CARDAMOM PLANTERS' ASSOCIATION COLLEGE
(Re-Accredited With 'A' Grade By NAAC)
Pankajam Nagar, Bodinayakanur - 625 582.

DEPARTMENT OF CS & IT

Unit -2

Boolean Algebra: Laws and Theorems – SOP, POS Methods – Simplification of Boolean Functions – Using Theorems, K-Map, Prime – Implicant Method – Binary Arithmetic: Binary Addition – Subtraction – Various Representations of Binary Numbers – Arithmetic Building Blocks – Adder – Subtractor

2.1. Boolean Algebra

Boolean Algebra in Digital Computer Fundamentals is the mathematical foundation used to design and analyse digital circuits (like logic gates, processors, memory, etc.).

2.1.1. What is Boolean Algebra?

Boolean algebra is a mathematical system used to represent and manipulate logical relationships between variables. It works with binary values:

- 1 (True)
- 0 (False)

It was introduced by **George Boole** and forms the foundation of all digital systems.

Basic Concept

Boolean algebra enables logical operations using basic operators such as:

- **AND (·)**
- **OR (+)**
- **NOT (′)**

These operators are used to model logical decisions and simplify complex expressions into binary form:

- **1 (True)**
- **0 (False)**

2.1.1.1. Order of Operations in Boolean Algebra

Just like in arithmetic (BODMAS), Boolean algebra follows a specific **order of operations** to evaluate expressions correctly. This ensures consistent and accurate results in logical computations.

Priority Order

1. **Parentheses ( )**
   - Expressions inside brackets are evaluated first.
2. **NOT (′)**
   - Also called **complement** or **inversion**
   - Applied next to invert the value (0 → 1, 1 → 0)

S.ROHINI

3. **AND (·)**
    - Multiplication operation in Boolean algebra
4. **OR (+)**
    - Addition operation, evaluated last

Summary Rule

( ) → NOT → AND → OR

Example

Evaluate:
$A + B \cdot C'$

Step-by-step:

1. NOT: C'
2. AND: $B \cdot C'$
3. OR: $A + (B \cdot C')$

Understanding the order of operations is essential for correctly simplifying Boolean expressions and designing accurate digital circuits.

Importance in Digital Systems

Boolean algebra is essential for designing and optimizing:

- Logic gates (AND, OR, NOT, NAND, NOR, XOR)
- Combinational circuits (adders, multiplexers)
- Sequential circuits (flip-flops, registers)
- Memory units in hardware systems

It allows efficient logic gate design by reducing complexity and improving performance.

Simplification and Optimization

- Boolean expressions can be simplified using Boolean laws
- Karnaugh Maps (K-Maps) are used to minimize logic functions
- Simplification helps:
    - Reduce circuit complexity
    - Lower hardware cost
    - Minimize power consumption
    - Increase speed and efficiency

Applications in Software

Boolean algebra is widely used in software development for:

- Conditional statements (if–else logic)
- Search algorithms
- Database queries (using AND, OR conditions)
- Decision-making and control structures

2.1.2. Laws and Theorems of Boolean Algebra

Boolean laws and theorems are used to simplify logical expressions and design efficient digital circuits.

1. Identity Law

A variable remains unchanged when combined with 0 (OR) or 1 (AND).

$A+0=A$

$A\cdot 1=A$

A	A + 0 = A	A·1 = A
0	0 + 0 = 0	0.1 = 0
1	1 + 0 =1	1.1 = 1

2. Null Law (Domination Law)

A variable ANDed with 0 gives 0, and ORed with 1 gives 1.

$A+1=1$

$A\cdot 0 = 0$

A	A + 1 =1	A · 0 =0
0	0 + 1 = 1	0 . 0 = 0
1	1 + 1 = 1	1 . 0 = 0

3. Idempotent Law

Combining a variable with itself gives the same variable.

$A+A=A$

$A\cdot A=A$

A	A + A = A	A · A = A
0	0 + 0 = 0	0 . 0 = 0
1	1 + 1 = 1	1 . 1 = 1

S.ROHINI

4. Complement Law

A variable combined with its complement gives 1 (OR) or 0 (AND).

$A+A' = 1$

$A \cdot A' = 0$

A	A'	A+A' = 1	A·A' = 0
0	1	0 + 1 = 1	0 . 1 =0
1	0	1 + 1 = 1	1. 0 = 0

5. Commutative Law

The order of variables does not affect the result.

$A+B=B+A$

$A \cdot B=B \cdot A$

A	B	A+B	B+A	A.B	B.A
0	1	1	1	0	0
1	0	1	1	0	0

6. Associative Law

The grouping of variables does not affect the result.

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

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

7. Distributive Law

AND distributes over OR, and OR distributes over AND.

1. A·(B+C) = AB+AC

A	B	C	B+C	A(B+C)	AB+AC
1	0	1	1	1	1

2. A+(BC)=(A+B)(A+C)

RHS

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

$= A.A + A.C + A.B + B.C$ $\qquad (\because A.A=A)$

S.ROHINI

= A + AC + AB + BC

= A (1 +C) +AB + BC (∵ A+1 =1)

= A + AB + BC = A (1 +B) + BC (∵ A+1 =1)

= A+ BC

Important Theorems

1. De Morgan’s Theorems

The complement of a sum becomes product of complements, and vice versa.

$(A+B)'=A'B'$

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

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

2. Absorption Law

A variable absorbs the term combined with it.

