Syllabus for B.Tech. Electronics And Communication Engineering

Digital System Design
Code: EC-302
Contacts: 3L
Credits: 3

Module I (10L)
Review of Number System, Signed and Unsigned Number.
Logic Simplification and Combinational Logic Design: Review of Boolean Algebra and De-Morgan's Theorem, SOP & POS forms, Canonical forms, Karnaugh's map, Binary codes, Code Conversion.
MSI devices like Comparators, Multiplexers, Encoder, Decoder, Half and Full Adders, Subtractors, Serial and Parallel Adders, BCD Adder, Fast adders, Barrel shifter and ALU.

Module II (6L)
Ripple and Synchronous counters, Shift registers, Finite state machines, Design of synchronous FSM. Designing synchronous circuits like Synchronous Counter, Pulse train generator, Pseudo Random Binary Sequence generator.

Module III (8L)
Logic Families and Semiconductor Memories: TTL, ECL, CMOS families.
Semiconductor Memories, Concept of Programmable logic devices like FPGA. Logic implementation using Programmable Devices.
Different types of A/D and D/A conversion techniques. Sample and hold circuit.

Unit IV (8L)
VLSI Design flow: Design entry Schematic, FSM & HDL, different modeling styles in VHDL, Data types and objects, Data flow, Behavioral and Structural Modeling, Synthesis and Simulation VHDL constructs and codes for combinational and sequential circuits.

Text/Reference Books:

1. R.P. Jain, "Modern digital Electronics", Tata McGraw Hill, 4th edition, 2009.
2. Schilling & Belove, Digital Integrated Electronics, Tata McGraw Hill,
3. Douglas Perry, "VHDL", Tata McGraw Hill, 4th edition, 2002.
4. W.H. Gothmann, "Digital Electronics- An introduction to theory and practice", PHI, 2nd edition ,2006.
5. D.V. Hall, "Digital Circuits and Systems", Tata McGraw Hill, 1989
6. Charles Roth, "Digital System Design using VHDL", Tata McGraw Hill 2nd edition 2012.
7. R. Anand, "Digital Electronics", Khanna Publishing House, New Delhi, 2017.

Course outcomes:
At the end of this course students will demonstrate the ability to
1. Design and analyze combinational logic circuits
2. Design & analyze modular combinational circuits with MUX/DEMUX, Decoder, Encoder
3. Design & analyze synchronous sequential logic

Lesion Plan, Digital System Design (EC302), Session 2023-24, ODD SEM

Module-I (10L)

- Review of Number System (2L):
  - Lec1: Unsigned Number - 1L
  - Lec2: Signed Number - 1L
- Logic Simplification and Combinational Logic Design (5L):
  - Lec3: Review of Boolean Algebra and De-Morgan’s Theorem - 1L
  - Lec4: SOP & POS forms, Canonical forms - 1L
  - Lec5-6: Karnaugh’s map - 2L
  - Lec7-8: Binary codes, Code Conversion - 2L
- MSI devices like (3L):
  - Lec9-10: Comparators, Multiplexers, Encoder, Decoder - 2L
  - Lec11-12: Half and Full Adders, Subtractors, Serial and Parallel Adders - 2L lecture
  - Lec-13: BCD Adder, Fast adders, Barrel shifter and ALU - 1L

Module-II (6L)

- Lec14: Ripple and Synchronous counters - 1L
- Lec15: Shift registers - 1L
- Finite state machines (4L):
  - Lec16: Design of synchronous FSM - 1L
  - Designing synchronous circuits like:
    - Lec17-18: Synchronous Counter - 2L
    - Lec19: Pulse train generator, Pseudo Random Binary Sequence generator - 1L

Module III (8L)

- Logic Families and Semiconductor Memories:
  - Lec20-21: TTL, ECL, CMOS families - 2L
- Lec22: Semiconductor Memories - 1L
- Lec23: Concept of Programmable logic devices like FPGA - 1L
- Lec24: Logic implementation using Programmable Devices - 1L
- Different types of A/D and D/A conversion techniques (3L):
  - Lec25: Sample and hold circuit - 1L
  - Lec26: D/A Converters - 1L
  - Lec27: A/D Converters - 1L

Unit IV (8L)

- Lec28-29: VLSI Design flow: Design entry Schematic, FSM & HDL - 2L
- Lec30-33: different modeling styles in VHDL, Data types and objects, Data flow, Behavioral and Structural Modeling - 4L
- Lec34-35: Synthesis and Simulation VHDL constructs and codes for combinational and sequential circuits - 2L

Lecture Notes

Module-I:
NUMBER SYSTEMS: Binary, Octal and Hexadecimal representation and their conversions; BCD, ASCII, EBDIC, Gray codes and their conversions; Signed binary number representation with 1's and 2's complement methods, Binary arithmetic.
4

Introduction:
What is Electronics?
Electronics means study of flow of electrons in electrical circuits. The word Electronics comes from electron mechanics which means learning the way how an electron behaves under different conditions of externally applied fields. IRE – The Institution of Radio Engineers has given a definition of electronics as "that field of science and engineering, which deals with electron devices and their utilization." Fundamentals of electronics are the core subject in all branches of engineering nowadays.

Electronics is distinct from electrical and electro-mechanical science and technology, which deals with the generation, distribution, switching, storage, and conversion of electrical energy to and from other energy forms using wires, motors, generators, batteries, switches, relays, transformers, resistors, and other passive components. This distinction started around 1906 with the invention of the triode by Lee De Forest, which made electrical amplification of weak radio signals and audio signals possible with a non-mechanical device. Until 1950 this field was called "radio technology" because its principal application was the design of radio transmitters, receivers, and vacuum tubes.

Analog electronics are electronic systems with a continuously varying signal, in contrast to digital electronics where signals usually take only two different levels. The term "analogue" describes the proportional relationship between a signal and a voltage or current that represents the signal. The word analogue is derived from the Greek word "Analogos" meaning "proportional".

BINARY NUMBER SYSTEM
We are comfortable with Decimal Number System having base or radix of 10. Implementation of decimal system in digital circuits thus requires signals (voltage/current) to be divided into 10 levels. The same reduces the reliability of the system giving birth of binary system which uses only two levels – logic 0 and logic 1. In digital circuits, logic 0 is represented by 0V and logic 1 is represented by 5V.

Binary number system is also a positional weighted system like decimal number system where each binary digit (called bit) carries a certain weight according to its position with respect to the decimal point as shown below:

Table-1: Weights of the bits of a binary number with 8-bit integer and 4-bit fraction.

Bit	d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	.	d_{-1}	d_{-2}	d_{-3}	d_{-4}
Weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	128	64	32	16	8	4	2	1	.	0.5	0.25	0.125	0.0625

Example1: Convert 110100_b to its equivalent decimal

$$\begin{aligned} 110100_b &= 1x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 0x2^0 \\ &= 32 + 16 + 0 + 4 + 0 + 0 \\ &= 52_d \end{aligned}$$

Example-2: Convert 110.101_b to its equivalent decimal

$$\begin{aligned} 110.101_b &= 1\times2^2 + 1\times2^1 + 0\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} \\ &= 4 + 2 + 0 + 0.5 + 0 + 0.125 \\ &= 6.625_d \end{aligned}$$

These examples illustrate binary to decimal conversion. Decimal to binary conversion is as follows:

- First divide the number at the decimal point and treat the two parts separately
- For the integer part, repeatedly divide it by 2 and store the remainder until nothing is left
- Write the remainders from bottom to top to get the binary number. The reverse-ordering comes as the first division by 2 gives the *least significant bit* (lsb) and so on until the last division which gives the *most significant bit* (msb).
- For the fractional part, repeatedly multiply the fraction part by 2 and record the carries i.e. when the resulting number is greater than 1. Repeat this process until the desired precision is achieved.

Example-3: Convert **57.4801_d** to its binary equivalent to 14-bit accuracy.

As the decimal number has both an integer and a fractional part the problem has to be done in two steps

First take the integer part i.e. 57 and repeatedly divide by 2 noting the remainders of each division.

57 / 2 = 28 remainder **1 lsb**
28 / 2 = 14 remainder **0**
14 / 2 = 7 remainder **0**
7 / 2 = 3 remainder **1**
3 / 2 = 1 remainder **1**
1 / 2 = 0 remainder **1 msb**

The binary equivalent of **57_d** is therefore given by the remainders ordered from *most significant bit* (**msb**) to *least significant bit* (**lsb**) and is hence **1110001_b**.

The fractional part is given by repeatedly multiplying by 2 and storing the carries (when the result of the multiplication exceeds 1) until the required bit accuracy is reached.

.4801 x 2 = .9602 + **0**
.9602 x 2 = .9204 + **1**
.9204 x 2 = .8408 + **1**
.8408 x 2 = .6816 + **1**
.6816 x 2 = .3632 + **1**
.3632 x 2 = .7264 + **0**
.7264 x 2 = .4528 + **1**
.4528 x 2 = .9056 + **0**

and so, **$57.4801_d = 111001.01111010_b$**

OCTAL NUMBER SYSTEM

Octal number system was extensively used in earlier microcomputer system. It is also a positional weighted system having base or radix of 8. Eight symbols 0, 1, 2, 3, 4, 5, 6 and 7 are used to express octal numbers. As the base is $8 = 2^3$, every 3-bit group of binary can be represented by an octal digit. Conversion between octal and binary is therefore extremely easy.

Example-4: Convert 567.34_8 to equivalent binary

Octal number	5	6	7	.	3	4
3-bit binary for each octal digit	101	110	111	.	011	100
Result, putting all bits together	101110111			.	011100_2	

Example-5: Convert 110101001.011101_2 to equivalent octal

The binary number writing in groups of 3-bits starting from the decimal point on either side	110	101	001	.	011	101
Octal digits, one for each group	6	5	1	.	3	5
Result, putting all digits together			651	.	35_8	

Octal to decimal conversion can be achieved by multiplying the octal digits by their respective weights and then adding them together.

Example-6: Convert 567.34_8 to equivalent decimal

$567.34_8 = 5x8^2 + 6x8^1 + 7x8^0 + 3x8^{-1} + 4x8^{-2} = 320 + 48 + 7 + 0.375 + 0.0625 = 375.4375_{10}$

Decimal to octal conversion for integer requires repeated division by 8 till the quotient is 0 and then putting them together from bottom to top. On the other hand, for fractions multiply the decimal fractions by 8 repeatedly till the fraction part of the product is 0 or till the required accuracy is achieved.

Example-7: Convert 567.98_{10} to equivalent octal number

First let us take the integer part of the decimal number which will be successively divided by 8 to get the octal integer

567 / 8 = 70 with remainder 7 lsd
70 / 8 = 08 with remainder 6
08 / 8 = 01 with remainder 0
01 / 8 = 00 with remainder 1 msd

The octal equivalent of the integer part of the decimal number 567_{10} is given by the remainders written from msd to lsd and is hence 1067_8

Now let us take the fraction part of the decimal number and multiply successively the fraction by 8 to get the octal fraction

.98 x 8 = 7.84 (msd)
.84 x 8 = 6.72
.72 x 8 = 5.76
.76 x 8 = 6.08
.08 x 8 = 0.64
.64 x 8 = 5.12 and so on ………

Thus the fractional part of the octal number is 0.765605_8

So, the complete result is $567.98_{10} = 1067.765605_8$

HEXADECIMAL NUMBER SYSTEM

A useful way of expressing long pure binary coded numbers is by the use of hexadecimal numbers i.e. base 16. This is because each group of four bits (called a *nibble* since 2 nibbles make a byte) can be converted into one hexadecimal number. The mapping between binary, decimal and hexadecimal (hex.) numbers is shown below.

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9

2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

To convert a binary number into its hexadecimal equivalent, first ensure that the binary number has a few bits that is a multiple of 4, if not add zeros to the left until it does. Of course, this is true in case of unsigned/positive numbers; in case of negative number, add ones to the left. Then split the number into nibbles and convert each nibble into its hexadecimal counterpart.

Example-8: Convert the binary number **1101101101101** to its hexadecimal equivalent.

The binary number **1101101101101** has 13 digits. First extend this to a multiple of 4 (i.e. 16) by adding three leading 0s to the number, i.e.

11011011001101 becomes 0001101101101101

Next break the binary number up into nibbles (4-bit groups) and convert each nibble to its hexadecimal equivalent.

Binary	0001	1011	0110	1101
Hexadecimal	1	B	6	D

Hence, **1101101101101_b = $1B6D_h$**

SIGNED NUMBERS

In decimal system, + and – signs are used to represent positive and negative numbers. However, in binary only 0's and 1's are recognized. So a 1 or a 0 is used to represent whether a number is positive or negative. A 0 at the left indicates that the number is positive and a 1 for negative. However there are a number of ways a negative number can be represented. These are:

a) Sign-Magnitude (Signed Magnitude) Method:

In signed magnitude method, most significant bit (msb) is the sign bit and remaining bits are used to represent the magnitude of the number. For example, in an 8-bit binary number system, +/-125_{10} and +/-100_{10} can be represented as:

$+125_{10}$ = $0111\ 1101_2$, -125_{10} = $1111\ 1101_2$

$+100_{10}$ = $0110\ 0100_2$, -100_{10} = $1110\ 0100_2$

Arithmetic operation in signed magnitude method needs to check the signs and magnitudes of the operands before the actual operation to be performed. This increases the complexity of the circuit concerned. Also it takes more time. The complication may be avoided by representing negative numbers by their complements viz. 1's complement and 2's complement.

b) 1's Complement Method:

In this method also the msb represents the sign of the number and remaining bits represent the magnitude. But unlike signed magnitude method, the magnitude of a negative number is represented with bit by bit complements of the corresponding positive number. +/-125_{10} and +/-100_{10} can be represented in 1's complement method as:

$+125_{10}$ = $0111\ 1101_2$, -125_{10} = $1000\ 0010_2$, $+100_{10}$ = $0110\ 0100_2$, -100_{10} = 10011011_2

Arithmetic operation also needs no decision making before the actual operation to be performed. But any carry generated for addition of two numbers in this method needs to be added with the result to get the correct value.

c) 2's Complement Method:

In 2's complement method too, the msb is used to represent the sign bit and the remaining bits represent the magnitude. But the magnitude of a negative number is represented by the 2's complement of the corresponding positive number. 2's complement is calculated by adding 1 to 1's complement of the number. +/-125_{10} and +/-100_{10} can be represented in 2's complement method as:

$+125_{10}$ = $0111\ 1101_2$, -125_{10} = $1000\ 0011_2$, $+100_{10}$ = $0110\ 0100_2$, -100_{10} = $1001\ 1100_2$

Arithmetic operation also needs no decision making before the actual operation to be performed. Carry generated for addition of two numbers in this method is discarded.

Example-9: Represent the following decimal numbers in 8-bit singed magnitude, 1's and 2's complement forms: **$+35_d$, -35_d, $+65_d$, -65_d, $+127_d$, -127_d, $+128_d$, -128_d**

Decimal Number	8-it Binary Number			Remarks
	Signed Magnitude	1's Complement	2's Complement	
+35	0 001 0011	0 001 0011	0 001 0011	2's complement representation has only one 0 unlike signed magnitude and 1's complement system
-35	1 001 0011	1 110 1100	1 110 1101	
+65	0 010 0001	0 010 0001	0 010 0001	
-65	1 010 0001	1 101 1110	1 101 1111	
+127	0 111 1111	0 111 1111	0 111 1111	
-127	1 111 1111	1 000 0000	1 000 0001	

+128	Out of range. Range is: -127 to -0, +0 to +127	Out of range. Range is: -127 to -0, +0 to +127	Out of range. Range is: -128 to -1, 0, +1 to +127	which have two 0s namely -0 and +0
-128			1000 0000	

Shortcut methods of finding 2's complements are as follows:

Method-2: -128 in 2's complement is 1000 0000 and thus the sign bit (msb) can be considered to have a positional weight of -128 and other places have their usual weights but positive i.e. the weights of the bits of an 8-bit 2's complement representation are -128, +64, +32, +16, +8, +4, +2, +1. So any −ve number can be expressed as sum of -128 and some positive values in 2's powers and then can be represented by putting 1s in their respective positions.

Example-10: Represent, -128, -127, -80, -10 and -1 in 2's complement system

---

a) $-128_d = -128+0 = 1000\ 0000_2$
b) $-127_d = -128+1 = 1000\ 0001_2$
c) $-80_d = -128 +32+16 = 1011\ 0000_2$
d) $-10_d = -128+64+32+16+4+2 = 1111\ 0110_2$
e) $-1_d = -128+64+32+16+8+4+2+1 = 1111\ 1111_2$

Method-3: First represent the positive number in equivalent binary; proceed from the right (lsb), keep the bits unchanged up to and including the first 1 and then complement all the remaining bits.

Example-11: Express the numbers of Example-10 in 2's complement following the third method.

-ve Number	Corresponding +ve number	Binary representation of +128	-ve number in 2's complement form
-128_D	128_D	1000 0000	$1000\ 0000_2$
-127_D	127_D	0111 1111	$1000\ 0001_2$
-80_D	80_D	0101 0000	$1011\ 0000_2$
-10_D	10_D	0000 1010	$1111\ 0110_2$
-1_D	1_D	0000 0001	$1111\ 1111_2$

BINARY ARITHMETIC

Signed Magnitude Arithmetic

Binary addition and subtraction in signed magnitude method require certain decision before and after the actual operation is performed. Let us take the following examples:

```
+ 98      +98     -98      - 98
+ 78      -78     +78      - 78
-----    ------  ------   -------
+176      +20     -20      -176
```

The observations are:

a) If signs of both the operands are identical, they will be added and the sign of the result will be that of either of them.
b) If their signs are different, the smaller operand will be subtracted from the larger one and the sign of the result will be that of the larger operand.

