and a programmable AND array.
- Because only the AND gates are programmable, the PAL is easier to program but is not as flexible as the PLA. Some manufacturers also allow output inversion to be programmed.
- Thus, like AND-OR and AND-OR-INVERT logic, they implement a sum of products logic function.
- Figure-1 below shows a small example of the basic structure.
- The fuse symbols represent fusible links that can be burned open using equipment similar to a PROM programmer.
- Note that every input variable and its complement can be left either connected or disconnected from every AND gate.
- We then say that the AND gates are programmed.
- Figure-2 below shows how the circuit is programmed to implement F = A'BC + AB'C.
- Note this important point. All input variables and their complements are left connected to the unused AND gate, whose output is, therefore, AA'BB'CC' = 0.



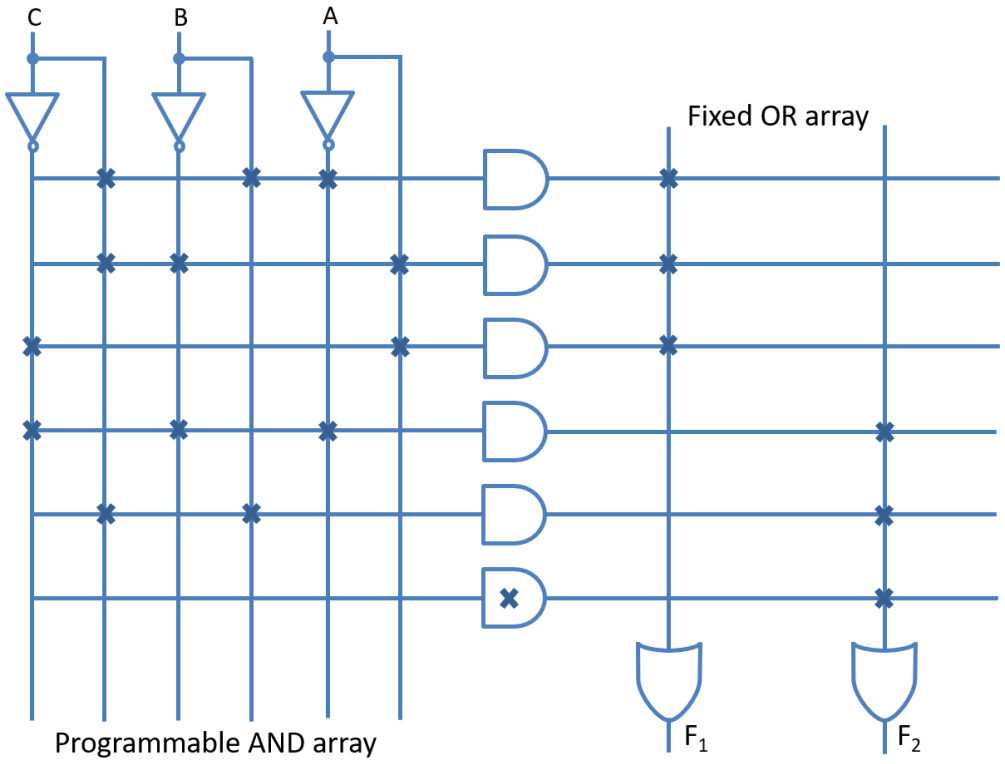
- The 0 has no effect on the output of the OR gate. On the other hand, if all inputs to the unused AND gate were burned open, the output of the AND gate would 'float' HIGH (logic 1), and the output of the OR gate in that case would remain permanently 1.
- The actual PAL circuits have several groups of AND gates, each group providing inputs to separate OR gates.