1. A+AB=A,

=A+AB

=A(1+B)

=A

2. A(A+B) =A

= A.A + A.B (∵ A.A = A – use Idempotent law)

= A + A.B

= A(1+B) (∵ A+1 =1 – use Null law)

=A

3. Double Negation Law

The complement of a complement gives the original variable.

$(A')'=A$

A	A'	(A')'

S.ROHINI

0	1	0
1	0	1

4. Redundant Literal Rule

A **redundant literal rule** states that a term containing a variable and its complement can be simplified by eliminating the unnecessary (redundant) part without changing the result. (Extra or repeated literals that do not affect the output can be removed to simplify the Boolean expression.)

Law 1: $A + A'B = A+B$

LHS

$= A + A'B$ (∵ Apply distributive law: $A+(BC)=(A+B)(A+C)$)

$= (A + A')(A + B)$ (∵ Using complement law $(A + A' = 1)$)

$= 1.(A + B)$

$= A + B$ =RHS

Law 2: $A(A'+B)=AB$

LHS

$= A(A'+B)$ (∵ Apply distributive law: $A\cdot(B+C) = AB+AC$)

$= A.A' + A.B$ (∵ Using complement law $(A \cdot A' = 0)$)

$= A.B$ =RHS

5. Transposition Theorem

This theorem helps to convert a **sum of products into product of sums** by rearranging terms.

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

RHS

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

$= A.A' + A.B + A'C + BC$ (∵ Using complement law $(A \cdot A' = 0)$)

$= 0 + A.B + A'C + BC$

$= A.B + A'C + BC(A + A')$ (∵ Using complement law $(A + A' = 1)$)

$= AB + A'C + ABC + A'BC$

$= AB + ABC + A'C + A'BC$

$= AB(1 + C) + A'C(1 + B)$

$= AB + A'C$ = LHS

6. Duality Theorem

S.ROHINI

The duality theorem states that a Boolean expression remains valid if we interchange **AND (·) with OR (+)** and **0 with 1**.

Key Idea

- Replace + ↔ ·
- Replace **0 ↔ 1**
- Variables remain the same

Example

Given expression:

$$A + 0 = A$$

Dual expression:

$$A \cdot 1 = A$$

Another example:

$$A + B = B + A$$

Dual:

$$A \cdot B = B \cdot A$$

2.1.3. SOP and POS Methods in Boolean Algebra

2.1.3.1. SOP (Sum of Products)

SOP is a Boolean expression where multiple **product (AND) terms** are combined using **OR (+)**.

F=AB+A′C+BC

- Each term is a product (AND)
- Terms are added (ORed) together

Types of SOP

1. Standard SOP (Non-Canonical SOP)

Product terms are ORed together, but **each term may not contain all variables**.

Example:

$$F(A, B, C) = AB + C$$

- Term **AB** → missing variable **C**
- Term **C** → missing variables **A, B**

✔ So, this is **Standard SOP (Non-Canonical)**

2. Canonical SOP (Minterm Form)

SROHINI

Each product term contains **all variables exactly once** (either complemented or uncomplemented).

Example 1:

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

- Each term has **A, B, C**
- Variables appear in either normal or complemented form

✔ So, this is **Canonical SOP**

Example 2: (with Minterm Notation):

$$F(A,B,C) = \Sigma m(1,3,7)$$

Expanded form:

$$= A'B'C + A'BC + ABC$$

✔ This is also **Canonical SOP**

2. POS (Product of Sums)

POS is a Boolean expression where multiple **sum (OR) terms** are combined using **AND (·)**.

Form:

$$F = (A + B)(A' + C)(B + C)$$

- Each term is a sum (OR)
- Terms are multiplied (ANDed) together

Types of POS

1. Standard POS (Non-Canonical POS)

Each term is a sum (OR), but **may not contain all variables**.

Example:

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

- Term **(A + B)** → missing variable **C**
- Term **(C + A')** → missing variable **B**

✔ So, this is **Standard POS**

2. Canonical POS (Maxterm Form)

Each sum term contains **all variables exactly once** (either complemented or uncomplemented).

Example:

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

- Each term has **A, B, C**
  ✔ So, this is **Canonical POS**

S. ROHINI

Maxterm Notation Example:

$$F(A, B, C) = \Pi M(0,2,5)$$

Expanded form:

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

✔ This is **Canonical POS**

2.2. Simplification of Boolean Functions

Simplification of Boolean functions is the process of reducing a Boolean expression to its **simplest form** so that it uses **minimum logic gates** and variables.

Why Simplification is Needed

- Reduces number of logic gates
- Lowers hardware cost
- Minimizes power consumption
- Increases speed and efficiency

2.2.1. Methods of Simplification of Boolean Functions

1. Algebraic Method (Using Boolean Laws)

Simplifies Boolean expressions by applying standard **Boolean laws and theorems** such as:

- Identity, Null, Complement laws
- Absorption law
- De Morgan's theorem

Example:

$$A + AB = A$$

✔ Simple and effective for small expressions

2. Karnaugh Map (K-Map) Method

- A **graphical method** used for simplification
- Best suited for **2 to 4 variables**
- Groups:
    - **1's** for SOP form
    - **0's** for POS form

✔ Provides minimal expression easily

3. Prime–Implicant Method

- A **systematic/tabular method** for simplification

S.ROHINI

- Involves identifying:
  - **Prime Implicants (PI)**
  - **Essential Prime Implicants (EPI)**
- Suitable for **larger expressions**

✔ More accurate for complex problems

2.2.1.1. Algebraic Method (Using Boolean Laws)

The algebraic method simplifies Boolean expressions by applying **Boolean laws and theorems** to reduce them into a simpler form.