Implementation of signed magnitude arithmetic requires more hardware. Decision making also takes larger time to complete the operation and thus is not followed in computer system. Rather subtraction in computer system is implemented through addition; the complement of the subtrahend is added with the minuend.

1's Complement Arithmetic

a) Express the subtrahend in 1's complement form
b) Add it with the minuend
c) If there is a carry out, bring the carry around and add it to the lsb. This is called the *end around carry*.
d) If the sign bit (msb) is 0, the result is +ve
e) If it is 1, the result is –ve and is in 1's complement form. Take its 1's complement to get the magnitude of the result.

Example-12: Perform using 8-bit 1's complement method: a) 127 – 59, b) 59 – 127, c) -35 – 65 and d) 35 + 65

```
a)   127     0111 1111
     -59    +1100 0100           1's complement of 59
    ------  -----------------
     +68    10100 0011
                     +1          End around carry to be added with lsb
            ------------------
             0100 0100
```

The msb is 0, the result is +ve and is in pure binary. Therefore, the result is $+68_d$.

```
b)    59      0011 1011
    -127    + 1000 0000          1's complement of 127
    ------  -----------------
    - 68      1011 1011
```

The msb is 1, the result is –ve and is in 1's complement form. 1's complement of 1011 1011 is0100 0100. Therefore, the result is -68.

```
c)   -35     1101 1100           1's complement of 35
     -65    +1011 1110           1's complement of 65
    -----   -----------------
    -100    11001 1010
                    +1           End around carry
            ------------------
             1001 1011
```

The msb is 1, the result is –ve and is in 1’s complement form. 1’s complement of 1001 1011 is 0110 0100. Therefore, the result is -100.

```
d)    35      0010 0011
     +65     +0100 0001
    ------   -----------------
    +100      0110 0100
```

The msb is 0, the result is +ve and is in pure binary. Therefore, the result is $+100_d$.

2’s Complement Arithmetic

a) Express the subtrahend in 2’s complement form
b) Add it with the minuend
c) Ignore carry out, if any
d) If the sign bit (msb) is 0, the result is +ve
e) If it is 1, the result is –ve and is in 2’s complement form. Take its 2’s complement to get the magnitude of the result.

Example-13: Perform using 8-bit 2’s complement method: a) 127 – 59, b) 59 – 127, c) -35 – 65 and d) 35 + 65

```
a)    127     0111 1111
      -59    +1100 0101  2’s complement of 59
    ------   ----------
     + 68    10100 0100
              0100 0100  Final result ignoring carry
```

The msb is 0, the result is +ve and is in pure binary. Therefore, the result is $+68_d$.

```
b)     59     0011 1011
      -127   +1000 0001  2’s complement of 127
    ------   ----------
     - 68     1011 1100
```

The msb is 1, the result is –ve and is in 2’s complement form. 2’s complement of 1011 1100 is 0100 0100. Therefore, the result is -68.

```
c)    -35     1101 1101  2’s complement of 35
      -65    +1011 1111  2’s complement of 65
     -----   ----------
     -100    11001 1100
              1001 1100  Final result ignoring carry
```

The msb is 1, the result is –veand is in 2’s complement form. 2’s complement of 1001 1100 is 0110 0100. Therefore, the result is -100.

```
d)     35     0010 0011
        +65  +0100 0001
    ------   ----------
    +100      0110 0100
```

The msb is 0, the result is +ve and is in pure binary. Therefore, the result is $+100_d$.

Example-14: Add 30.125 to -93.625 using 12-bit (8-bit integer plus 4-bit fraction) 2’s complement arithmetic.

	30.125	0001 1110.0010	
	-93.625 ----------	+ 1010 0010.0110 ----------------	2’s complement of 93.625 (express 93.625 in binary and then take 2’s complement of the whole number ignoring the decimal point)
	-63.500	1100 0000.1000	No carry

The msb is 1, the result is –ve and is in 2’s complement form. 2’s complement of 1100 0000.1000 is 0011 1111.1000. Therefore, the result is -63.5.

Example-15: Considering 8-bit 2’s complement system, add (i) -100 -30, (ii) 100 +30 and explain the overflow condition

----

30d = $0001\ 1110_b$ -30d = $1110\ 0010_b$ 100d = $0110\ 0100_b$ -100d = $1001\ 1100_b$

```
-30          1110 0010
-100         1001 1100
---------------------------
-130       1 0111 1111
```

-130 is out of the range as in 8-bit 2’s complement system the range of number is: -128 to +127. This is an overflow condition and is satisfied by the carry flow (no carry from D6 but a carry from D7).

```
100          0110 0100
+30          0001 1110
-------------------------------
+130   0     1000 0010
```

+130 is again out of the range. This is an overflow condition and is satisfied by the carry flow (a carry from D6 but no carry from D7).

Note:

1. In case of –ve mixed number (integer plus fraction), if you observe carefully the integer part in 2’s complement form, you will find that it is actually the 2’s complement of the next higher number. For example, 1010 0010 is 2’s complement of 94 (not of 93) i.e. -94. Therefore, the fraction should be positive and it must be +0.375 so that -94 + 0.375 will be producing -93.625. *So in complement representation, fraction is always considered to be positive.*
2. In case of signed arithmetic, if the result is not limited within the range of the numbers of the system, overflow occurs. Consider two numbers a and b. If we subtract a from b, the result can't have a greater magnitude than either a or b. Therefore, adding two numbers of different sign cannot generate overflow. If a and b have the same sign, but a+b has a different sign, then overflow has occurred.
3. Also overflow is said to occur if there is a carry either from D6 or from D7 position but not from the both.

Binary Codes

Representation of Numerals and Alphabets plus Numerals in terms of binary numbers is called Binary Coding. The binary number itself is called the binary code. The binary codes representing numerals are called Numeric codes and that representing Alphanumeric information are called Alphanumeric Codes.

Binary Codes

1. **Numeric:** Numeric Codes may also be classified as Weighted, Non-weighted, Self Complementing, Sequential, Error Detecting and Correcting, Reflective and Cyclic Codes.
   - **Weighted:** In weighted code, each binary bit position is having a fixed value called weight. Examples are Binary and BCD codes. BCD codes may also be classified as positively weighted and negatively weighted. In positively weighted codes, all the positional weights are positive and for negatively weighted codes, some of the positional weights are negative. Examples of positively weighted codes are: 8421, 4221, 2421, 5211, 3321 and 4311 are some of 17 such codes. Examples of negatively weighted codes: 642-3, 631-1, 84-2-1 etc. All these are BCD codes.
   - **Non-weighted:** In non-weighted code, the bits are having no fixed values or weights. Examples are: Gray, Excess-3 etc. **Non-weighted codes are not suitable for arithmetic operations.**
   - **Self Complementing:** 9's complement can be calculated by interchanging 0's and 1's in the code. 2421, 5211 and XS-3 are examples of self-complementing codes.
   - **Sequential:** The codes in which succeeding code word is one greater than the preceding one. 8421, XS-3 are sequential codes.
   - **Error Detecting and Correcting:** Data transmission from one place to other distant place or even within a computer, 1 or 0 may be misinterpreted as 0 or 1 due to noise. Error detection techniques allow detecting such errors, while error correction enables reconstruction of the original data in many cases. **Parity** and **Hamming** codes are used to detect and correct errors respectively.
   - **Reflective:** It is a binary code where two successive values differ by one bit only. The name "reflective" was coined by Bell Labs researcher Frank Grey since it may be built up from the conventional binary code by a sort of reflection process. The code later became popular after the name of Gray. The reflected binary code was originally designed to prevent spurious output from electromechanical switches. Today, Gray codes are widely used to facilitate error correction in digital communications such as digital terrestrial television and some cable TV systems.

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

   - **Cyclic:** The code in which each code word differs from the preceding code word in only one bit position is called cyclic code. Gray code is a cyclic code. Sometimes it is also called unit distant code.
2. **Alphanumeric:** Computers not only need to process numeric data, also they have to process letters of alphabet, special characters and symbols. Binary codes representing alphabet, numerals, special characters, and symbols are called Alphanumeric codes. Examples are ASCII, EBCDIC, Hollerith. Input output devices such as keyboards, monitors, mouse can be interfaced using these codes.
   The full form of **ASCII code** is American Standard Code for Information Interchange. It is a seven-bit code based on the English alphabet. In 1967 this code was first published and since then it is being modified and updated. ASCII code has 128 characters.
   The **EBCDIC** stands for Extended Binary Coded Decimal Interchange Code. IBM

invented this code to extend the Binary Coded Decimal which existed at that time. All the IBM computers and peripherals use this code. It is an 8-bit code and therefore can accommodate 256 characters.

Hollerith code is a code for relating alphanumeric characters to holes in a punched card. It was devised by Herman Hollerith in 1888 and enabled the letters of the alphabet and the digits 0–9 to be encoded by a combination of punchings in 12 rows of a card.

BCD Codes

The usual way of expressing a decimal number in terms of a binary number is known as pure binary coding. A number of other techniques can be used to represent a decimal number. These are:

8421 BCD Code

In the 8421 Binary Coded Decimal (BCD) representation each decimal digit is converted to its 4-bit pure binary equivalent.

For example: 57dec = 0101 0111bcd

Addition is analogous to decimal addition with normal binary addition taking place from right to left. For example,

```
  6  0110  BCD for 6        42  0100 0010  BCD for 42
 +3  0011  BCD for 3       +27  0010 0111  BCD for 27
 -------------------       ---------------------------
     1001  BCD for 9            0110 1001  BCD for 69
```

Where the result of any addition exceeds 9(1001) then six (0110) must be added to the sum to account for the six invalid BCD codes that are available with a 4-bit number. Six (0110) must also be added to the sum in case there is a carry. This is illustrated in the examples below:

```
 8     1000  BCD for 8
+7     0111  BCD for 7
-----------------------------------
       1111  exceeds 9 (1001), so
       0110  add six (0110)
...................................
0001   0101  BCD for 15


 9     1001  BCD for 9
+9     1001  BCD for 9
-----------------------------------
    1  0010  A carry is generated
       0110  Add six (0110)
...................................
0001   1000  BCD for 18
```

Note that in the last examples the 1 that carried forward from the first group of 4 bits has made a new 4-bit number and so represents the "1" in "15" or “18”.

In the examples above the BCD numbers are split at every 4-bit boundary to make reading them easier. This is not necessary when writing down a BCD number.

4221 BCD Code

The 4221 BCD code is another binary coded decimal code where each bit is weighted by 4, 2, 2 and 1 respectively. Unlike BCD coding there are no invalid representations. The decimal numbers 0 to 9 have the following 4221 equivalents

Decimal	4221	1's complement
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	1000	0111
5	0111	1000
6	1100	0011
7	1101	0010
8	1110	0001
9	1111	0000

The 1's complement of a 4221 representation is important in decimal arithmetic. In forming the code remember the following rules

- Below decimal 5, use the right-most bit representing 2 first
- Above decimal 5, use the left-most bit representing 2 first
- Decimal 5 = 2+2+1 and not 4+1

Gray Code

Gray coding is an important code and is used for its speed, it is also relatively free from errors. In pure binary coding or 8421 BCD, the counting from 7 (0111) to 8 (1000) requires 4 bits to be changed simultaneously. If this does not happen at a time then various numbers could be momentarily generated during the transition so creating spurious numbers which could be read.

Gray coding avoids this since only one bit changes between subsequent numbers. To construct the code there are two simple rules. First start with all 0s and then proceed by changing the least significant bit (lsb) which will bring about a new state.

The first 16 Gray coded numbers are indicated below.

Decimal	Binary	Gray Code	Decimal	Binary	Gray Code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Code Conversion:

Binary to Gray

$G_n = B_n$

$G_{n-1} = B_n \text{ XOR } B_{n-1}$

$G_{n-2} = B_{n-1} \text{ XOR } B_{n-2}$

$G_1 = B_2 \text{ XOR } B_1$

The conversion procedure is as follows:

- Record the MSB of Binary as MSB of Gray
- Add the MSB of the binary to the next bit of the binary, recording the sum and ignoring the carry, if any, i.e. XOR the bits. This sum is the next bit of the Gray code.
- Add the second bit of the binary to the 3rd bit of Binary, 3rd bit to the 4th bit, and so on.
- Record the successive sums as the successive bits of gray code until all the bits of binary are exhausted.

Example-16: Convert the binary 1010 to Gray code.

Binary	1	0	1	0	
Shifted binary		1	0	1	0
Gray	1	1	1	1	(Taking Sum and discarding Carry)

Gray to Binary:

$B_n = G_n$

$B_{n-1} = B_n \text{ XOR } G_{n-1}$

$B_{n-2} = B_{n-1} \text{ XOR } G_{n-2}$

$B_1 = B_2 \text{ XOR } G_1$

Conversion Procedure:

- Record the MSB of Gray as MSB of Binary
- Add the MSB of the binary to the next significant bit of Gray i.e. XOR the bits. Record the sum and ignore the carry.
- Add the second bit of the binary to the 3rd bit of Gray, 3rd bit to the 4th bit of the Gray, and so on.
- Continue this till all the Gray bits are exhausted. The sequence of bits that has been written down is the binary equivalent of the Gray code number.

Example-17: Covert the Gray code 1111 to binary.

Gray	1		1		1		1
		$\oplus$		$\oplus$		$\oplus$	
Binary	1		0		1		0

Example-18: Convert the Gray coded number 10011011 to its binary equivalent.

$B_7 = G_7 = 1$

$B_6 = B_7 \text{ XOR } G_6 = 1 \text{ XOR } 0 = 1$

$B_5 = B_6 \text{ XOR } G_5 = 1 \text{ XOR } 0 = 1$

$B_4 = B_5 \text{ XOR } G_4 = 1 \text{ XOR } 1 = 0$

$B_3 = B_4 \text{ XOR } G_3 = 0 \text{ XOR } 1 = 1$

$B_2 = B_3 \text{ XOR } G_2 = 1 \text{ XOR } 0 = 1$

$B_1 = B_2 \text{ XOR } G_1 = 1 \text{ XOR } 1 = 0$

$B_0 = B_1 \text{ XOR } G_0 = 0 \text{ XOR } 1 = 1$

So,10011011 gray = 11101101 bin

Gray coding is a non-BCD, non-weighted reflected binary code.

FURTHER READING ON CODING:

ASCII Code

The American Standard Code for Information Interchange (ASCII) is a character-encoding scheme originally based on the English alphabet that encodes 128 specified characters - the numbers 0-9, the letters a-z and A-Z, some basic punctuation symbols, some control codes that originated with Teletype machines, and a blank space - into the 7-bit binary integers.

ASCII codes represent text in computers, communications equipment, and other devices that use text. Most modern character-encoding schemes are based on ASCII, though they support many additional characters. The ASCII Table is given below.

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	

EBCDIC Codes

Extended Binary Coded Decimal Interchange Code (EBCDIC) is an 8-bit character encoding used mainly on IBM mainframe and IBM midrange computer operating systems.