- Figure-3 shows an example of how the PAL structure is represented using the abbreviated connections.
- It is a 3-input 3-wide AND-OR structure. In this example, each function can have three minterms or product terms.
- Notice that there are six AND gates, which implies only six chosen products of not more than three variables ABC.
- Inputs to the OR gates at the outputs are fixed as shown by x marked on the vertical lines.
- The inputs to the AND gates are marked on the corresponding line by the x.
- Removing the x implies blowing off the corresponding fuse which in turn implies that the corresponding input variable is not applied to the particular AND gate.
- In this example, the circuit is unprogrammed because all the fusible links are intact.
- Note that, the 3-input OR gates are drawn with a single input line.



Example: Implement following functions using PAL: F_1 = A'BC + AC' + AB'C and F_2 = A'B'C' + BC.



5.3.1. PAL Programming table

- The fuse map of a PAL can be specified in a tabular form. The PAL programming table consists of three columns.
- The first column lists the product terms numerically. The second column specifies the required paths between inputs and AND gates. The third column specifies the outputs of the OR gates.
- For each product term the inputs are marked with 1, 0, or – (dash).
- If a variable in the product term appears in its true form, the corresponding input variable is marked with a 1.
- If it appears in complemented form, the corresponding input variable is marked with a 0.
- If the variable is absent in the product term, it is marked as a – (dash).
- The paths between the inputs and the AND gates are specified under the column heading *inputs* in the programming table.
- A 1 in the input column specifies a connection from the input variable to the AND gate.
- A 0 in the input column specifies a connection from the complement of the variable to the input of the AND gate.
- A – (dash) specifies a blown fuse in both the input variable and its complement.
- It is assumed that a open terminal in the input of an AND gate behaves like a 1.
- The outputs of the OR gates are specified under the column heading *outputs*.
- The size of a PAL is specified by the number of inputs, the number of product terms, and the number of outputs.
- For n inputs, k product terms, and m outputs the internal logic of the PAL consists of n buffer inverter gates, k AND gates, and m OR gates.
- When designing a digital system with a PAL, there is no need to show the internal connections of the unit.
- All that is needed is a PAL programming table from which the PAL can be programmed to supply the required logic.

Example: Implement the following Boolean functions using PAL with four inputs and 3-wide AND-OR structure. Also write the PAL programming table.

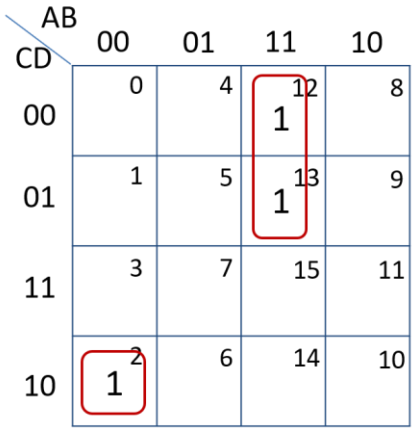
$F_1(A, B, C, D) = \sum_m(2, 12, 13)$

$F_2(A, B, C, D) = \sum_m(7, 8, 9, 10, 11, 12, 13, 14, 15)$

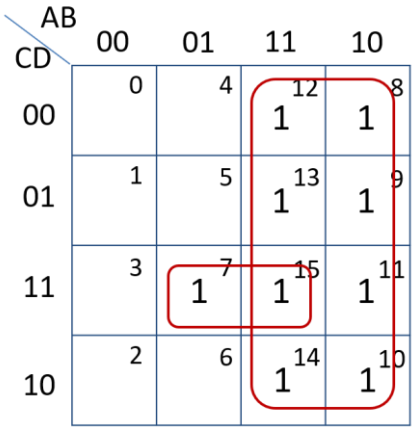
$F_3(A, B, C, D) = \sum_m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$

$F_4(A, B, C, D) = \sum_m(1, 2, 8, 12, 13)$

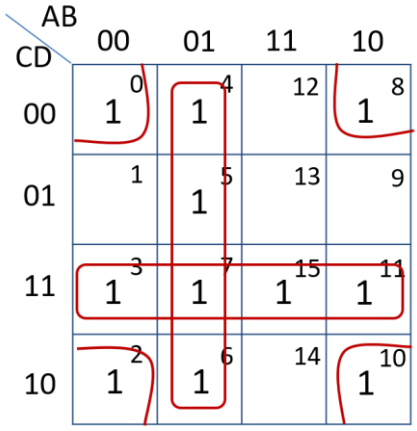
Solution:



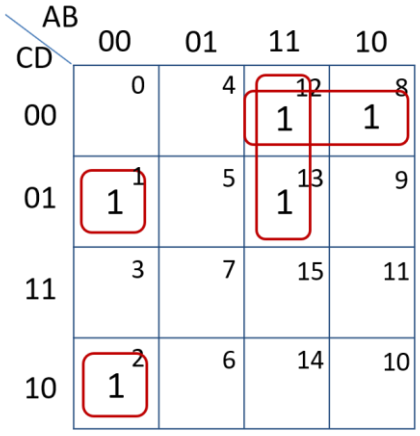
$F_1 = ABC' + A'B'CD'$



$F_2 = A + BCD$

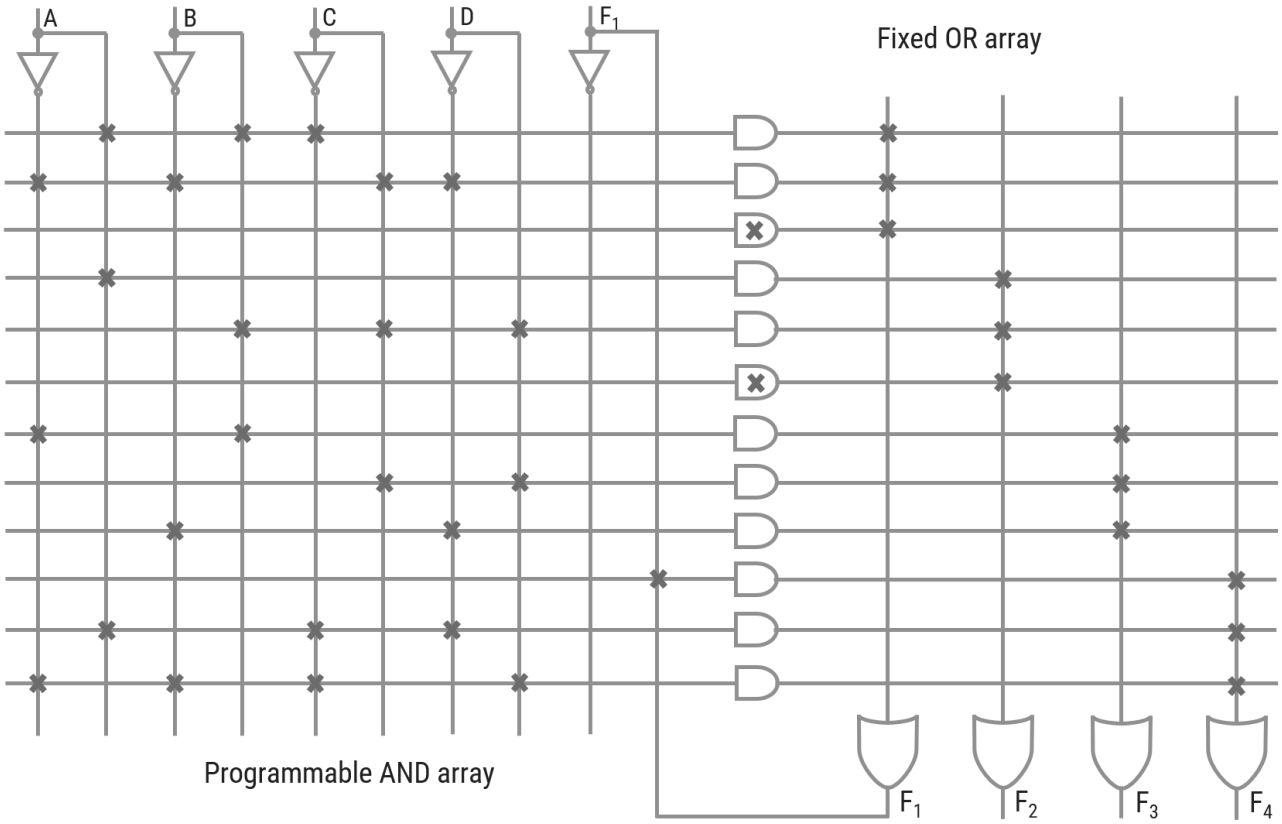


$F_3 = CD + A'B + B'D'$



$F_4 = ABC' + AC'D' + A'B'C'D + A'B'CD'$

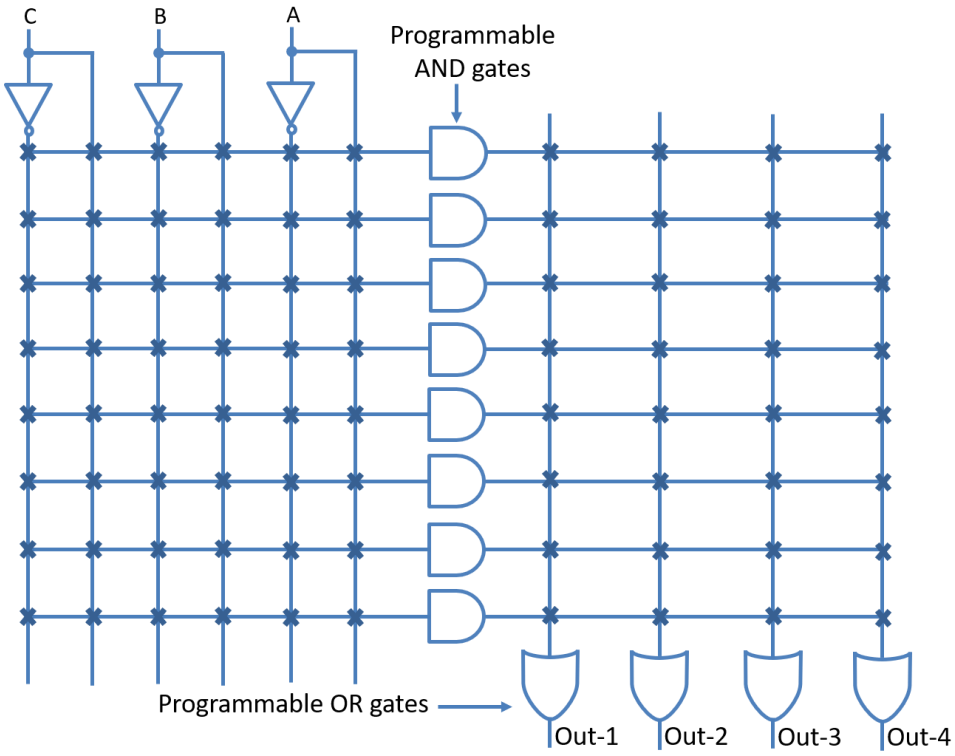
Product term	AND Inputs					Outputs
	A	B	C	D	F_1	
1	1	1	0	-	-	$F_1 = ABC' + A'B'CD'$
2	0	0	1	0	-	
3	-	-	-	-	-	
4	1	-	-	-	-	$F_2 = A + BCD$
5	-	1	1	1	-	
6	-	-	-	-	-	
7	0	1	-	-	-	$F_3 = CD + A'B + B'D'$
8	-	-	1	1	-	
9	-	0	-	0	-	
10	-	-	-	-	1	$F_4 = F_1 + AC'D' + A'B'C'D$
11	1	-	0	0	-	
12	0	0	0	1	-	