Steps to Simplify

1. Apply basic laws (Identity, Null, Complement)
2. Use **Distributive, Absorption, and De Morgan's laws**
3. Remove redundant terms
4. Repeat until no further simplification is possible

Common Laws Used

- $A + 0 = A, A \cdot 1 = A$
- $A + A' = 1, A \cdot A' = 0$
- $A + AB = A$(Absorption)
- $(A + B)' = A'B'$(De Morgan)

Example Problems:

1. Simplify: $A + A'B + AB$

Solution:

$= A + AB + A'B$ (since $A + AB = A$)
$= A + A'B$ (since distributive law $A+(BC)=(A+B)(A+C)$)
$= (A + A')(A + B)$ $\because (A + A' = 1)$
$= 1 \cdot (A + B)$
$= A + B$

2. Simplify: $AB + A'C + BC$

Solution:

$= AB + A'C + BC$ $\because (A + A' = 1)$
$= AB + A'C + BC(A + A')$
$= AB + A'C + ABC + A'BC$
$= AB(1 + C) + A'C(1 + B)$
$= AB + A'C$

3. **Simplify: $(A + B)(A + B')$**

S ROHINI

Solution:

$= (A + B)(A + B')$
$= AA + AB' + AB + BB'$ $\quad \because (A.A' = 0)\ (A.A = A)$
$= A + AB + AB'$ $\quad \because (A + AB = A)$
$= A + AB' = A(1 + B')$ $\quad \because (1+B' = 1)$
$= A$

4. **Simplify: $(A+B)'+AB$**

Solution:

$= (A+B)'+AB$ $\quad \because (A + B)' = A'.B'$
$= A'B' + AB$

5. Simplify: $(A + B)(A' + C)$

Solution:

$= (A + B)(A' + C)$
$= AA' + AC + A'B + BC$ $\quad \because (A.A' = 0)$
$= 0 + AC + A'B + BC$
$= AC + A'B + BC$

Now observe:

- Term **BC** is actually redundant

Use the identity:

$= AC + A'B + BC = AC + A'B$

- If **A = 1**, then **AC covers the output**
- If **A = 0**, then **A'B covers the output**
- So **BC doesn't add anything new** → it is unnecessary

Final Answer

$= AC + A'B$

6. Simplify: $A + (A + B)(A + C)$

Solution:

$= A + (A + B)(A + C)$
= A + AA + AC+ AB+BC $\quad (A.A = A)$
= A+A+AC+AB+BC $\quad (A + A = A)$
= A+ AC + AB+ BC $\quad (A + AC = A)$
= A+ AB+BC $\quad (A + AB = A)$
= A+BC

7. Simplify: $(A + B + C)(A + B + C')$

Solution:

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

$= AA + AB + AC' + AB + BB + BC' + AC + BC + CC'$ $(A.A = A)$ $(A.A' = 0)$

$= A + AB + AC' + AB + B + BC' + AC + BC$ $(A + AB = A)$ $(B + AB = B)$
$= A + AC' + B + BC' + AC + BC$

$= A + B + BC' + BC + AC + AC' = A + B + B(C + C') + A(C + C')$ $\because (A + A' = 1)$

$= A + B + B + A = A + A + B + B$ $(A + A = A)$

$= A + B$

8. Simplify: $AB + A'B + AB'$

Solution:

$= AB + A'B + AB'$

$= B(A + A') + AB'$ $\because (A + A' = 1)$

$= B + AB'$ $\because (A + \text{BC} = (A + B)(A + C))$

$= (A + B)(B + B')$ $\because (A + A' = 1)$

=A+B

2.2.1.2. Karnaugh Map (K-Map) Method

K-Map is a **graphical method** used to simplify Boolean expressions by grouping adjacent cells to obtain a minimal expression.

Key Features

- Used for **2 to 4 variables** (sometimes 5)
  - One variable $2^1 = 2$ cells
  - 2 Variable $2^2 = 4$ Cells
  - 3 variable $2^3 = 8$ cells
  - 4 Variable $2^4 = 16$ cells
- Based on **Gray code** (only one-bit changes between adjacent cells)
- Adjacent cells can be grouped

SOP Form (K-map)

K -Map representation

SROHINI

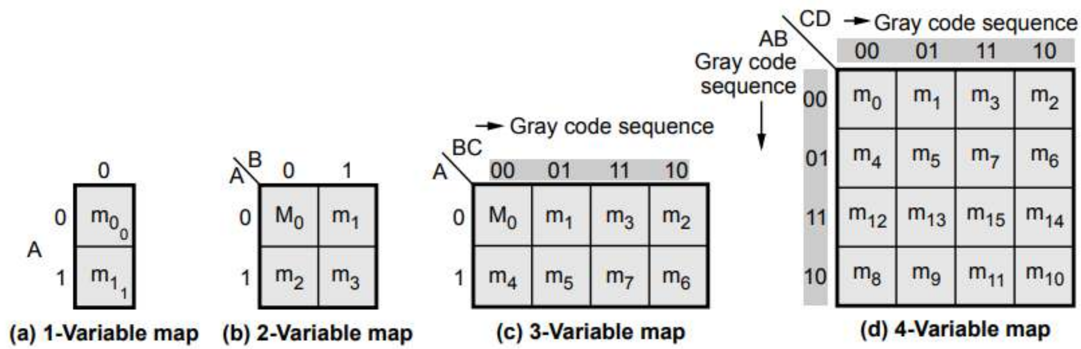


Fig. 3.3.3 Another way to represent 1, 2, 3 and 4-variable maps for SOP expressions