Dec	Hx	Oct	Char		Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char
0	0	000	nul	(Null)	65	41	101		130	82	202	b	195	c3	303	C
1	1	001	soh	(Start of Heading)	66	42	102		131	83	203	c	196	c4	304	D
2	2	002	stx	(Start of Text)	67	43	103		132	84	204	d	197	c5	305	E
3	3	003	etx	(End of Text)	68	44	104		133	85	205	e	198	c6	306	F
4	4	004	pf	(Punch Off)	69	45	105		134	86	206	f	199	c7	307	G
5	5	005	ht	(Horizontal Tab)	70	46	106		135	87	207	g	200	c8	310	H
6	6	006	lc	(Lower Case)	71	47	107		136	88	210	h	201	c9	311	I
7	7	007	del	(Delete)	72	48	110		137	89	211	i	202	ca	312	
8	8	010	ge		73	49	111		138	8a	212		203	cb	313	
9	9	011	rlf		74	4a	112	¢	139	8b	213		204	cc	314	
10	a	012	smm	(Start of Manual Message)	75	4b	113	.	140	8c	214		205	cd	315	
11	b	013	vt	(Vertical Tab)	76	4c	114	>	141	8d	215		206	ce	316	
12	c	014	ff	(Form Feed)	77	4d	115	(	142	8e	216		207	cf	317	
13	d	015	cr	(Carriage Return)	78	4e	116	+	143	8f	217		208	d0	320	}
14	e	016	so	(Shift Out)	79	4f	117	|	144	90	220		209	d1	321	J
15	f	017	si	(Shift in)	80	50	120	&	145	91	221	j	210	d2	322	K
16	10	020	dle	(Data Link Escape)	81	51	121		146	92	222	k	211	d3	323	L
17	11	021	dc1	(Device Control 1)	82	52	122		147	93	223	l	212	d4	324	M
18	12	022	dc2	dc2 (Device Control 2)	83	53	123		148	94	224	m	213	d5	325	N
19	13	023	tm	(Tape Mark)	84	54	124		149	95	225	n	214	d6	326	O
20	14	024	res	(Restore)	85	55	125		150	96	226	o	215	d7	327	P
21	15	025	nl	(New Line)	86	56	126		151	97	227	p	216	d8	330	Q
22	16	026	bs	(Backspace)	87	57	127		152	98	230	q	217	d9	331	R
23	17	027	il	(Idle)	88	58	130		153	99	231	r	218	da	332	
24	18	030	can	(Cancel)	89	59	131		154	9a	232		219	db	333	
25	19	031	em	(End of Medium)	90	5a	132	!	155	9b	233		220	dc	334	
26	1a	032	cc	(Cursor Control)	91	5b	133	$	156	9c	234		221	dd	335	
27	1b	033	cu1	(Customer Use 1)	92	5c	134	*	157	9d	235		222	de	336	
28	1c	034	ifs	(Interchange File Separator)	93	5d	135	)	158	9e	236		223	df	337	
29	1d	035	igs	(Interchange Group Separator)	94	5e	136	;	159	9f	237		224	e0	340	\
30	1e	036	irs	(Interchange Record	95	5f	137		160	a0	240		225	e1	341	
31	1f	037	ius	(Interchange Unit Separator)	96	60	140	-	161	a1	241	~	226	e2	342	S
32	20	040	ds	(Digit Select)	97	61	141	/	162	a2	242	s	227	e3	343	T
33	21	041	sos	(Start of Significance)	98	62	142		163	a3	243	t	228	e4	344	U
34	22	042	fs	(Field Separator)	99	63	143		164	a4	244	u	229	e5	345	V
35	23	043			100	64	144		165	a5	245	v	230	e6	346	W
36	24	044	byp	(Bypass)	101	65	145		166	a6	246	w	231	e7	347	X
37	25	045	lf	(Line Feed)	102	66	146		167	a7	247	x	232	e8	350	Y
38	26	046	etb	(End of Transmission Block)	103	67	147		168	a8	250	y	233	e9	351	Z
39	27	047	esc	(Escape)	104	68	150		169	a9	251	z	234	ea	352	
40	28	050			105	69	151		170	aa	252		235	eb	353	
41	29	051			106	6a	152	|	171	ab	253		236	ec	354	
42	2a	052	sm	(Set Mode)	107	6b	153	,	172	ac	254		237	ed	355	
43	2b	053	cu2	(Customer Use 2)	108	6c	154	%	173	ad	255		238	ee	356	
44	2c	054			109	6d	155		174	ae	256		239	eF	357	
45	2d	055	enq	(Enquiry)	110	6e	156	<	175	af	257		240	f0	360	0
46	2e	056	ack	(Acknowledge)	111	6f	157	?	176	b0	260		241	f1	361	1
47	2f	057	bel	(Bell)	112	70	160		177	b1	261		242	f2	362	2
48	30	060			113	71	161		178	b2	262		243	f3	363	3
49	31	061			114	72	162		179	b3	263		244	f4	364	4
50	32	062	syn	(Synchronous Idle)	115	73	163		180	b4	264		245	f5	365	5
51	33	063			116	74	164		181	b5	265		246	f6	366	6
52	34	064	pn	(Punch On)	117	75	165		182	b6	266		247	f7	367	7
53	35	065	rs	(Reader Stop)	118	76	166		183	b7	267		248	f8	370	8
54	36	066	uc	(Upper Case)	119	77	167		184	b8	270		249	f9	371	9
55	37	067	eot	(End of Transmission)	120	78	170		185	b9	271		250	fa	372	|
56	38	070			121	79	171	`	186	ba	272		251	fb	373	
57	39	071			122	7a	172	:	187	bb	273		252	fc	374	
58	3a	072			123	7b	173	#	188	bc	274		253	fd	375	
59	3b	073	cu3	(Customer Use 3)	124	7c	174	@	189	bd	275		254	fe	376	
60	3c	074	dc4	(Device Control 4)	125	7d	175	'	190	be	276		255	ff	377	eo
61	3d	075	nak	(Negative Acknowledge)	126	7e	176	=	191	bf	277					
62	3e	076			127	7f	177	"	192	c0	300	{				
63	3f	077	sub	(Substitute)	128	80	200		193	c1	301	A				
64	40	100	Sp	(Space)	129	81	201	a	194	c2	302	B				

Source: www.pubblinet.com

Additional Reading:

Floating Point data:

- It is similar to scientific notation in base 10
- The floating point format is suitable to express large numbers
- It is used to store mixed as well as integer data
- Floating point numbers are often stored in four bytes
- Format of 4-byte (single precision) floating point number is:

31	30 ... 23	22 ... 0
S	Exponent	Mantissa
	8 bits	23 bits

- The left most bit indicates the sign of the mantissa
- Next 8 bits are for exponent stored in excess 127 notation
- Exponent in excess 127 is an unsigned integer that is equal to the actual exponent plus 127
- Mantissa is a normalized 23-bit number with a hidden or implied 1 in 24th bit position

Examples:

$100_{10} = 1100100_{2} = 1.1001 \times 2^{6}$

S	Exponent	Mantissa
0	1000 0101	10010000000000000000000

-12.7010 = -1100.112 = 1.10011 x 23

S	Exponent	Mantissa
1	1000 0010	10011000000000000000000

The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point; that is, the decimal point can float. There are also representations in which the number of digits before and after the decimal point is set, called fixed-point representations. In general, floating-point representations are slower and less accurate than fixed-point representations, but they can handle a larger range of numbers.

Note that most floating-point numbers a computer can represent are just approximations. One of the challenges in programming with floating-point values is ensuring that the approximations lead to reasonable results. If the programmer is not careful, small discrepancies in the approximations can snowball to the point where the final results become meaningless.

Because mathematics with floating-point numbers requires a great deal of computing power, many microprocessors come with a chip, called a floating point unit (FPU), specialized for performing floating-point arithmetic. FPUs are also called math coprocessors and numeric coprocessors.

College of Engineering & Management, Kolaghat

Sample Questions and answers

1. **Convert the following numbers in 8-bit binary codes:**
   (a) 127_{10}, (b) 63_8, (c) 85.95_{10} (up to 4 bit fraction).

   a) 127_{10} = 64+32+16+8+4+2+1 = 0111 1111_2
   b) 63_8 = 110 011_2 (just converting each octal digit to its equivalent binary code).
   c) 85.95_{10}= (64+16+4+1)+(0.5+0.25+0.125+0.0625)= 0101 0101.1111

2. **Express the following Decimal Numbers in 8-bit Binary code:**
   (a) -120 – in signed magnitude, (b) -108 in 1's complement, (c) -125 – in 2's complement.
   a) -120_{10} = -(64+32+16+8)= 1111 1000_2 – in signed magnitude
   b) -108_{10} = 1's complement of +108 (0110 1100) = 1001 0011
   c) -125_{10} = 2's complement of +125 (0111 1101) = 1000 0011

3. **Express the following decimal number in BCD codes:**
   (a) 79, (b) 65, (c) 63;   a) & b) in 8421 BCD and c) in 4221 BCD codes
   a) 79_{10} = 0111 1001 – in 8421 BCD
   b) 65_{10} = 0110 0101 – in 8421 BCD
   c) 63_{10} = 1100 0011 – in 4221 BCD

4. **Express the following signed binary numbers (2's complement) to equivalent Decimal Numbers:**
   (a) 1011 1111, (b) 1010 1010, (c) 0111 1111
   a) 1011 1111, its msb being 1, it is −ve and its magnitude can be found by taking its 2's complement. 1011 1111 => 2's complement is 0100 0001 => 65. So the number is -65_{10}
   b) 1010 1010, msb being 1, it is −ve and its magnitude can be found by taking its 2's complement. 1010 1010 => 2's complement is 0101 0110 => -86_{10}
   c) 0111 1111, msb being 0, it is +ve and its magnitude is 127. So the number is $+127_{10}$.

5. **Convert the following numbers into Gray Codes:**
   (a) 45_8, (b) 97_{10}, (c) 1010 1010_2
   a) 45_8 = 100 101_2

```
Binary              100101
Shifted Binary       100101
--------------------------------
Gray Code           110111
```

b) 97_{10} = 110 0001_2

```
Binary                1100001
Shifted Binary         1100001
-------------------------------
Gray                  1010001
```

c)

```
Binary                10101010
Shifted Binary         10101010
----------------------------------
Gray                  11111111
```

6. Do the following operations:

(a) (-100-15) in 1's complement method

-100 = 1's complement of +100
+100 = 0110 0100
-100 = 1001 1011

+15 = 0000 1111
-15 = 1111 0000

```
-100 =    1001 1011
-15  =    1111 0000
----------------------
-115 =  1 1000 1011
```

adding end around carry => 1000 1100, msb being 1, answer is -ve and the magnitude can be obtained by taking 1's complement which gives, 0111 0011 => 115. So the result is -115 which matches with the decimal addition.

(b) (-128+127) in 2's complement method

```
-128 = 1000 0000
+127 = 0111 1111
----------------------
```

-1 = 1111 1111, msb being 1, answer is -ve and the magnitude can be obtained by taking 2's complement which gives, 0000 0001 => 1. So the result is -1 which matches with the decimal addition.

Boolean Algebra:
Various Logic Gates – their truth tables and circuits

INTRODUCTION

We have learned that binary variables can assume only two values, and these are called logic values; logic 1 and logic 0. They may also be denoted as ON/OFF, HIGH/LOW, YES/NO, TRUE/FALSE. Logical operators operate on binary values and binary variables. Variable identifiers may be A, B, C, x, y, z, RESET, START etc.

Basic **logical operators** are logic functions OR, AND and NOT. Logic gates are electronic circuits implementing logic functions. Mathematical system developed for specifying and transforming logic functions is called ***Boolean Algebra***. We study Boolean algebra as a foundation for designing and analyzing digital systems.

AND is denoted as a dot (.), OR is denoted by a plus (+) and NOT is denoted by an over bar (¯), a single quote mark (') after, or (~) before the variable. For Example

Y = A.B is read as "Y is equal to A AND B"

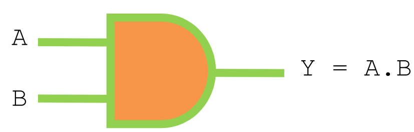
z = x + y is read as "z is equal to x OR y" and

X = ~A is read as "X is equal to NOT A"

Behaviors of logic functions can be defined in words or in a table. Such a table lists all possible combinations of input variables and corresponding outputs and is called a ***truth table***.

AND GATE

An AND gate has two or more inputs and only one output. The output assumes logic 1 when all the inputs assume logic 1 and assumes logic 0 when any one of its inputs assumes logic 1. Therefore, the AND gate may be defined as a device whose output is 1 if and only if all its inputs are 1. Thus, an AND gate is also called all or nothing gate. The logic symbol and the truth table for a two-input AND gate is shown in Fig.1.



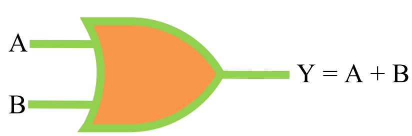
Inputs		Output
A	B	Y=A.B
0	0	0
0	1	0
1	0	0
1	1	1

Logic symbol (b) Truth Table

Fig.1: A two-input AND Gate

OR GATE

Like an AND gate, an OR gate may also have two or more inputs and only one output. The output assumes logic 1 when any of its inputs assumes logic 1 and assumes logic 0 when all the inputs assume logic 0. Therefore, an OR may be defined as a device whose output is 1, even if one of its inputs is 1. Hence an OR gate is called any or all gates. The logic symbol and the truth table of a two-input OR gate is shown in Fig.2.



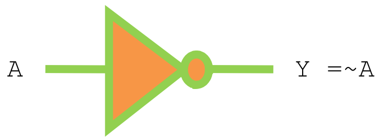
Inputs		Output
A	B	Y=A+B
0	0	0
0	1	1
1	0	1
1	1	1

(a) Logic symbol (b) Truth Table

Fig.2: A two-input OR Gate

NOT GATE (INVERTER)

A NOT gate, also called an inverter, has only one input and one output. It is a device whose output is always the complement of the input. Therefore, the output of a NOT gate assumes a logic 1 if its input is logic 0 and vice-versa. The logic symbol and the truth table of a NOT gate is shown in Fig.3.



Input	Output
A	Y=~A
0	1
1	0

(a) Logic symbol (b) Truth Table

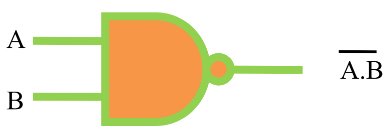
Fig.3: A NOT Gate

UNIVERSAL GATES

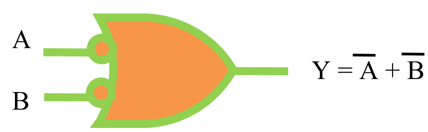
Though logic circuits of any complexity can be realized by using the three basic gates AND, OR and NOT, there are two universal gates NAND and NOR. Any logic circuit can be implemented by either NAND or NOR gates.

NAND GATE

A NAND gate is the combination of AND followed by NOT. The symbol and the truth table of a NAND gate are shown in Fig.4.



Inputs		Output
A	B	Y=(A.B)'
0	**0**	**1**
0	1	1
1	0	1
1	1	0



(a) Logic symbol (b) Truth Tale (c) Equivalent representation

Fig.4: A two-input NAND Gate

NAND gate as a bubble OR gate

Truth table of the two-input NAND gate reveals that the output is 1 if either of the inputs or both are 0. Or in other words, output is 1 when either A'=1 or B'=1 or both A' and B' are 1 which is the characteristics of an OR gate. *Thus, a NAND function in positive logic is equivalent to an OR function in negative logic and a NAND gate is called a bulled OR gate or a negative OR gate.* This can be expressed mathematically as,

$$Y = \overline{A.B} = \overline{A} + \overline{B}$$

NAND gate as an OR gate

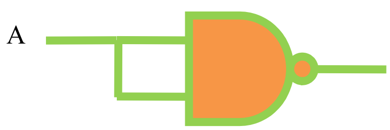
If we apply bubble inputs to a NAND gate, the expression of the output of the NAND becomes,

$$Y = \overline{\overline{A}.\overline{B}} = \overline{\overline{A}} + \overline{\overline{B}} = A + B$$

Therefore, bubbled NAND gate is equivalent to an OR gate.

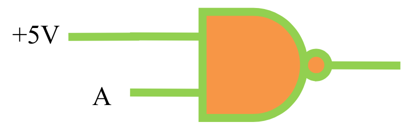
NAND gate as inverter

A NAND gate can be used as an inverter by shorting its inputs and applying the input signal to the common terminal or by connecting all the inputs except one to the logic 1 and applying the signal to it as shown in Fig.5.



$Y = \overline{A.A} = \overline{A}$

(a) Signal is applied to the input pins shorted together



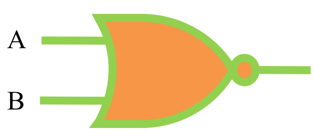
$Y = \overline{A.1} = \overline{A}$

(b) Signal is applied to one input terminal and others are shorted and connected to logic 1

Fig.5: Realization of Logic NOT by using NAND gate

NOR GATE

A NOR gate is the combination of an OR followed by a NOT gate. The symbol and the truth table of a NAND gate is shown in Fig.6.

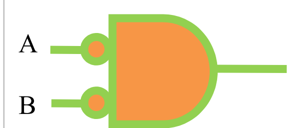


$\overline{A+B}$

(a) Logic symbol

Inputs		Output
A	B	Y=(A+B)′
0	0	1
0	1	0
1	0	0
1	1	0

(b) Truth Tale



$Y = \overline{A} . \overline{B}$

(c) Equivalent representation

Fig.6: A two-input NOR Gate

NOR gate as a bubble AND gate

Truth table of the two-input NOR gate reveals that the output is 1 if and only if both the inputs are 0 which is the characteristics of an AND gate. *Thus a NOR function in positive logic is equivalent to an AND function in negative logic and thus a NOR gate can be called a bulled AND gate or a negative AND gate*. This can be expressed mathematically as,

$$Y = \overline{A+B} = \overline{A} . \overline{B}$$

NOR gate as an AND gate

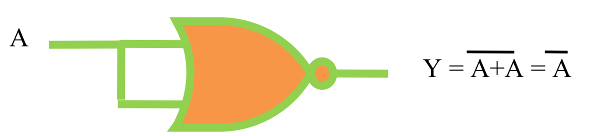
If we apply bubble inputs to a NOR gate, the expression of the output of the NOR becomes,

$$Y = \overline{\overline{A}+\overline{B}} = \overline{\overline{A}} . \overline{\overline{B}} = A . B$$

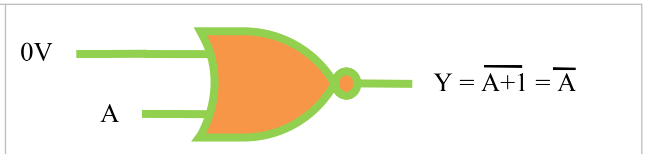
Therefore, a bubbled NOR gate is equivalent to an AND gate.

NOR gate as inverter

A NOR gate can also be used as an inverter like a NAND gate by shorting its inputs and applying the input signal to the common terminal or by connecting all the inputs except one to the logic 0 and applying the signal to it as shown in Fig.7.