5.4. Programmable Logic Array (PLA)

- The PLA represents another type of programmable logic but with a slightly different architecture.
- The PLA combines the characteristics of the PROM and the PAL by providing both a programmable OR array and a programmable AND array, i.e. in a PLA both AND gates and OR gates have fuses at the inputs.
- A third set of fuses in the output inverters allows the output function to be inverted if required.
- Usually X-OR gates are used for controlled inversion. This feature makes it the most versatile of the three PLDs.
- However, it has some disadvantages. Because it has two sets of fuses, it is more difficult to manufacture, program and test it than a PROM or a PAL.
- Figure demonstrates the structure of a three-input, four-output PLA with every fusible link intact.
- Like ROM, PLA can be mask programmable or field programmable. With a mask programmable PLA, the user must submit a PLA programming table to the manufacturer.

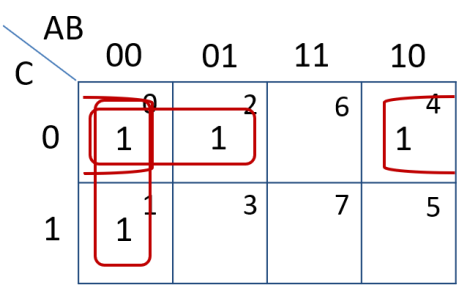
- This table is used by the vender to produce a user made PLA that has the required internal paths between inputs and outputs.
- A second type of PLA available is called a field programmable logic array or FPLA.
- The FPLA can be programmed by the user by means of certain recommended procedures.



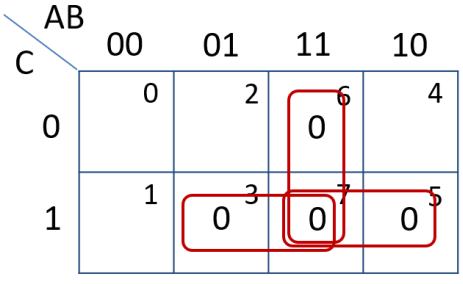
Example: Implement the following two Boolean functions with a PLA:

$F_1(A, B, C) = \sum_m(0, 1, 2, 4)$
$F_2(A, B, C) = \sum_m(0, 5, 6, 7)$

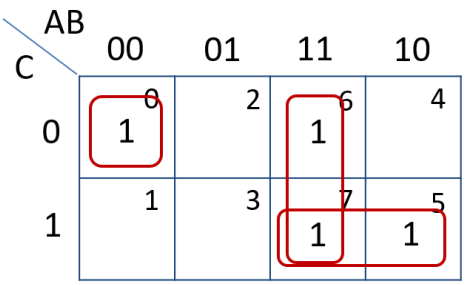
Solution:



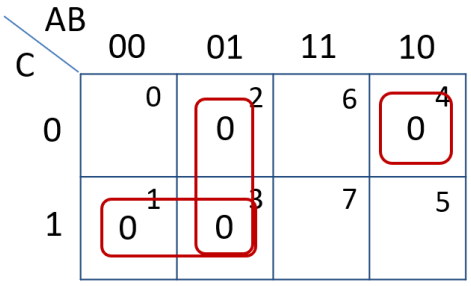
$F_1(T) = A'C' + B'C' + A'B'$



$F_1 = (A'+B')\,(B'+C')\,(A'+C')$
$F_1' = AB + AC + BC$
$F_1(C) = (AB + AC + BC)'$



$F_2(T) = A'B'C' + AB + AC$



$F_2 = (A+B')\,(A+C')\,(A'+B+C)$

$F_2' = A'B + A'C + AB'C'$

$F_2(C) = (A'B + A'C + AB'C')'$

Product term		Inputs			Outputs	
		A	B	C	$F_1(C)$	$F_2(T)$
1	AB	1	1	-	1	1
2	AC	1	-	1	1	1
3	BC	-	1	1	1	-
4	A'B'C'	0	0	0	-	1