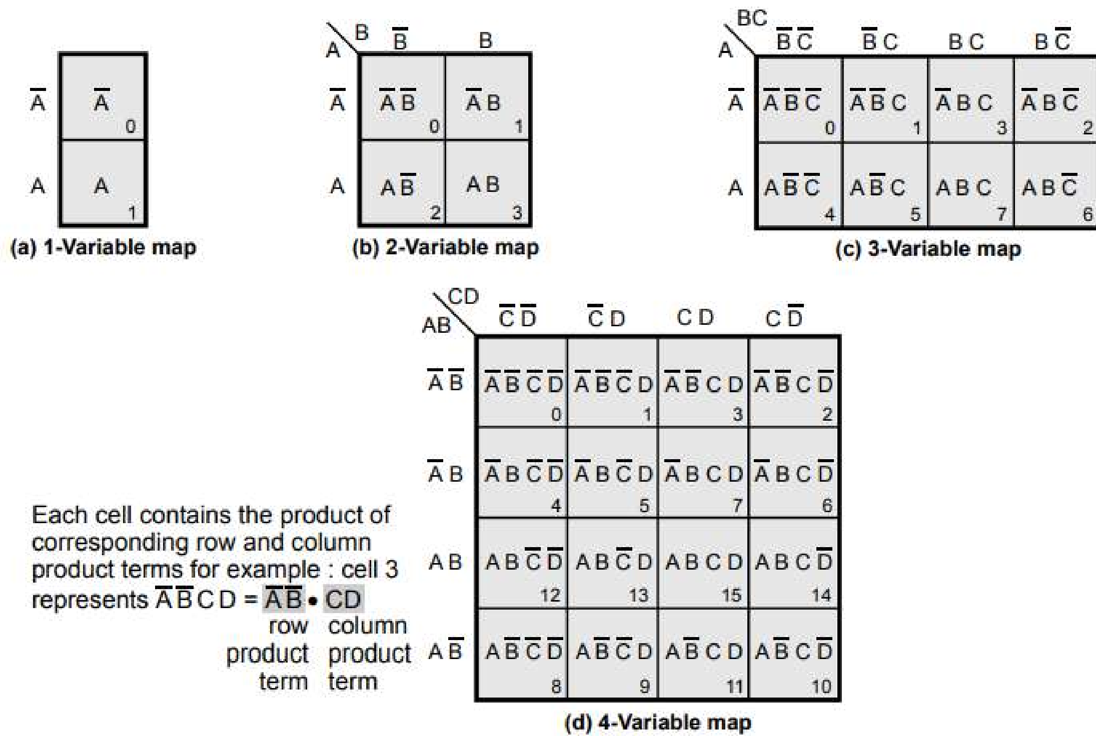


Fig. 3.3.2 1, 2, 3 and 4-variable maps with product terms

Steps to Solve (SOP Form)

1. Draw K-map based on number of variables (2, 3, or 4 variables)
2. Fill **1's** in cells corresponding to **minterms**
3. Group adjacent 1's in sizes of:
    - 1, 2, 4, 8… (powers of 2)
4. Make **largest possible groups**
5. **Grouping Rules**
    - Groups must be **rectangular**
    - Groups can **overlap**

- Don’t group diagonally
- Corners are **adjacent (wrap-around)**
- Every 1 must be included at least once

6. Write simplified expression from groups

   For each group:

   - Include only variables that are **constant (same)**
   - If variable = 1 → write **normal variable**
   - If variable = 0 → write **complement**

<u>POS Form (K -Map representation)</u>

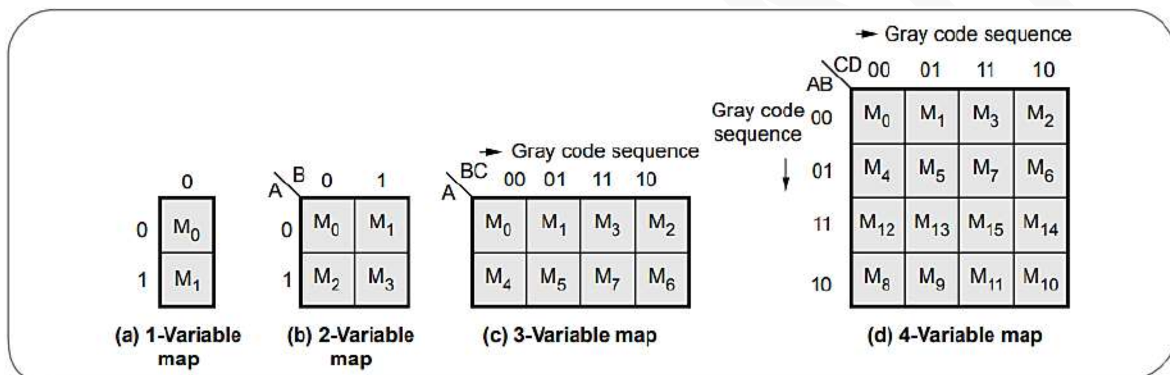


Fig. 3.3.5 Another way to represent 1, 2, 3 and 4-variable map for POS expressions