(a) Signal is applied to the input pins shorted together

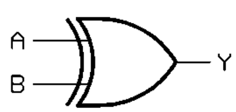


(b) Signal is applied to one input terminal and others are shorted and connected to logic 0

Fig.7: Realization of Logic NOT by using NOR gate

XOR Gate

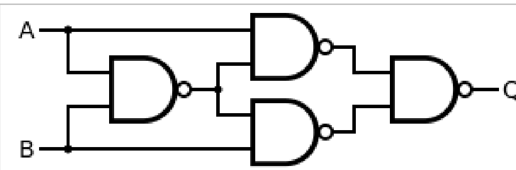
An Exclusive OR (XOR) gate has two inputs and one output. The output assumes logic 1 if one and only one of the inputs assumes logic 1. If both the inputs are either 0 or 1, the output is 0. A way to remember the XOR gate is "one or the other but not both". XOR represents the inequality function, i.e., the output is HIGH (1) if the inputs are not alike otherwise the output is LOW (0). XOR can also be viewed as addition modulo 2. As a result, XOR gates are used to implement binary addition in computers. A half adder consists of a XOR gate and an AND gate.



(a) Logic symbol

Inputs		Output
A	B	$Y=(A \oplus B)$
0	0	0
0	1	1
1	0	1
1	1	0

(b) Truth Tale

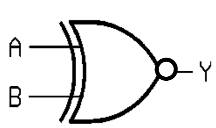


(c) Implementation by NAND Gate only

Fig.8: A two-input XOR Gate

XNOR Gate

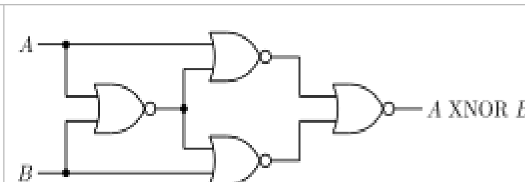
The **XNOR gate** is a digital logic gate whose function is the inverse of the XOR gate. The two-input version implements logical equality, behaving according to the truth table. A HIGH output (1) results if both inputs to the gate are the same. If one but not both inputs are HIGH (1), a LOW output (0) results.



(a) Logic symbol

Inputs		Output
A	B	$Y=(A \odot B)$
0	0	1
0	1	0
1	0	0
1	1	1

(b) Truth Tale



(c) Implementation by NOR Gate only

Fig.8: A two-input XNOR Gate

Important

XOR is called an odd function as its output assumes logic 1, if odd numbers of inputs assume logic 1 and XNOR is called an even function as its output assumes logic 1 when even numbers of inputs assume logic 1.

BOOLEAN ALGEBRA

Axioms and Laws of Boolean Algebra

Axioms or postulates of Boolean Algebra are a set of logical expressions that we accept without proof and based on these postulates useful theorems are built. Actually, the axioms are nothing but the definitions of three basic logic operations AND, OR and NOT. Each axiom is actually the outcome of an operation performed by a logic gate.

AND Operation

Axiom 1: 0.0 = 0

Axiom 2: 0.1 = 0

Axiom 3: 1.0 = 0

Axiom 4: 1.1 = 1

OR Operation

Axiom 5: 0 + 0 = 0

Axiom 6: 0 + 1 = 1

Axiom 7: 1 + 0 = 1

Axiom 8: 1 + 1 = 1

NOT operation

Axiom 9: 1′ = 0

Axiom 10: 0′ = 1

Complementation Laws:
If A = 0, then A′ = 1
If A = 1, then A′ = 0
(A′)′ = A

AND Laws:
A . 0 = 0
A . 1 = A
A . A = A
A . A′ = 0

OR Laws:
A + 0 = A
A + 1 = 1
A + A = A
A + A′ =1

Commutative Laws:
A + B = B + A
A . B = B . A

Associative Laws:
(A + B) + C = A + (B + C)
(A . B) . C = A . (B . C)

Distributive Laws:
A (B + C) = AB + AC
A + BC = (A + B)(A + C)
Proof: RHS = (A+B)(A+C) = AA+AC+BA+BC = A+AC+BA+BC = A(1+C+B)+BC = A+BC = LHS

Redundant Literal Rule (RLR):
A + A′B = A + B
Proof: LHS = A+A′B = (A+A′)(A+B), using distributive Law
= 1.(A+B) = A+B = RHS
A (A′ + B) = AB
Proof: LHS = A(A′+B) = AA′+AB = AB = RHS

Absorption Laws:
A+AB = A
A(A+B) = A

Consensus Theorem (Included Factor Theorem):
1. AB + A′C + BC = AB + A′C
Proof:
LHS = AB+A′C+BC = AB+A′C+BC(A+A′) = AB+A′C+ABC+A′BC = AB(1+C)+A′C(1+B)
= AB + A′C = RHS

This theorem can be extended to any number of variables. Like
AB + A'C + BCD = AB + A'C
LHS = AB + A'C + BCD = AB + A'C + BC + BCD = AB + A'C + BC(1+D) = AB + A'C + BC = AB + A'C = RHS.

2. (A+B)(A'+C)(B+C) = (A+B)(A'+C)
Proof:
LHS = (A+B)(A'+C)(B+C) = (AA'+AC+BA'+BC)(B+C) = ACB+ACC+BA'B+BA'C+BBC+BCC = ACB+AC+BA'+BA'C+BC = AC+BA'+BC
RHS = (A+B)(A'+C) = AA'+AC+BA'+BC = AC+BA'+BC = LHS

Transposition Theorem:
AB + A'C = (A+C)(A'+B)
Proof: RHS = (A+C)(A'+B) = AA'+AB+A'C+BC = 0+AB+A'C+BC(A+A')
= AB+A'C+ABC+A'BC = AB(1+C)+A'C(1+B) = AB + A'C = LHS

De Morgan's Theorem
Law 1: (A + B)' = A'.B'
Law 2: (AB)' = A' + B'

Duality Principle:

An OR logic in +ve logic becomes an AND logic in –ve logic and vice-versa. Positive and negative logics thus give rise to a basic duality in all Boolean identities. Also changing from one logic to another, 0 becomes 1 and 1 becomes 0. Thus from a Boolean identity, we can produce the dual identity by changing all '+' signs to '.' signs, all '.' signs to '+' signs and complementing all 0's and all 1's but the variables are not complemented.

Relation between complement and dual
Let us consider a function f as:
f(A, B,C,…….)
Then the complement of the function may be written as:

$$fc(A,\ B,\ C\ldots\ldots) = \overline{f(A,\ B,\ C,\ldots\ldots)} = fd\ (A',\ B',\ C'\ \ldots\ldots)$$

The dual of the function f can be defined as:

$$fd(A,\ B,\ C,\ldots\ldots) = \overline{f(A',\ B',\ C'\ldots\ldots)} = fc\ (A',\ B',\ C'\ \ldots\ldots)$$

The first relation states that the complement of a function can be obtained by complementing all the variables in the dual function. Similarly the second relation states that the dual can be obtained by complementing all the literals in the function. Some dual identities are given below:

Expression	**Dual**
0' = 1	1' = 0
0 . 1 = 0	1 + 0 = 1
0 . 0 = 0	1 + 1 = 1
A . 0 = 0	A + 1 = 1
A . 1 = A	A + 0 = A
A . A = A	A + A = A
A . A' = 0	A + A' = 1
A . B = B . A	A + B = B + A
A . (B . C) = (A .B) . C	A + (B + C) = (A + B) + C
A . (B + C) = AB + AC	A + B.C = (A + B) (A +C)
A . (A+B) = A	A + AB = A
(AB)' = A' + B'	(A + B)' = A'B'

Representation in SOP and POS forms

Any Boolean expression can be simplified in many different ways resulting in different forms of the same Boolean function. All Boolean expressions, regardless of their forms, can be converted into one of two standard forms; the sum-of-product form and the product-of sum forms. Standardization makes the evaluation, simplification and implementation of Boolean expression more systematic and easier.

The Sum of Product (SOP) Form

Writing functions in SOP form means that the inputs of each term are multiplied using AND function, then all terms are added together using OR function. The variables in each term are not necessarily all the variables of the function. For example, a SOP of F(A,B,C) may contain a term that contains only the variable A but not B nor C, in such case the term is not in its standard SOP form. Standard SOP term must contain all the variables. Boolean algebra states that X+X' =1, then any nonstandard product term may be multiplied X+X' for any missing literal X without affecting its value.

Example:

The following function is written in the SOP form:

F(A,B,C)=A+BC'+A'BC

The inputs to the function F are A, B and C. In each term the inputs are ANDed then all terms are ORed to form the function F. Note that the last term A'BC contains all the inputs of the function (A, B and C), so, this term is written in standard form. But the second term BC' is not in standard form because the input A does not exist, then multiply it by (A'+A). The same is done for the remaining term as follows:

F(A,B,C)=A(B+B')(C+C')+BC'(A+A')+A'BC = ABC+ABC'+AB'C+AB'C'+ABC'+A'BC'+A'BC

From Boolean algebra, (A+A=A) then all similar terms in the equation will be reduced to one term. Now the function F becomes

F(A,B,C)=ABC+ABC'+AB'C+AB'C'+A'BC'+A'BC

The Product of Sum Form (POS)

Writing functions in POS form means that the inputs of each term are Added together using OR function and then all terms are multiplied together using AND function. The variables in each term are not necessarily all the variables of the function. For example, a POS of F(A,B,C) may contain a term that contains only the variable A but not B nor C, in such case the term is not in its standard POS form. Standard POS term must contain all the function variables. Boolean algebra states that X.X'=0, then if the term is added to (X.X'), it becomes in the standard POS form, but its value is not affected.

Example

The following function is written in the POS form:
F(A,B,C)=A.(B+C').(A'+B+C')

The inputs to the function F are A, B and C. In each term the inputs are ORed then all terms are ANDed to form the function F. Note that the last term (A'+B+C') contains all the inputs of the function (A, B and C), so, this term is written in standard form. But the second term (B+C') is not in standard form because the input A does not exist, then add (A'.A). The same is done for the remaining term as follows:

F(A,B,C)=[A+(B.B')+(C.C')].[(B+C')+(A.A')].(A'+B+C')
F(A,B,C)=[(A+B+C).(A+B+C').(A+B'+C).(A+B'+C')].[(A+B+C').(A'+B+C')].(A'+B+C')

From Boolean algebra, (A.A=A) then all similar terms in the equation will be reduced to one term. Now the function F becomes

F(A,B,C)=(A+B+C).(A+B+C').(A+B'+C).(A+B'+C').(A'+B+C')

Minterms

Writing a function in its minterm format is equivalent to writing the function in its standard SOP format such that the value of the function at these terms is 1. So that if we have the truth table relating the input variables to the function F, then we can determine which cases result in F=1 and write the minterm form of the function.

Maxterms

Writing a function in its maxterm format is equivalent to writing the function in its standard POS format such that the value of the function at these terms is 0. So that if we have the truth table relating the input variables to the function F, we can determine which cases result in F=0 and write the maxterm form of the function.

Minimization of logic expressions by algebraic method

Every Boolean expression must be reduced to as simple form as possible before implementation to reduce hardware cost as well as to reduce time delay. Laws of Boolean algebra must be utilized while reducing a Boolean expression. The following steps are useful for reduction of expression.

a) Remove parenthesis
b) Drop all the identical terms except one. For example, AB+AB+AB+AB = AB
c) A term containing a literal and its complement may be dropped. For example, A.B.B′ = A.0 = 0
d) Two terms those are identical except for one variable which is missing in one term, the larger term may be dropped. For example, ABC′D + ABC′ = ABC′ (D+1) = ABC′
e) Two terms having some literals those are present in true form in one and in complemented form in the other, they can be combined to a single term by dropping those literals. For example, ABCD + ABCD′ = ABC (D+D′) = ABC.1 = ABC.

EXAMPLES:

AB′ + A(B + C)′ + B(B + C)′
= AB′+AB′C′+BB′C′
= AB′

[AB′(C + BD) + A′B′]C
= [AB′C+AB′BD+A′B′]C
= AB′CC+A′B′C
= AB′C+A′B′C
= B′C

A′BC + AB′C′ + A′B′C′ + AB′C + ABC
= A′BC + ABC +AB′C′+A′B′C′+AB′C
= BC+B′C′+AB′C
= C(B+AB′)+B′C′
= C[(B+A)(B+B′)]+B′C′
= C(A+B)+B′C′
= AC + BC + B′C′

XYZ+YYZ+Y′Z+XY′
= XYZ+YZ+Y′Z+XY′
= YZ+Y′Z+XY′
= Z+XY′

[(AB′C)(AB′)′+BC]′
= [(AB′C)(A′+B)+BC]′
= [AB′CA′+AB′CB+BC]′
= [BC]′

[(ABC+A′B′)′+BC]′
= [(ABC)′(A′B′)′+BC]′
= [(A′+B′+C′)(A+B)+BC]′
= [A′A+A′B+B′A+B′B+C′A+C′B+BC]′
= [A′B+AB′+AC′+BC′+BC]′
= [A′B+AB′+B]′
= [B+AB′]′
= B′.(AB′)′
= B′.(A′+B)
= A′B′

Venn Diagram
Venn diagram is a graphical method to illustrate the relationships among the variables of a Boolean expression. The diagram consists of a rectangle and a few overlapping circles inside. Each circle represents a binary variable. A variable assumes logic 1 inside the circle and logic 0 outside it as shown in the Fig.1 below.

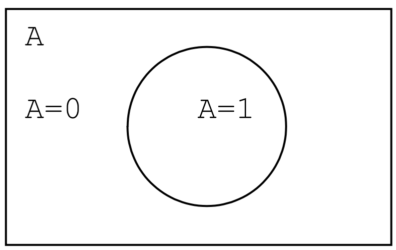


Fig.1: Venn Diagram representing the binary variable A

A two variable Venn Diagram is shown in Fig.2 which has two overlapping circles indicating two binary variables A and B.

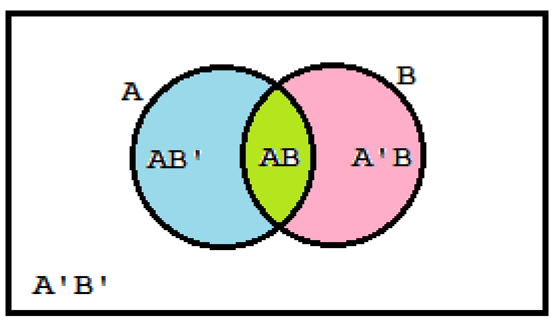


Fig.2: Venn Diagram for two variables A, B

The figure has four distinct areas inside the rectangle:
- the area not belonging to either A or B (A′B′),
- the area inside the circle A but outside B (AB′),
- the area inside B but outside A (A′B) and
- the area in both the circles (AB).

Venn Diagram may be used to illustrate the postulates of Boolean Algebra or to show the validity of the theorems. Fig.3-4 are a few such illustrations.

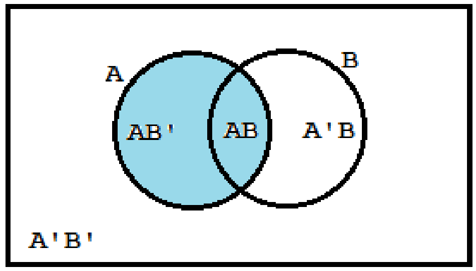
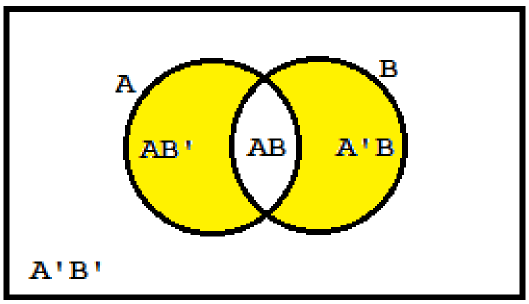
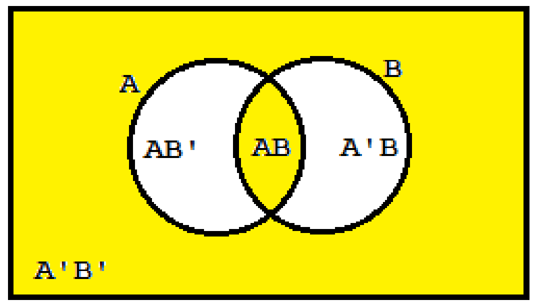


Fig.3: Venn Diagram showing the validity of the following relations:
(i) A=A+AB, (ii) A=AB'+AB, (iii) A=(A'B'+A'B)'



XOR Function, F=AB'+A'B=(A'B'+AB)'



XNOR Function, F=A'B'+AB=(AB'+A'B)'

Fig.4: Venn Diagram showing XOR and XNOR function

Three variable Venn Diagram is shown in Fig.5. There are eight distinct areas which are marked by the terms they represent.

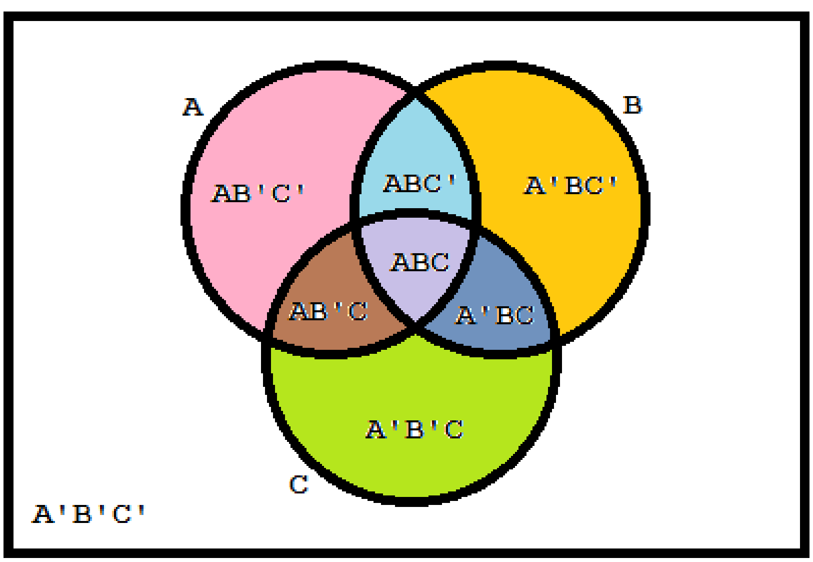


Fig.5: Venn Diagram showing relation of three variables

Minimization of logic expressions by Karnaugh Map Method

A Karnaugh Map or K-Map is a set of m-squares indicating minterms or maxterms of a Boolean function of n-variables such that $2^n = m$. For example, a function of 4 variables will have $2^4=16$ cells as shown below.