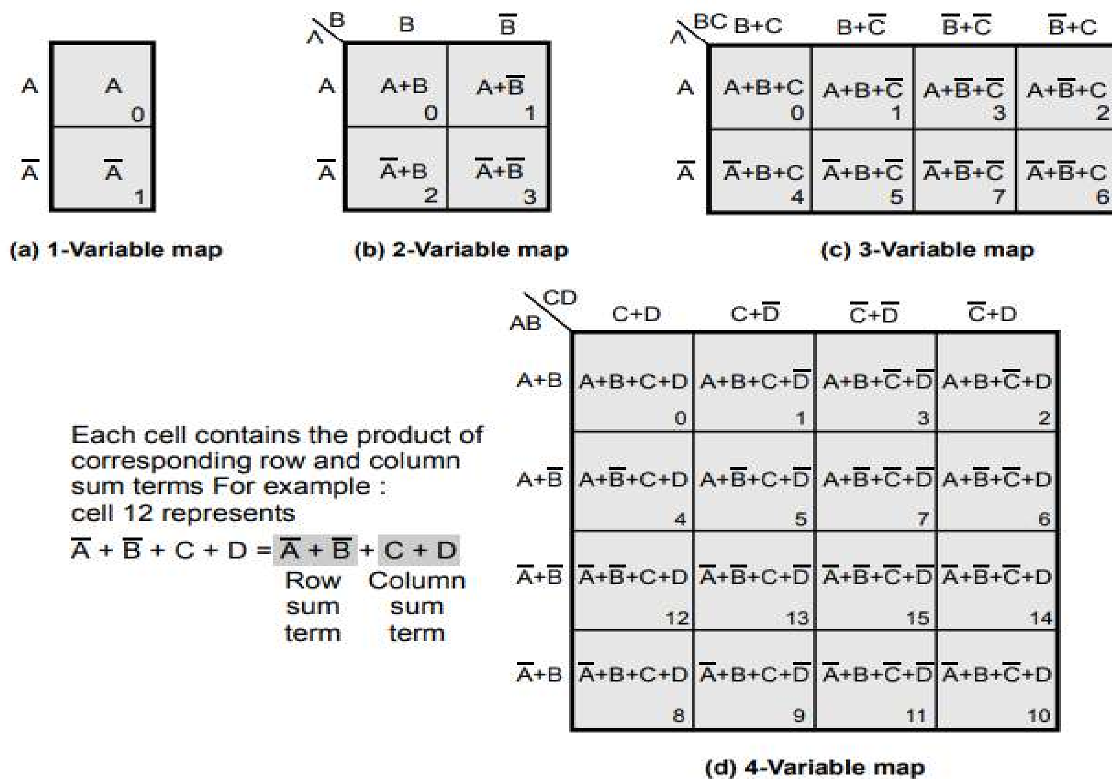


Fig. 3.3.4 1, 2, 3 and 4-variable maps with sum terms

Steps to Solve (POS Form)

1. Draw K-map based on number of variables (2, 3, or 4 variables)
2. Place **0's** in cells corresponding to given **maxterms**
3. Group the 0's
    - Form groups of 1, 2, 4, 8… (powers of 2)
4. Make largest possible groups
5. Follow Grouping Rules
    - Groups must be rectangular
    - Groups can overlap
    - Don't group diagonally
    - Wrap-around allowed (edges are adjacent)
    - Every 0 must be included
6. Write the Sum Terms (OR terms)

    For each group:

    - Include only variables that are constant (same)
    - If variable = 0 → write normal variable

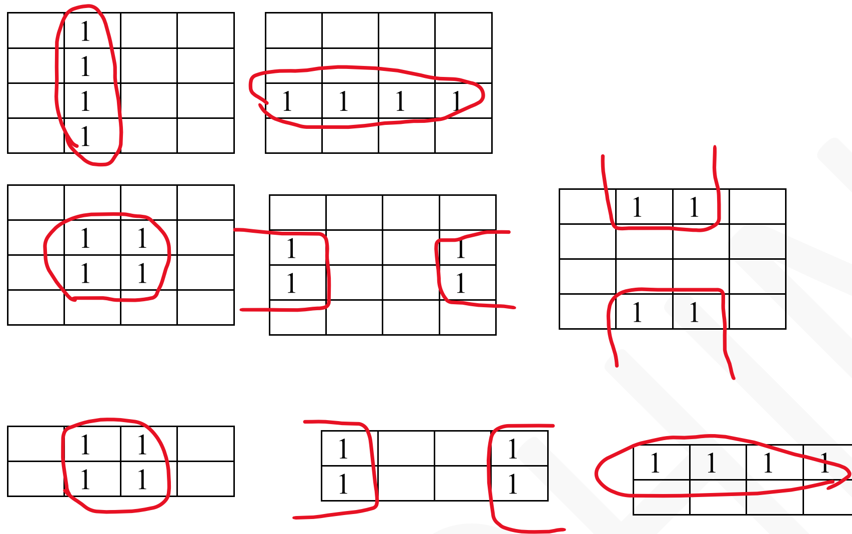
- If variable = 1 → write complement

DO (Correct Grouping Rules)

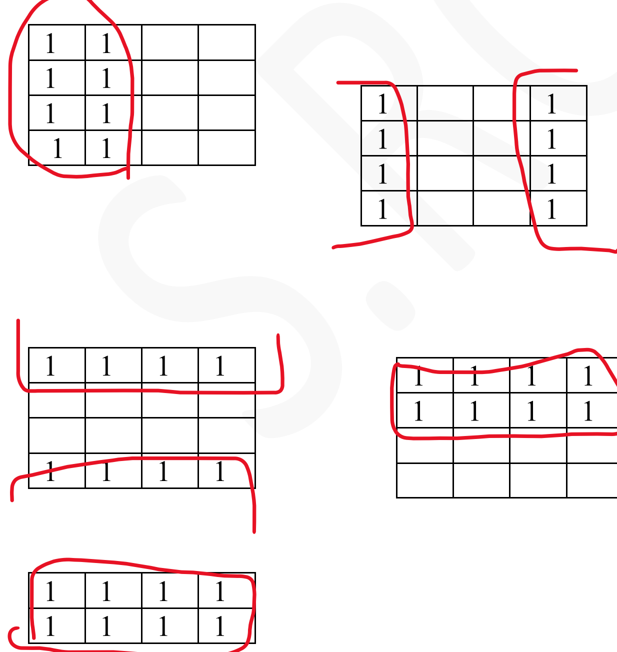
✔ 1. Group in powers of 2

Allowed sizes: **1, 2, 4, 8, 16**

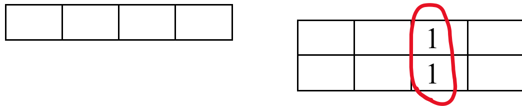
Example:1 → Group of 4 → 4 variable & 3 Variable K – Map (Make the largest possible group- one group of 4 or 8)

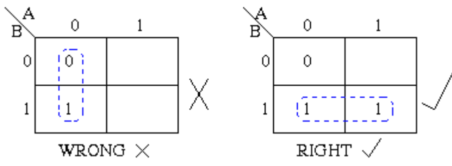


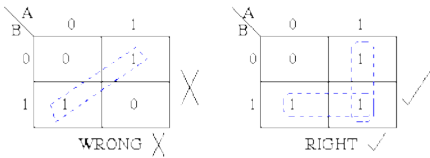
Example 2: → Group of 8 → 4 & 3 variable K - Map

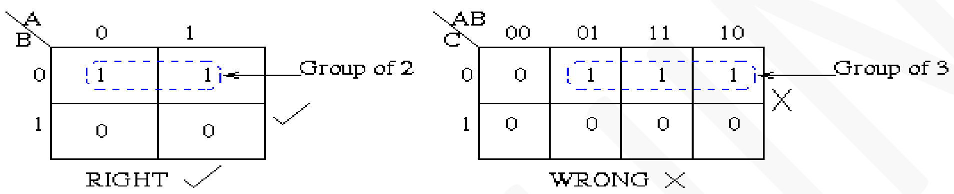


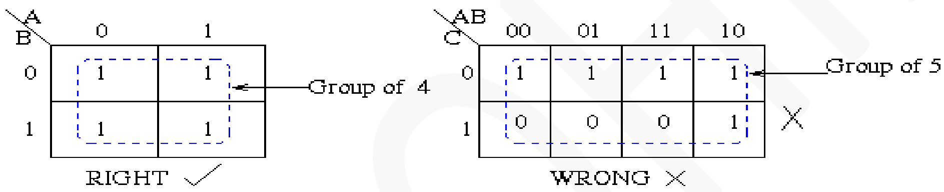
Group only adjacent cells









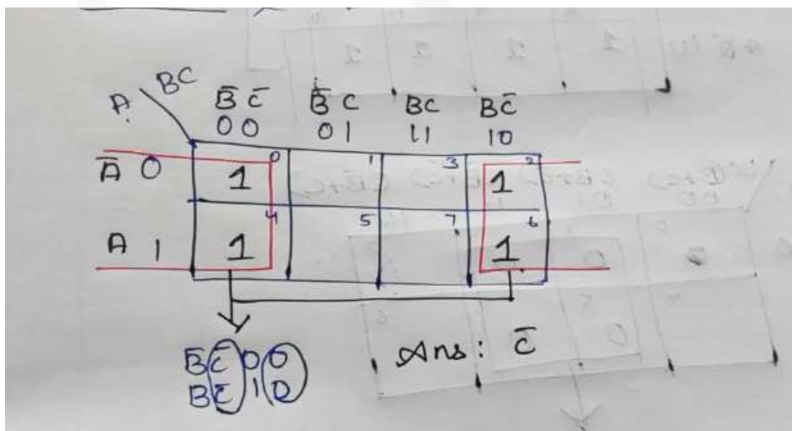


Problems:

1. F(A,B,C)=Σm(0,2,4,6)

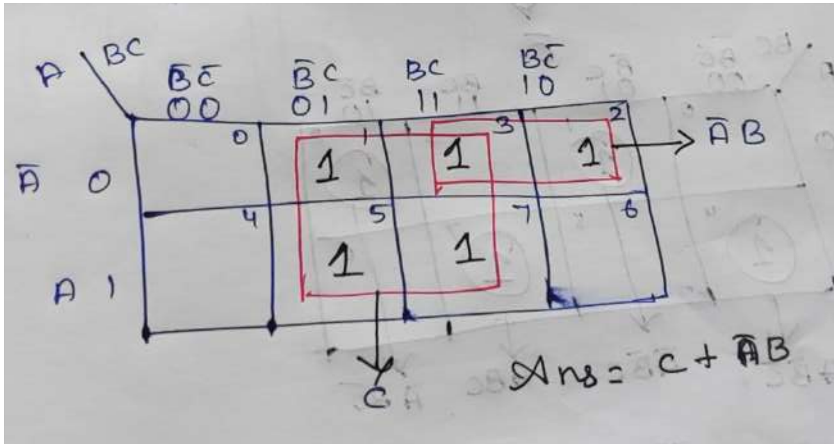
Solution:

3 Variable K map = (A, B, C) = 2^3=8 cells

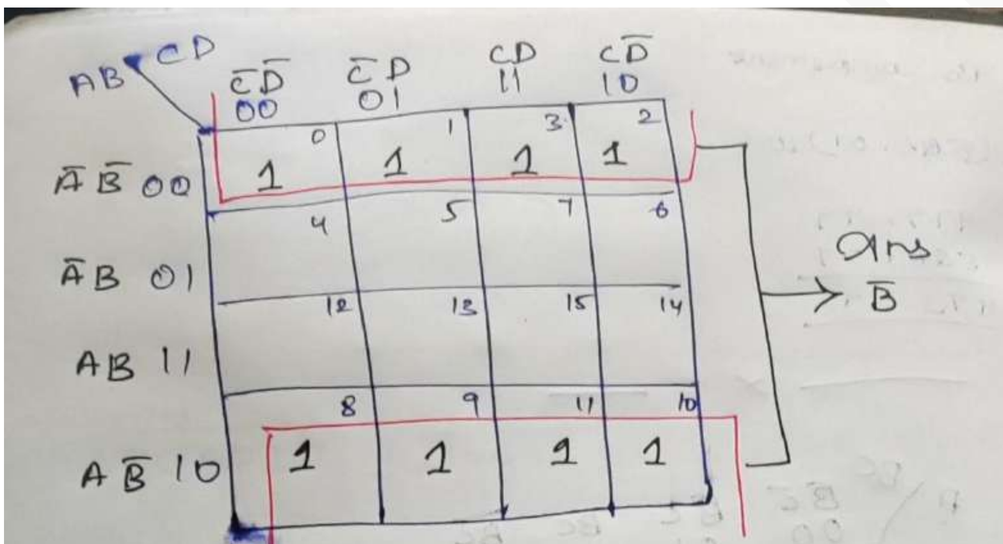


2. F(A,B,C)=Σm(1,3,5,7)

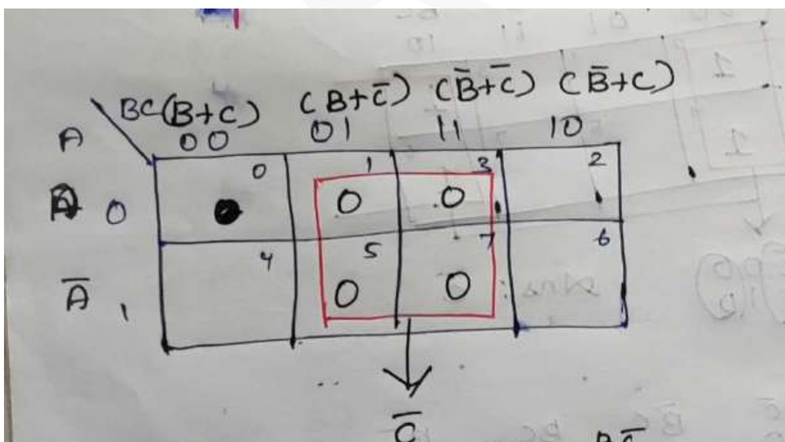
ROHINI



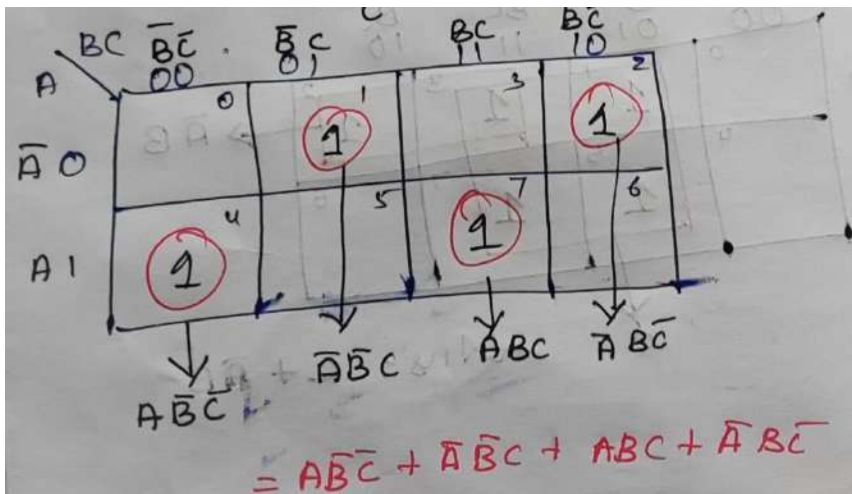
3. F(A,B,C,D)=Σm(0,1,2,3,8,9,10,11)