CD / AB	00	01	11	10	Minterms can be found by ANDing variables in either true (for 1) or complemented (for 0) form corresponding to the cell.
00	m_0	m_1	m_3	m_2	$m_0 = A'B'C'D'$, $m_1 = A'B'C'D$, ….., $m_{14} = ABCD'$ and $m_{15} = ABCD$
01	m_4	m_5	m_7	m_6	
11	m_{12}	m_{13}	m_{15}	m_{14}	
10	m_8	m_9	m_{11}	m_{10}	

CD / AB	00	01	11	10	Maxterms can be found by ORing complemented or un-complemented variables for 1 or 0 of the cell respectively
00	M_0	M_1	M_3	M_2	$M_0 = A+B+C+D$, $M_1 = A+B+C+D'$, ….., $M_{14} = A'+B'+C'+D$ and $M_{15} = A'+B'+C'+D'$
01	M_4	M_5	M_7	M_6	
11	M_{12}	M_{13}	M_{15}	M_{14}	
10	M_8	M_9	M_{11}	M_{10}	

Minterms contained in the function contribute 1's to the function and are plotted as 1's to the corresponding cells. Maxterms contribute 0's to the cell. Each of these leads to the solution for the function (F). Minterms contributing 0's to the function (not contained in the expression of the function) can be plotted as 0's to the corresponding cells and can be solved to get the complement of the function (F').

Relation between the Minterm and the Maxterm representing to the same cell of the Karnaugh Map

Let us consider a minterm m_0 for a three variable function, F(A, B, C). Therefore,

$m_0 = A'B'C'$

Now, $(m_0)' = (A', B', C')' = A+B+C = M_0$

Therefore, complement of a minterm gives the corresponding maxterm and vice-versa.

Thus for a function,

$F(A,B,C) = m_0 + m_2 + m_4 + m_7 = \sum m(0, 2, 4, 7) = \sum(0, 2, 4, 7)$

$F'(A,B,C) = (m_0 + m_2 + m_4 + m_7)' = m_0' m_2' m_4' m_7' = M_0.M_2.M_4.M_7 = \prod M(0, 2, 4, 7) = \prod (0, 2, 4, 7)$

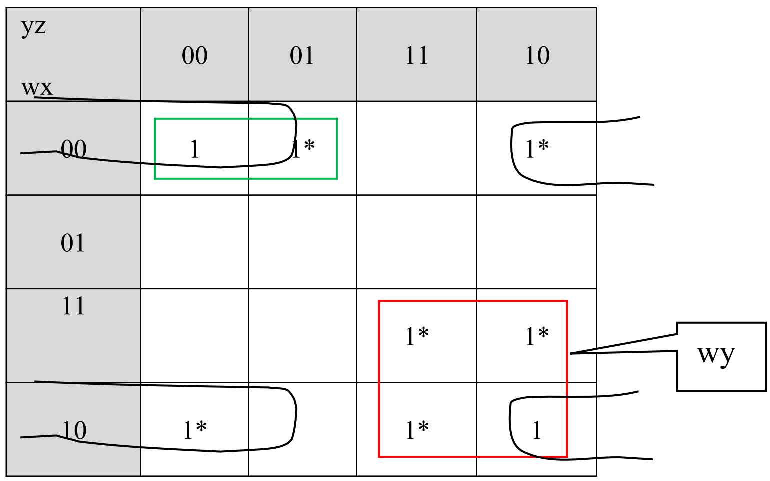
The complement of the function can also be written by including the minterms contributing 0 to the function. Therefore, the minterms not included in F gives the complement of F as:

$F'(A,B,C) = m_1 + m_3 + m_5 + m_6 = \sum m(1, 3, 5, 6) = \sum(1, 3, 5, 6)$

Note: If the number of minterms contributing 1 to the function is more than half of the total number of minterms then it will be wise to solve the complement of the function.

Cell Adjacency:

The cells in a Karnaugh map are arranged so that only a single variable changes between adjacent cells. Adjacency is defined by a single variable change. In the 3-variable map the 010 cell is adjacent to the 000 cell, the 011 cell, and the 110 cell. The 010 cell is not adjacent to the 001 cell, the 111 cell, the 100 cell, or the 101 cell. A number of cells (2^n in numbers where n=1, 2, 3 ………..) may also be adjacent provided n variables complement (change) amongst the cells. For example, four cells are said to be adjacent if only 2 variables complement, 8 cells are said to be adjacent if only 3 variables complement and so on. The following 4 variable K-map shows different adjacencies.



Adjacent cells in K-map can be combined to drop the variables that complement to get a term having less number of literals following the Boolean principle, A+A' = 1 and thus ABC + A'BC = (A+A')BC = BC or ABC'+A'B'C' = C'. Terms representing a group of adjacent cells are called ***Prime Implicants***(PI). Thus, a Prime Implicant is a reduced product term. For example, AB, A, BC are prime implicants of a Boolean function F(A,B,C). Different groups of adjacent cells may include a particular cell as a part of each such group. A prime implicant having at least one cell that is not combined by other prime implicants is called an ***Essential Prime Implicant*** (EPI). These 1-Cells covered by one group only are called ***Distinguished 1-Cells*** (Stared Cells).

KARNAUGH MAP MINIMIZATION

For an SOP expression in standard form, a 1 is placed in the Karnaugh map for each product term in the expression. Each 1 is placed in a cell corresponding to the product term. For example, for the product term AB'C, a 1 goes in the 101 cell on a 3-variable map.

Example:

Plot the following minterms in K-map and combine the adjacent cells and write the minimal function.

F(A,B,C) = A’B’C’+A’B’C+ AB’C’+AB’C+ABC’+ABC

= ∑(0, 1, 4, 5, 6, 7)

BC / A	00	01	11	10
0	1	1		
1	1	1	1	1

B’

A

Thus, the minimal function is:

F(A,B,C) = A + B’

Let us solve the problem by POS Minimization. For a POS in standard form, a 0 is placed in the Karnaugh map for each sum term in the expression. The complement of the function may be written in terms of Maxterms as:

F’(A,B,C) = ∏($M_0M_1M_4M_5M_6M_7$)

BC / A	00	01	11	10
0	0	0		
1	0	0	0	0

B

A’

Thus the simplified complement function may be written as:

F’(A,B,C) = BA’

And, F(A,B,C) = (F’)’ = (BA’)’ = A + B’

The complement of the same function can be solved by the missing minters in the expression of F as follows:

F(A,B,C) = ∑(0, 1, 4, 5, 6, 7)

So, F’(A.B.C) = ∑(2, 3)

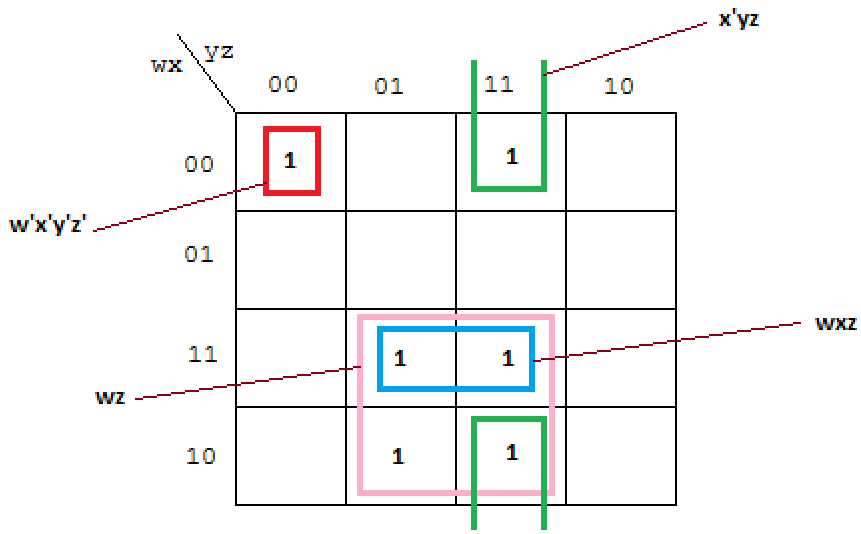
BC / A	00	01	11	10
0			0	0
1				

A’B

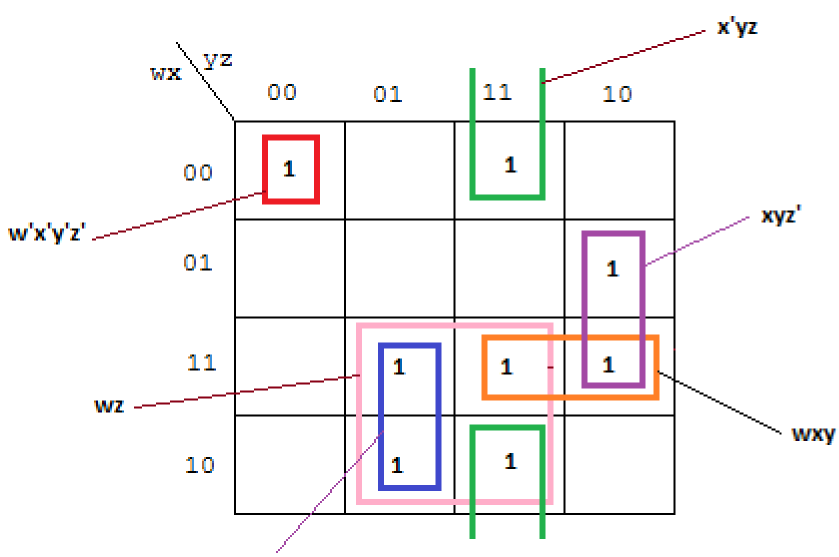
Thus, F’ = A’B, and F = (F’)’ = (A’B)’ = A + B’

Some Definitions

IMPLICANT:
Any single 1 or group of 1's that can be combined together on a Karnaugh map of the function F is called an **IMPLICANT.** So any possible grouping of 1's is an implicant.



PRIME IMPLICANT:
A **PRIME IMPLICANT** is a product term that cannot be combined with another term to eliminate a variable. A group contained in a large group is not a prime implicant.



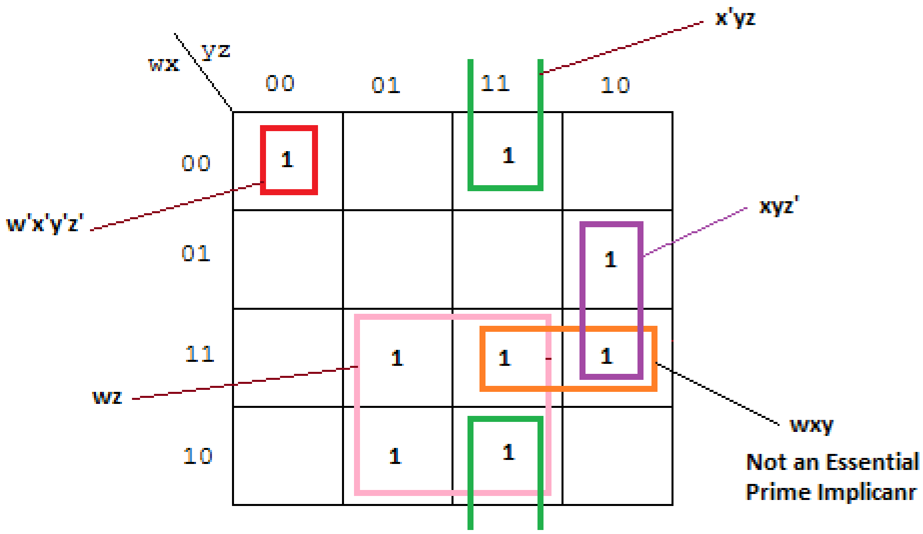
A single 1 is a prime implicant, if it is not adjacent to any other 1's.Two adjacent 1's form a prime implicant, if they are not contained in a group of four adjacent 1's.

Four adjacent 1's form a prime implicant, if they are not contained in a group of eight adjacent 1's.

The minimum sum of products (SOP) expression for a function consists of some (BUT NOT NECESSARILY ALL) of the prime implicants of a function.

If a SOP expression contains a term which is not a prime implicant, then it cannot be a minimum expression.

ESSENTIAL PRIME IMPLICANT:
A prime implicant is ESSENTIAL if it contains a minterm which is not covered by any other Prime Implicant.



K-map rules in a nutshell:

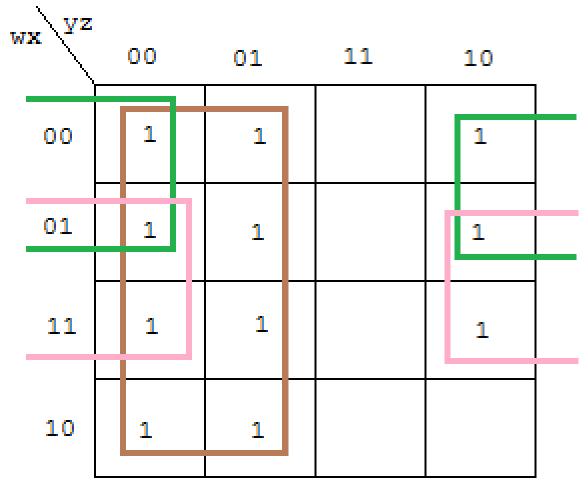
- Groups of 1's do not include any cell containing 0 and vice-versa
- Groups are either horizontal or vertical. Diagonal grouping is not allowed
- Groups must contain 1, 2, 4, 8, 16 or in general 2^n adjacent cells
- A group should be as large as possible
- Each cell containing a 1 must be included in at least one group
- Groups may overlap
- Groups may wrap around the map. The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell
- There should be as few groups as possible
- Groups are to be Essential Prime Implicants for minimal solution.

Example:
Simplify the Boolean function

$F(w,x,y,z)=\sum m(0,1,2,4,5,6,8,9,12,13,14)$

Solution:

- Since the function has 4 variables, the K-map has 16 cells as shown in figure below
- Minterms are marked with 1's in the corresponding cells in the K-map

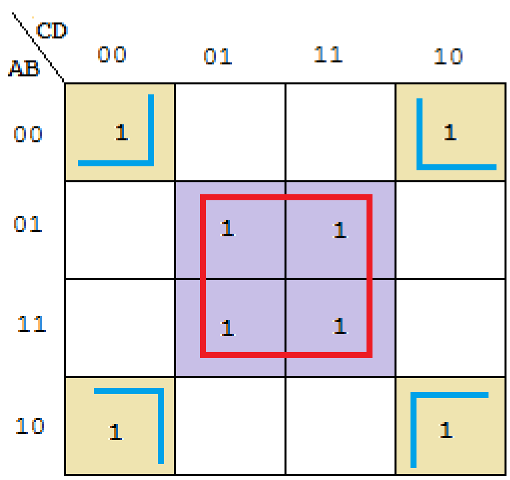


- 8 adjacent cells having 1's can be combined to form a single term y' by dropping 3 variables (w, x, z) that vary
- Remaining three 1's on the right can be combined in two groups of 4 squares. The top two 1's on the right are combined with top two 1's on the left giving the term w'z'
- The only cell left in 3rd row and 4th column. Instead of taking this cell alone (which will give a term of 4 literals), we can combine with four adjacent cells though already used to form the term xz'
- The final simplified expression is thus

  F = y' +m w'z' + xz'

Example:
Simplify F = ∑ (0, 2, 5, 7, 8, 10, 13, 15)

Solution:
- Since the function has the maximum minterm m15 it is a 4-variable function and thus the K-map has 16 cells/squares as shown below D.
- 8 minterms are marked with 1's in the corresponding squares in the K-map



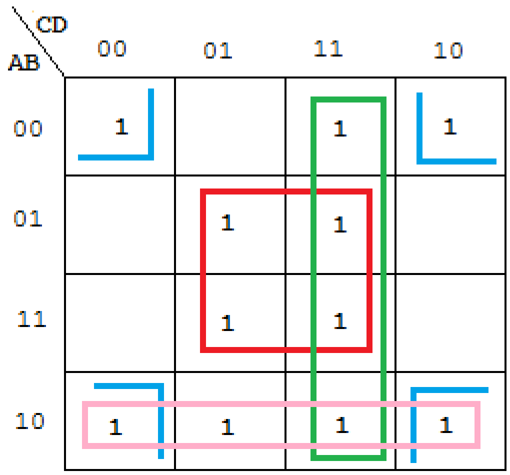
- They may be combined in two groups each having 4 squares
- Reduced term for the group having 4 central cells is BD by dropping A and C as they vary
- 4 corner squares being adjacent to each other form a group and the reduced term is B'D' by dropping A and C as they vary
- Thus the simplified function is: F = BD + B'D'

Example:

Simplify F = ∑ (0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)

Solution:

- Minterms are plotted in the corresponding cells in the K-map as shown



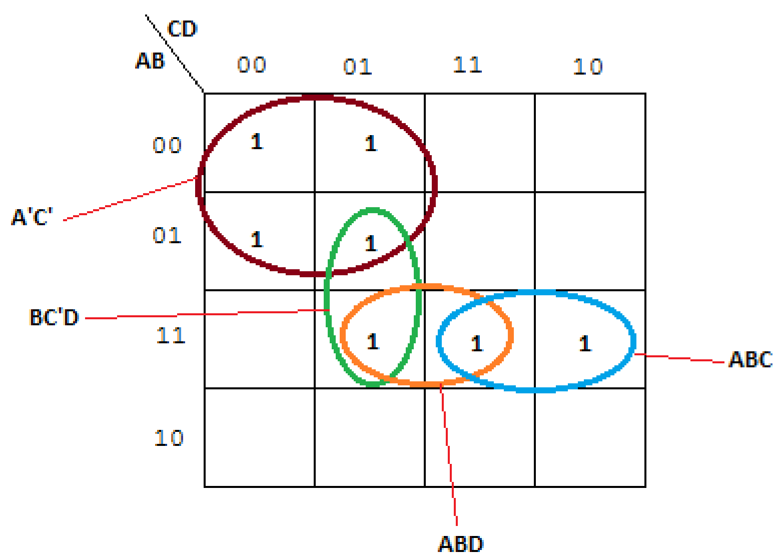
- 1's are covered in 4 groups all being essential prime implicants as they all have at least one 1 that is not shared by other.
- Thus minimal solution is:

  F = B'D' + BD + AB' + CD

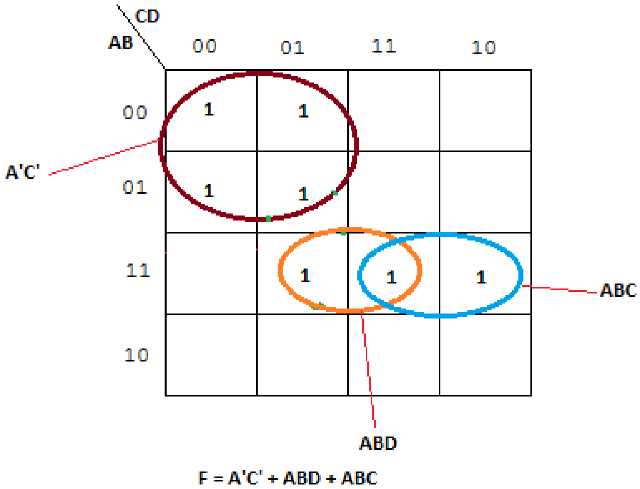
Example (with more than one solution):
Simplify F = ∑ (0, 1, 4, 5, 13, 14, 15)

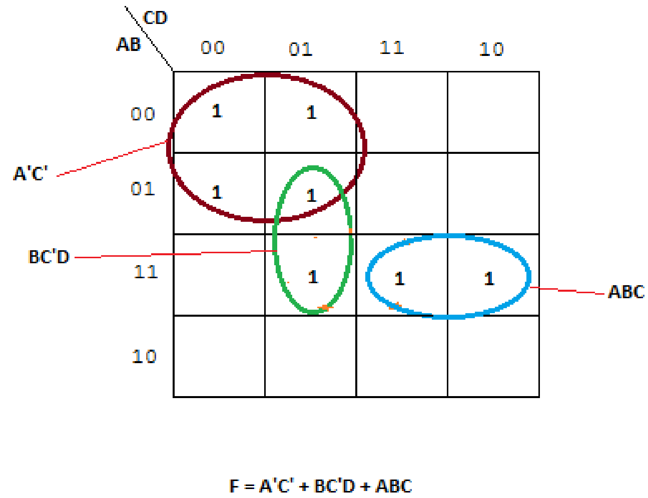
Solutions:

- Each of the Groups is a prime implecants (PI) but BC'D and ABD are not essential. One these implecants is to be removed.



- Removing one PI gives a solution and the other will give another as shown:



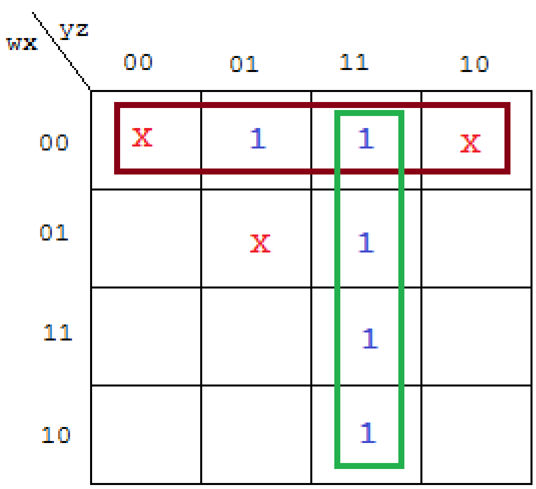


- Each solution is equally valid

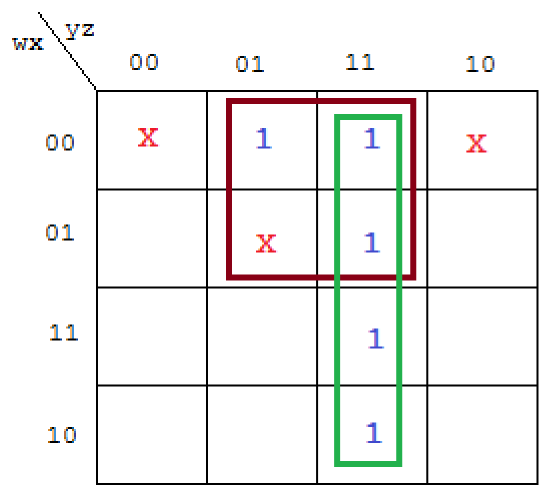
Don't Care Condition:
A Boolean function may be expressed as sum of minterms contributing 1. The function is 0 for remaining minterms. We assume that all the combinations of variables are valid. In practice, there are some applications where all combinations of variables are not valid. For example, 4-bit binary code for BCD number has six combinations (1010, 1011, 1100, 1101, 1110, 1111) that are not used and considered to be unspecified. Functions that have unspecified outputs for some input combinations are called **incompletely specified functions** and we simply don't care what value is assumed by the function for those unspecified minterms. These minterms are called don't care conditions. These don't care conditions can be used on the map to provide further simplification of boolen function.

Don't care minterms are marked in the map as X to distinguish them from 1's and 0's. When simplifying the function, X's can be included with the groups of 0 or 1 to make the group larger so that best minimization may be achieved.

Example:
Simplify the boolean function
$$F = \sum (1,3,7,11,15)$$
That has the don't care conditions
$$D = \sum (0,2,5)$$
Solution:
Two solutions are possible as shown below:

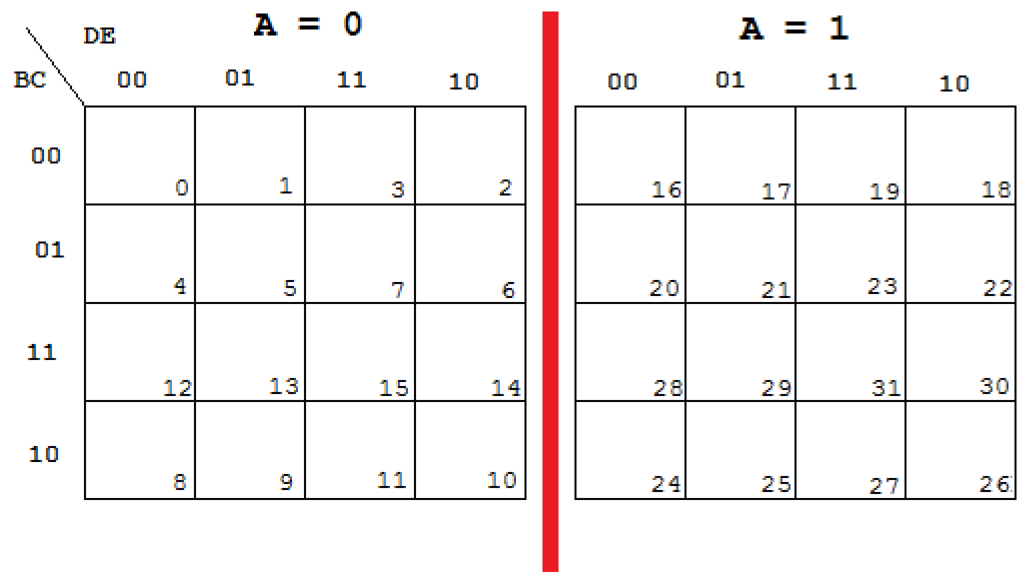


F = w'x'+yz



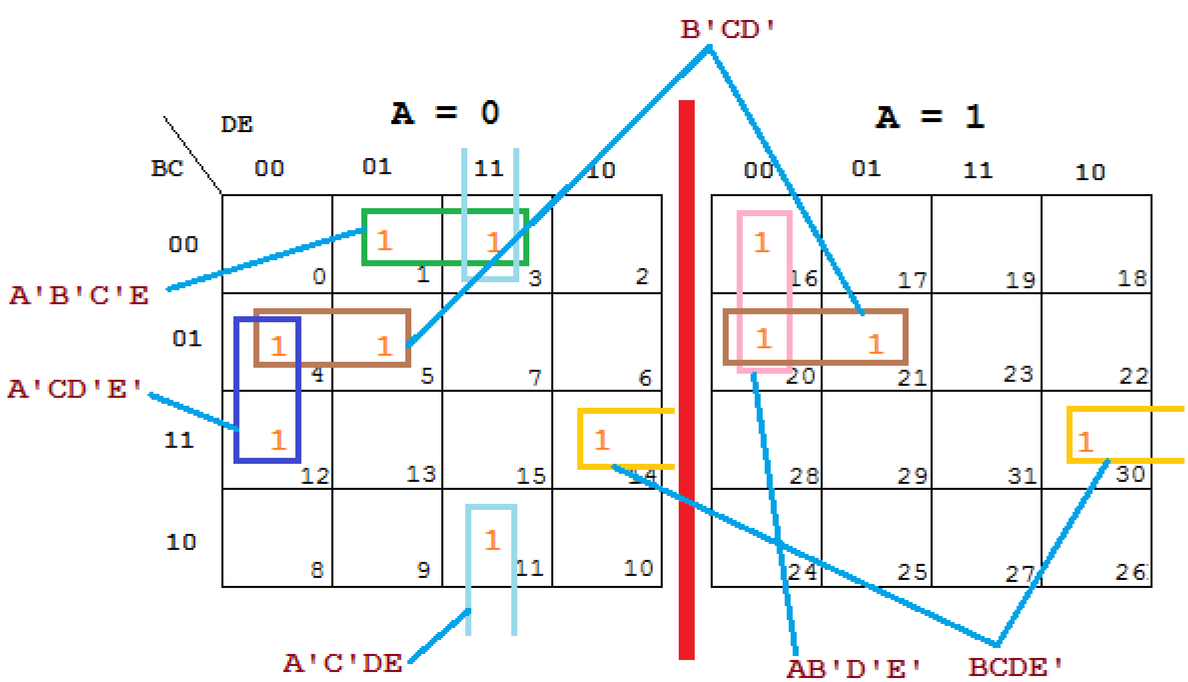
F = w'y+yz

5-Variable K-map
- Maps of more than 4 variables are more difficult to use because the geometry for combining adjacent squares becomes more involved
- For 5 variables, e.g. ABCDE, we need $2^5 = 32$ squares
- A 5-variable K-map can be drawn as two 4-variabe maps side by side. Left hand map corresponds to A=0 (minterms m0 to m15) and right hand map corresponds to A=1 (minterms m16 to m31)
- Each map has its own adjacent cells that can be combined to derive reduced terms
- Corresponding cells of the two maps are also adjacent as only one variable varies i.e. A
- For example, Cell #5 is adjacent to Cell #21, Cell #10 is adjacent to Cell #26 etc.



- A group of adjacent cells in one map is also said to be adjacent to the similar group in the other map.
- For example, a group of Cell Nos. (4, 5, 12, 13) is said to adjacent to the group of cells Nos. (20, 21, 28, 29) etc.
- This can be visualized by thinking one 4-variable map on TOP of the other

Example:
Simplify the function, F = ∑ m(1,3,4,5,11,12,14,16,20,21,30)

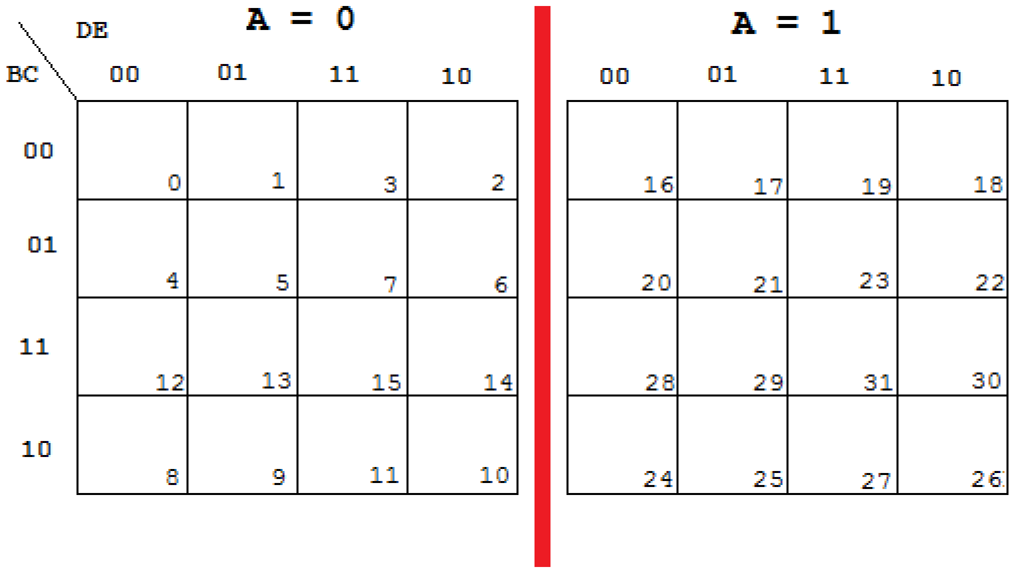


Simplified boolean function is:

F = B’CD’+BCDE’+A’B’C’E+A’CD’E’+A’C’DE+AB’D’E’

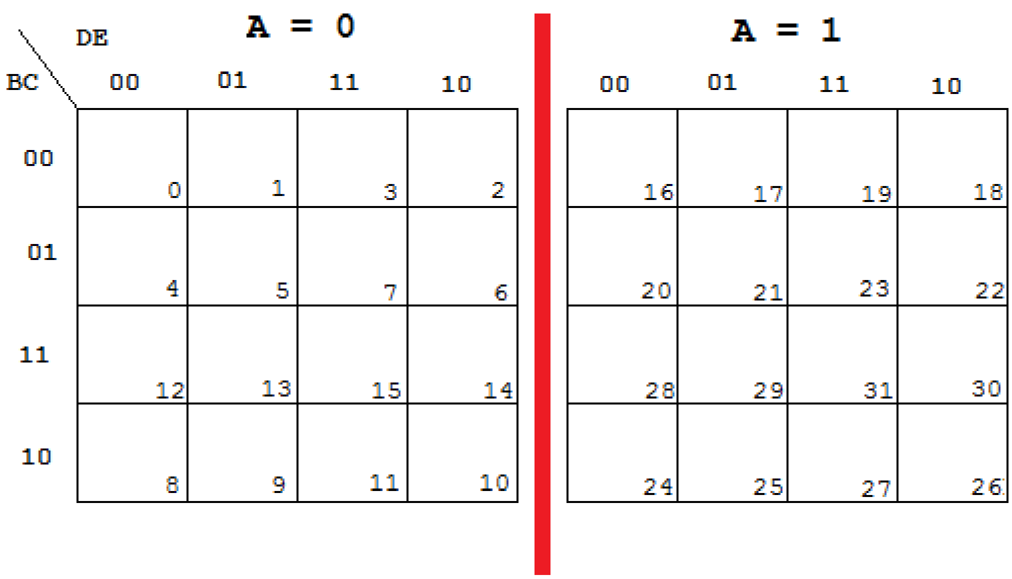
Problem:

Simplify: F = ∑ (0,2,3,5,7,8,11,13,17,19,23,24,29,30)



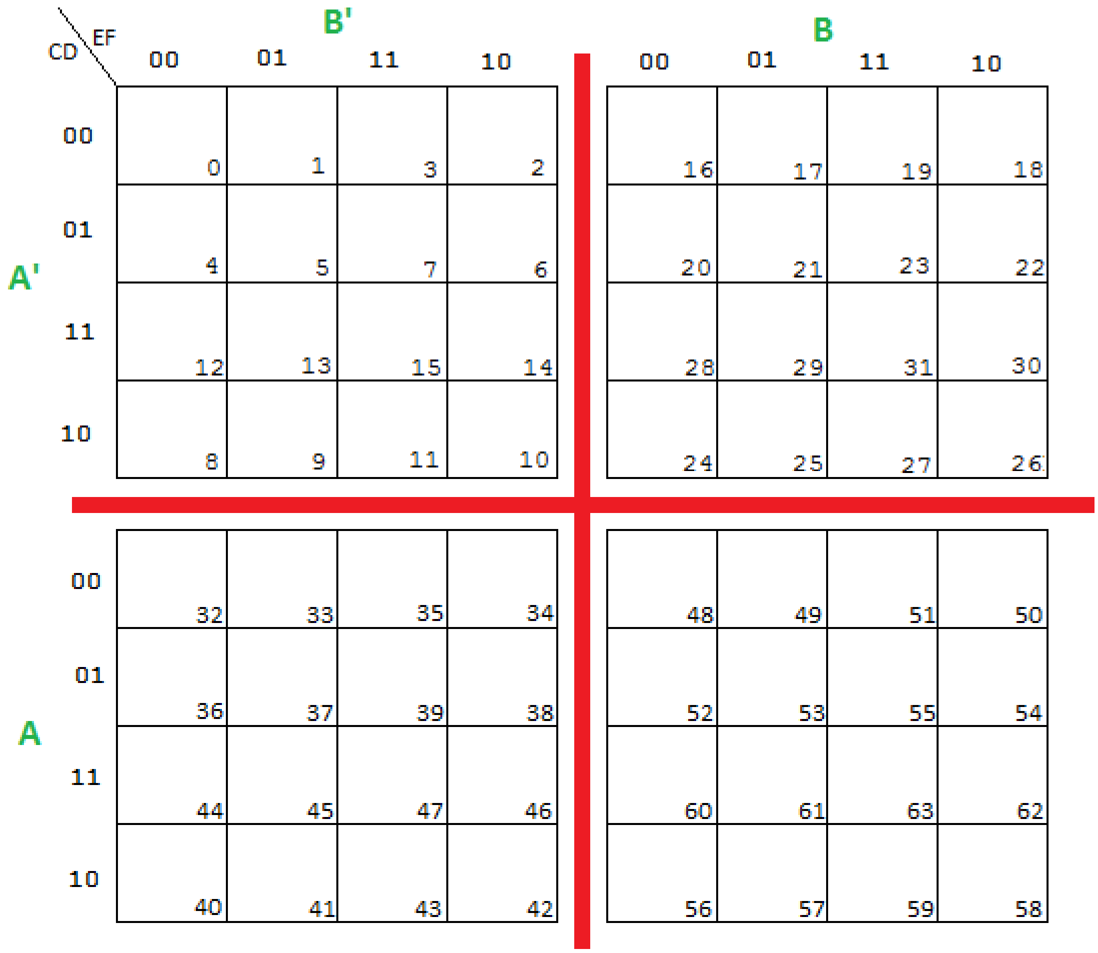
F =

Simplify: F = ∑ (0,1,2,3,8,9,16,17,20,21,24,25,28,29,30,31)



F =

6-Variable K-map

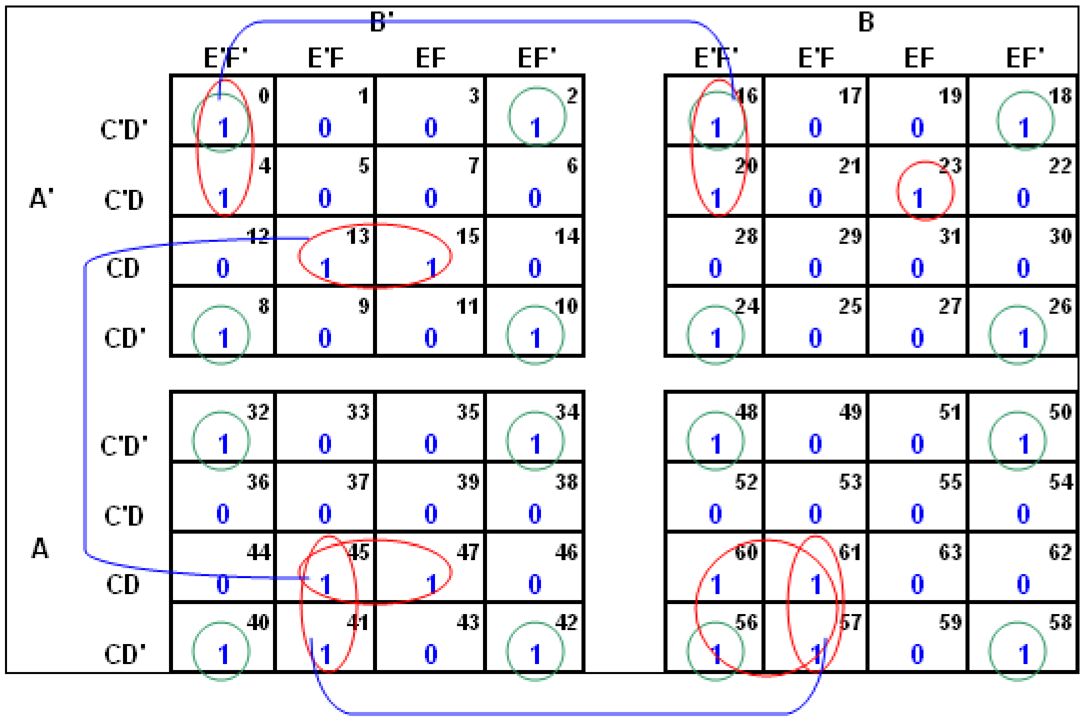


- A six variable map can be thought of 4 4-variable maps as shown
- Within a map, adjacency is similar to a 4-variable map
- Corresponding Cells in maps (in 2 maps or even in 4 maps) can be thought of adjacent
- Similar groups in maps (2 or 4 maps) can also be adjacent

Example:
F = Σ (0, 2, 4, 8, 10, 13, 15, 16, 18, 20, 23, 24, 26, 32, 34, 40, 41, 42, 45, 47, 48, 50, 56, 57, 58, 60, 61)

Solution:

- As the max term is 61, it is a six variable function, F(A,B,C,D,E,F)
- 1’s are put in the corresponding cells as below



- 16 Corner cells can be combined together to eliminate 4 variables. The term is: D’F’
- Cells 41, 45, 57, 61 can be combined to form – ACE’F
- Cells 13, 15, 45, 47 can be combined to form – B’CDF
- Cells 0, 4, 16, 20 can be combined to form – A’C'E’F’
- Cells 56, 57, 60, 61 can be combined to form – ABCE’
- Cell 23 is an isolated cell giving the term – A’BC’DEF

Thus the minimal function is:

F = D’F’ + ACE’F + B’CDF + A’C'E’F’ + ABCE’ + A’BC’DEF

Exercise:

F = Σ (0, 1, 2, 3, 4, 5, 8, 9, 12, 13, 16, 17, 18, 19, 24, 25, 36, 37, 38, 39, 52, 53, 60, 61)

Ans: F = A’B'E’ **+ A’C'D’ + A’D'E’ + AB’C'D + ABCE’**

Quine McCluskey Method:
The Quine McCluskey Method is an exact algorithm for finding minimal sum of products implementation of a Boolean function. This is tabular method and is the foundation of many software minimization methods.

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

1. Generate Prime Implicants
2. Construct Prime Implicant Table
3. Reduce Prime Implicant Table
   (a) Remove Essential Prime Implicants
   (b) Column Dominance
   (c) Row Dominance
4. Solve Prime Implicant Table

Coloumn Dominance and Row Dominance:
Row and column dominance relationships can be used to simplify the prime implicant table in the Quine McCluskey algorithm, as explained by the following definitions and theorems.

Definition 1
Two identical rows (or columns) **a** and **b** of a reduced prime implicant table are said to be *interchangeable.*

Definition 2
Given two rows **a** and **b** in a reduced prime implicant table, **a** is said to *dominate* **b**, if **a** has checks in all the columns in which **b** has checks and **a** and **b** are not interchangeable.

Definition 3
Given two columns **a** and **b** in a reduced prime implicant table, **a** is said to *dominate* **b**, if **a** has checks in all the rows in which **b** has checks and **a** and **b** are not interchangeable.

Theorem 1
Let **a** and **b** be rows of a reduced prime implicant table. Then, if **a** dominates **b** or **a** and **b** are interchangeable, there exists a minimal sum of products that does not include **b**; *dominated rows can be eliminated.*

Theorem 2
Let **a** and **b** be columns of a reduced prime implicant table. Then, if **b** dominates **a** or **a** and **b** are interchangeable, there exists a minimal sum of products that does not include **a**; *dominating columns can be eliminated*.

Examples:
Simplify the Boolean function
$F = \sum (0, 1, 2, 8, 10, 11, 14, 15)$

Generation of Prime Implicants:
Step 1: Group minterms based on the number 1's present in their binary representations

Step 2: Each minterm of a group is compared with all the minterms of the next group to find a mismatch only in one position, put tick marks to the right of both the minterms and write the new terms by omitting the variable that differs. This process will be continued until all the minterms are compared.

Step 3: The searching process in step 2 will be repeated for column II and column III will be created

No. of 1's in the binary representation of the minterms	Column I (Minterms)			Column II (Size 2 Implicants)			Column III (Size 4 Implicants)		Repeated Implicants are neglected except one
		wxyz			wxyz			wxyz	
0	0	0000	√	0, 1	000-	P1	0, 2, 8, 10	-0-0	P2
	------------------------			0, 2	00-0	√	0, 8, 2, 10	-0-0	x
1	1	0001	√	0, 8	-000	√	----------------------------		
	2	0010	√	-------------------------			10, 11, 14, 15	1-1-	P3
	8	1000	√	2, 10	-010	√	10, 14, 11, 15	1-1-	x
	-----------------------			8, 10	10-0	√	----------------------------		
2	10	1010	√	-------------------------					
	------------------------			10, 11	101-	√			
3	11	1011	√	-------------------------					
	14	1110	√	10, 14	1-10	√			
	-----------------------			-------------------------					
4	15	1111	√	11, 15	1-11	√			
	-----------------------			14, 15	111-	√			

Step 4: The unchecked terms form the prime implicants. Any term, however, appearing more than once will be cancelled except one. The sum of the prime implicants will give a valid expression of the function but not necessarily a minimal one. The selection of prime implicants that form the minimal function is made from a prime implicant table.
Step 5: Selection of Prime Implicants

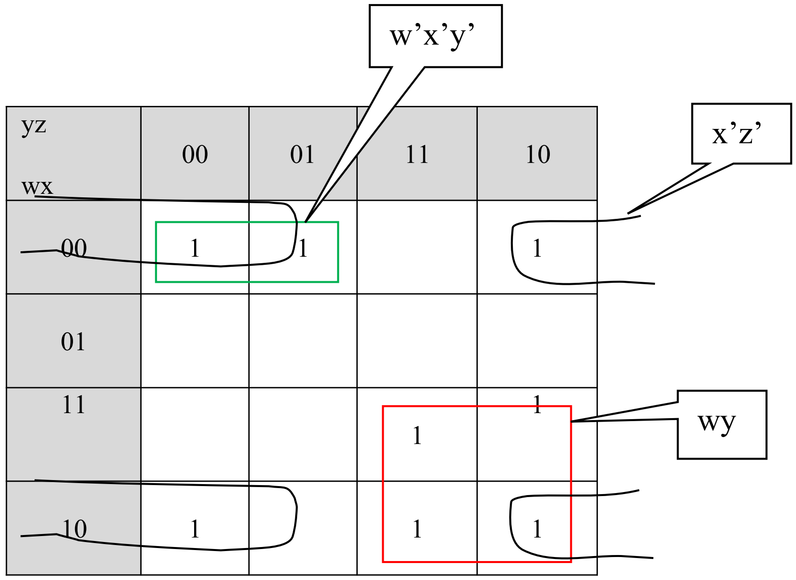
A table is formed having prime implicants in rows and minterms in columns as follows to select the necessary prime implicants.

EPI	Prime Implicants (PI)		Minterms contained in the PIs	Minterms							
				0	1	2	8	10	11	14	15
*	P1	w'x'y'	0, 1	X	X						
*	P2	x'z'	0, 2, 8, 10	X		X	X	X			
*	P3	wy	10, 11, 14, 15					X	X	X	X
				√	√	√	√	√	√	√	√

Minterms 1, 2 and 11 are contained only in prime implicants P1, P2 and P3 respectively. So all of them are essential prime implicants and are to be considered for final minimal expression. Therefore, the minimal Function is:

$$F = w'x'y' + x'z' + wy$$

Let us solve the same problem by K-Map method for varification.



F = w’x’y’ + x’z’ + wy

Both the methods give the same solution.

Coloumn and Row Dominance:
Necessary prime implicants can be selected by row dominance and coloumn dominance as follows:

		Minterms =>	0	1	2	8	10	11	14	15
*	w’x’y’	0, 1	X	X						
	~~wyz’~~	~~10, 14~~					X		X	
	~~wyz~~	~~11, 15~~						X		X
*	x’z’	0, 2, 8, 10	X		X	X	X			
*	wy	10, 11, 14, 15					X	X	X	X
			√	√	√	√	√	√	√	√

Coloumn 2 is dominated by col 0, 8 and 10 and thus they are eliminated. Coloumn 11 and 15 dominates each other and thus one can be eliminated.

Now row wy dominates row wyz’ and wyz and thus dominated rows may be eliminated.
There is no other dominance relationship among the rows or the columns in the reduced prime implicant table. As each of the remaining rows corresponds to an EPI, the algorithm successfully terminates with the following solution:

F = P1 + P4+ P5 = w’x’y’ + x’z’ + wy

Example:
Simplify F = ∑ (4, 8, 9, 10, 11, 12, 14, 15)

Column I (Minterms)				Column II (Size 2 Implicants)			Column III (Size 4 Implicants)		Repeated Implicants are neglected except one
No. of 1's		wxyz			wxyz			wxyz	
1	4	0100	√	4, 12 (8)	-100		8, 9, 10, 11	10--	
	8	1000	√	8, 9 (1)	100-	√	8, 10, 9, 11	10--	x
------------------------------				8, 10 (2)	10-0	√	8, 10, 12, 14	1--0	
2	9	1001	√	8, 12 (4)	1-00	√	8, 12, 10, 14	1--0	x
	10	1010	√	---------------------------------			---------------------------------		
	12	1100	√	9, 11 (2)	10-1	√	10, 11, 14, 15	1-1-	
------------------------------				10, 11 (1)	101-	√	10, 14, 11, 15	1-1-	x
3	11	1011	√	10, 14 (4)	1-10	√	---------------------------------		
	14	1110	√	12, 14 (2)	11-0	√			
------------------------------				---------------------------------					
4	15	1111	√	11, 15 (4)	1-11	√			
				14, 15 (1)	111-	√			

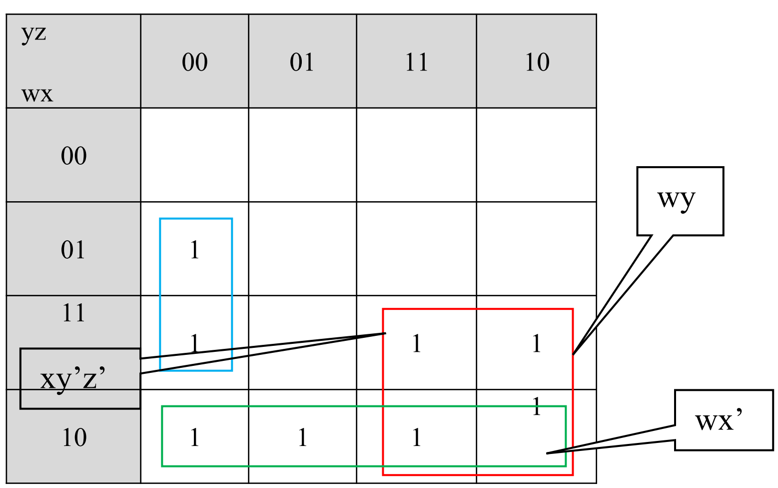
Prime implicant table to choose the necessary prime implicants:

Essential Prime Implicant (EPI)	Prime Implicants (PI)	Covering Minterms	Minterms							
			4	8	9	10	11	12	14	15
*	xy'z'	4, 12	x					x		
*	wx'	8, 9, 10, 11		x	x	x	x			
	wz'	8, 10, 12, 14		x		x		x	x	
*	wy	10, 11, 14, 15				x	x		x	x
			√	√	√	√	√	√	√	√

Here, the minterms xy'z', wx' and wy are essential prime implicants as they have minterms 4, 9 and 15 respectively those are not covered by any other prime implicant. These essential prime implicants also cover all the minterms and hence give the minimal SOP form of the function:

F = xy'z' + wx' + wy

Let us solve the problem by K map method



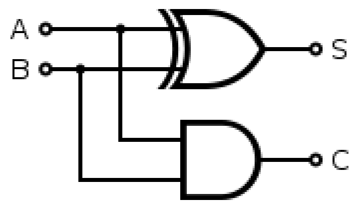
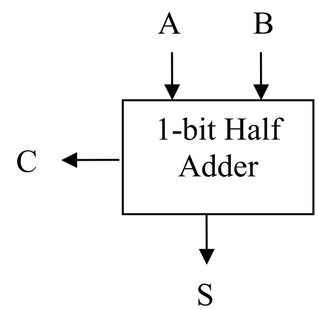
F = xy'z' + wx' + wy

Both the methods result in the same solution.

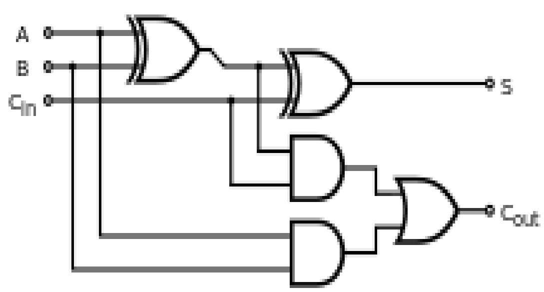
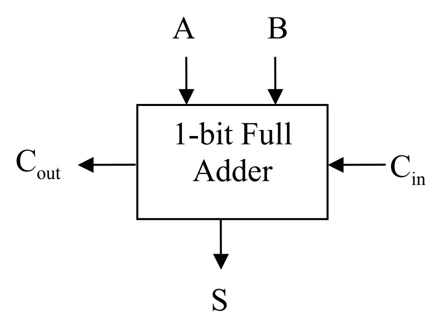
Combinational Circuits
Digital circuits are of two types, Combinational and sequential. In combinational circuits, the outputs at any instant are solely dependent on the present inputs. On the other hand, the outputs of sequential circuits at any instant depend not only on the present inputs but also on past inputs and past outputs. Thus sequential circuits need memory elements along with combinational circuits for its implementation.

BINARY ADDER AND SUBTRACTOR
A binary adder is a combinational circuit that adds binary data. Similarly, a binary subtractor subtracts a data from the other. A half adder/subtractor is a circuit that adds/subtracts two bits. A full adder/subtarctor takes care of carry/borrow.

The truth table and the circuit implementation of a half adder are as follows:

<table>
<tr><th colspan="2">Inputs</th><th colspan="2">Outputs</th><th>Implementation using gates</th><th>Symbol</th></tr>
<tr><td>A</td><td>B</td><td>C (Carry)</td><td>S (Sum)</td><td rowspan="5">
</td><td rowspan="5">
</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td></tr>
</table>

The truth table, expressions of sum and carry outputs, the circuit implementation of a full adder are as follows:

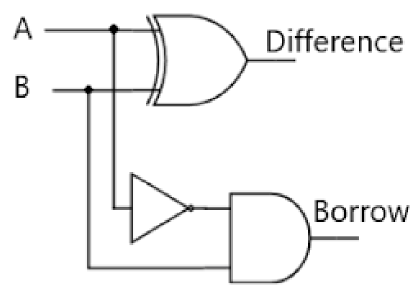
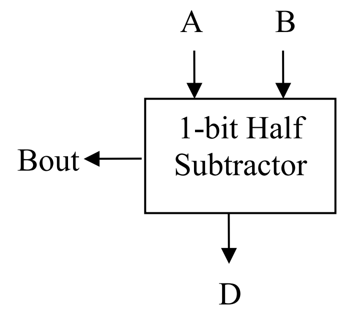
<table>
<tr><th colspan="3">Inputs</th><th colspan="2">Outputs</th><th>Implementation using gates</th><th>Symbol</th></tr>
<tr><td>A</td><td>B</td><td>C_{in}</td><td>C_{out}</td><td>S</td><td rowspan="9">
</td><td rowspan="9">
</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

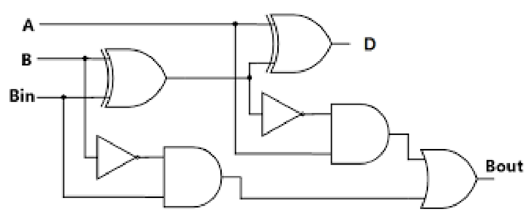
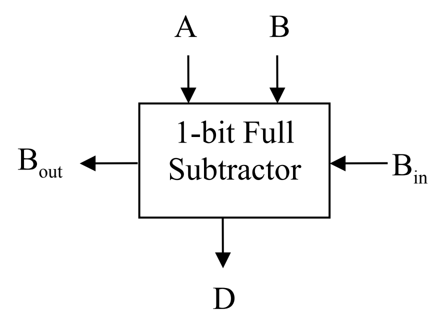
Expressions:

S = A’B’Cin + A’B(Cin)’ + AB’(Cin)’ + ABCin
= (A’B’Cin + ABCin) + (A’B(Cin)’ + AB’(Cin)’)
= (A XNOR B)Cin + (A XOR B)(Cin)’
= (A XOR B)’Cin + (A XOR B)(Cin)’
= A XOR B XOR Cin

Cout = A’BCin + AB’Cin + AB(Cin)’ + ABCin
= (A XOR B)Cin + AB

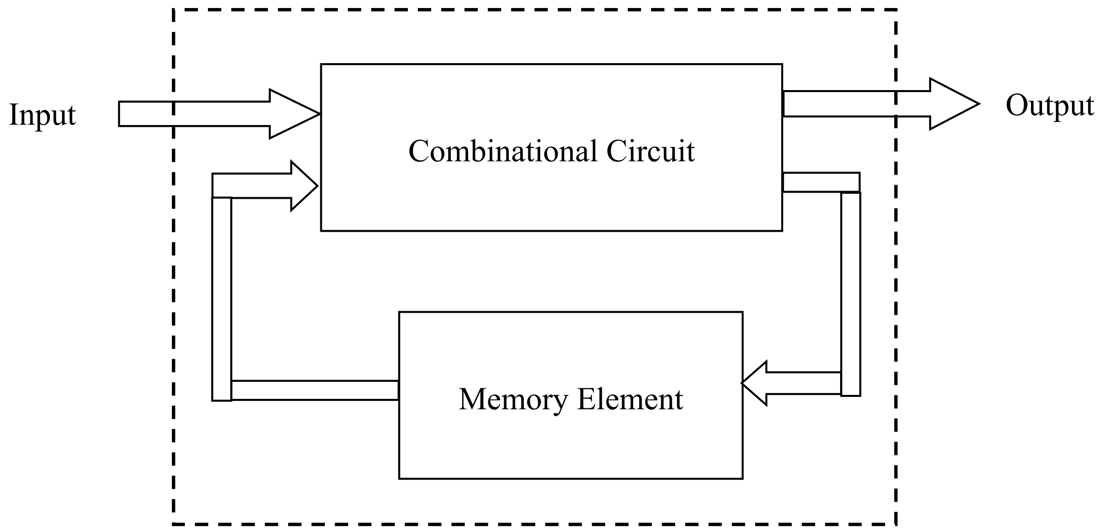
The truth table and the circuit implementation of a half subtarctor are as follows:

Inputs		Outputs (X-Y)		Implementation using gates	Symbol
A	B	Bout (Borrow)	D (Diff)		
0	0	0	0		
0	1	1	1		
1	0	0	1		
1	1	0	0		

Inputs			Outputs		Implementation using gates	Symbol
A	B	B_{in}	B_{out}	D		
0	0	0	0	0		
0	0	1	1	1		
0	1	0	1	1		
0	1	1	1	0		
1	0	0	0	1		
1	0	1	0	0		
1	1	0	0	0		
1	1	1	1	1		

Sequential Circuits:

So far we have studied combinational circuits. Output of a combinational circuit depends on its current inputs. There is also another class of circuit in digital electronics called sequential circuits. Output of a sequential circuit not only depends on the current inputs but also its past inputs and outputs i.e. output = f (current inputs, past inputs, past outputs). Then how to feed past inputs and past outputs? We need memory. Therefore, sequential circuit requires both combination circuit and memory elements for implementation. The block diagram of a sequential circuit is shown below:



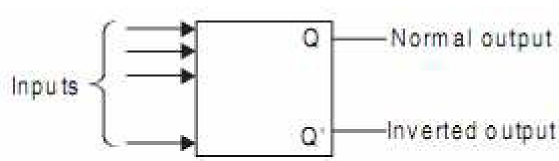
Sequential circuits are broadly classified into two main categories, known as synchronous or clocked and asynchronous or unclocked sequential circuits, depending on the timing of their signals. A sequential circuit whose behavior can be defined from the knowledge of its signal at discrete instants of time is referred to as a synchronous sequential circuit. A sequential circuit whose behavior depends upon the sequence in which the input signals change is referred to as an asynchronous sequential circuit.

LATCH and FLIP-FLOP

The basic 1-bit digital memory circuit is known as a latch. It can have only two states, either the 1 state or the 0 state. A latch is also known as a bistable multivibrator. Latches can be obtained by using NAND or NOR gates.

The major difference between flip-flop and latch is that the flip-flop is an edge-triggered type of memory circuit while the latch is a level-triggered type. It means that the output of a latch changes whenever the input changes. On the other hand, the flip-flop only changes its state whenever the control signal goes from low to high and high to low.

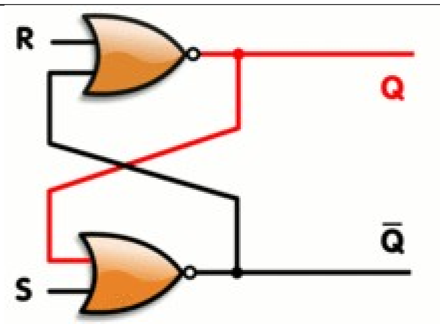
The general block diagram representation of a latch/flip-flop is shown in Figure below.

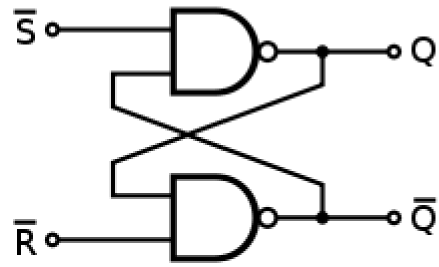


Block diagram of a flip-flop.

It has one or more inputs and two outputs. The two outputs are complementary to each other. If Q is 1 i.e., Set, then Q' is 0; if Q is 0 i.e., Reset, then Q' is 1. That means Q and Q' cannot be at the same state simultaneously.

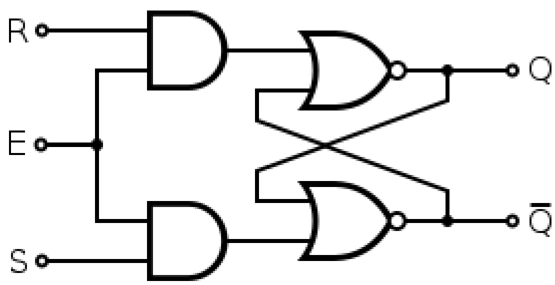
Basic Latch using NOR and NAND Gates and their Truth Tables

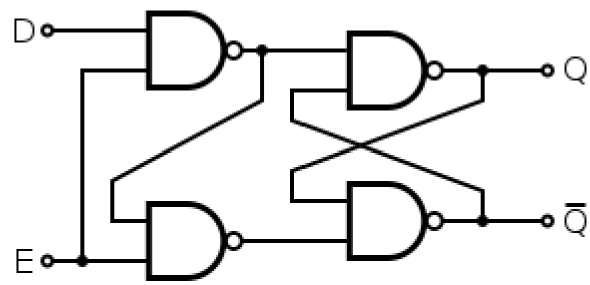


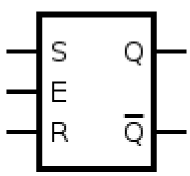


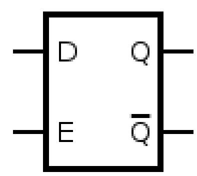
S	R	Q(t+1)	Action	S’	R’	Q(t+1)	Action
0	0	Q	Hold State	0	0	X	Not allowed
1	0	1	Set State	0	1	1	Set state
0	1	0	Reset State	1	0	0	Reset state
1	1	X	Not Allowed	1	1	Q	Hold state

Gated S-R and D latches and their symbols



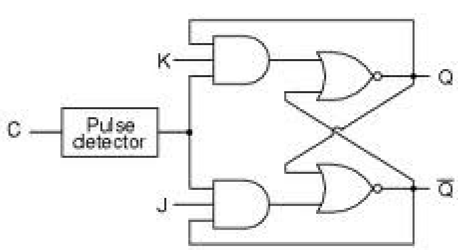




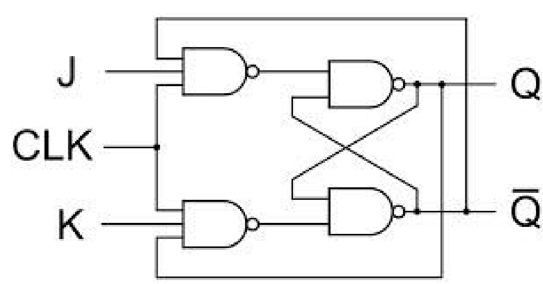


J-K Flip-flop:

JK flip-flop is basically an SR flip-flop with feedback which enables only one of its two input terminals, either SET or RESET to be active at any point of time thereby eliminating the invalid condition seen previously in the SR flip-flop circuit. Also when both the J and the K inputs are at logic level "1" at the same time, and the clock input is pulsed either "HIGH", the circuit will "toggle" from its SET state to a RESET state, or visa-versa. This results in the JK flip-flop acting more like a T-type toggle flip-flop when both terminals are "HIGH".



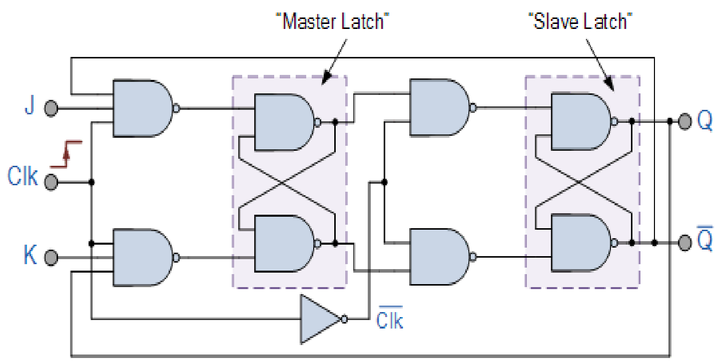
C	J	K	Q	Q̄
↑	0	0	latch	latch
↑	0	1	0	1
↑	1	0	1	0
↑	1	1	toggle	toggle
x	0	0	latch	latch
x	0	1	latch	latch
x	1	0	latch	latch
x	1	1	latch	latch



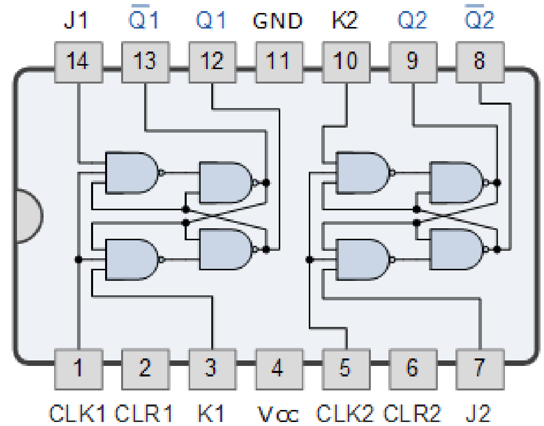
Although this circuit is an improvement on the clocked SR flip-flop it still suffers from timing problems called "race" if the output Q changes state before the timing pulse of the clock input has time to go "OFF". To avoid this the timing pulse period

(T) must be kept as short as possible (high frequency). As this is sometimes not possible with modern TTL IC's the much improved **Master-Slave JK Flip-flop** was developed.

Master Slave J K Flip flop



Dual JK Flip-flop 74LS73



The master-slave flip-flop eliminates all the timing problems by using two SR flip-flops connected together in a series configuration. One flip-flop acts as the "Master" circuit, which triggers on the leading edge of the clock pulse while the other acts as the "Slave" circuit, which triggers on the falling edge of the clock pulse. This results in the two sections, the master section and the slave section being enabled during opposite half-cycles of the clock signal.

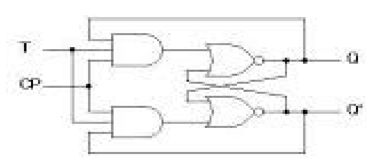
The input signals J and K are connected to the gated "master" SR flip-flop which "locks" the input condition while the clock (Clk) input is "HIGH" at logic level "1". As the clock input of the "slave" flip-flop is the inverse (complement) of the "master" clock input, the "slave" SR flip-flop does not toggle. The outputs from the "master" flip-flop are only "seen" by the gated "slave" flip-flop when the clock input goes "LOW" to logic level "0".

When the clock is "LOW", the outputs from the "master" flip-flop are latched and any additional changes to its inputs are ignored. The gated "slave" flip-flop now responds to the state of its inputs passed over by the "master" section. Then on the "Low-to-High" transition of the clock pulse the inputs of the "master" flip-flop are fed through to the gated inputs of the "slave" flip-flop and on the "High-to-Low" transition the same inputs are reflected on the output of the "slave" making this type of flip-flop edge or pulse-triggered.

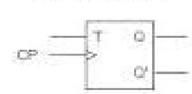
The 74LS73 is a Dual JK flip-flop IC, which contains two individual JK type bi-stable's within a single chip enabling single or master-slave toggle flip-flops to be made. Other JK flip-flop IC's include the 74LS107 Dual JK flip-flop with clear, the 74LS109 Dual positive-edge triggered JK flip-flop and the 74LS112 Dual negative-edge triggered flip-flop with both preset and clear inputs.

T Flip Flop

When both J = K = 1, Q(t+1) = Q' i.e. . Thus on arrival of every clock edge, the output will toggle. A toggle flip-flop or T flip-flop can be made from a J-K flip-flop by tying both of its inputs to logic 1. It is useful for constructing binary counters, frequency dividers etc.



(a) Logic diagram



(b) Graphical symbol

Q T	Q(t+1)
0 0	0
0 1	1
1 0	1
1 1	0

(c) Transition table

Clocked T flip-flop