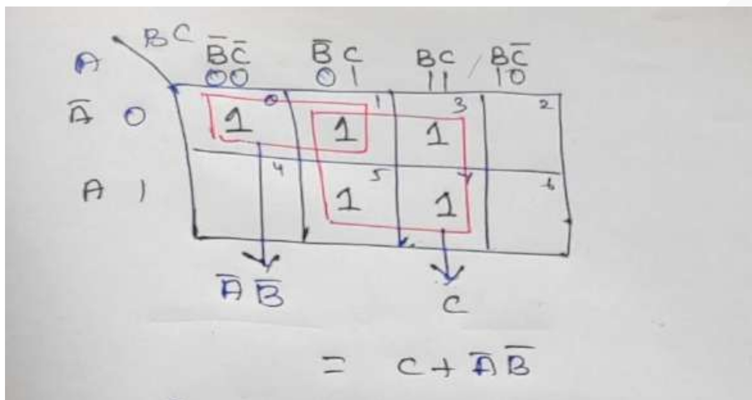
4. F(A,B,C)=ΠM(1,3,5,7)



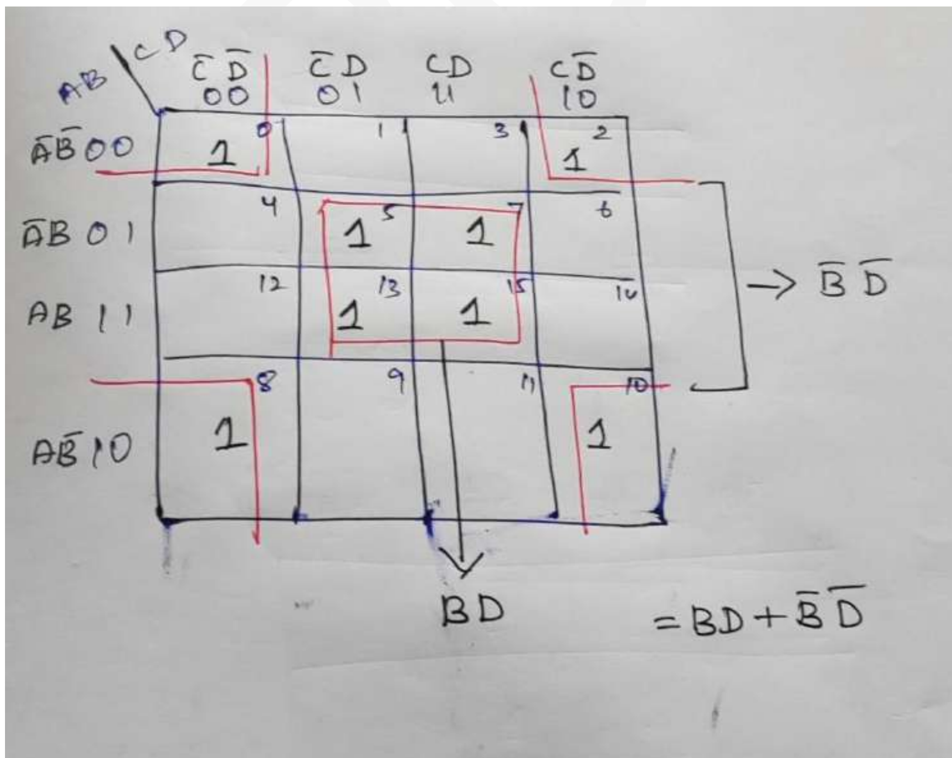
5. F(A,B,C)=Σm(1,2,4,7)



6. F(A,B,C)=Σm(1,3,5,7,0)



7. F(A,B,C,D)=Σm(0,2,5,7,8,10,13,15)



2.2.1.3. Prime–Implicant Method

The Prime–Implicant Method is a **systematic technique** used to simplify Boolean functions by identifying **prime implicants** and selecting the minimum set to cover all minterms.

Key Terms

- **Implicant:** A product term that represents one or more minterms
- **Prime Implicant (PI):** An implicant that cannot be simplified further
- **Essential Prime Implicant (EPI):** A prime implicant that covers at least one minterm not covered by any other PI

Steps Involved

1. List minterms in binary form
2. Group minterms based on number of 1's
3. Combine terms differing by one bit (replace with "–")
4. Repeat until no further combination is possible
5. Identify **Prime Implicants**
6. Select **Essential Prime Implicants** to cover all minterms
7. Write final simplified expression

Examples:

1. Simplify: $F(A, B, C) = \Sigma m(1, 3, 5, 7)$

Step 1: Binary

- 1 → 001
- 3 → 011
- 5 → 101
- 7 → 111

Step 2: Combine (Select differ bit and remove it)

- (001,011) → 0–1 = $A'C$

0	0	1
0	1	1

- (101,111) → 1–1 =**AC**

1	0	1
1	1	1

Step 3: Combine again

S.ROHINI

- (0–1,1–1) → —1 = **C**

0	-	1
1	-	1

Step 4: Final Expression

F = C

2. Given: $F(A,B,C) = \Sigma m(1,2,3,5)$

Step 1: Binary form

- 1 → 001
- 2 → 010
- 3 → 011
- 5 → 101

Step 2: Combine (Select differ bit and remove it)

- (001,011) → 0–1 → **A′C**

0	0	1
0	1	1

- (010,011) → 01– → **A′B**

0	1	0
0	1	1

Step 3: Prime Implicants

- **A′C, A′B**

0	-	1
0	1	-

Step 4: Final Expression

F = A′C + A′B

3. Given: $F(A,B,C) = \Sigma m(1,2,3,5,7)$

Step 1: Binary form

- 1 → 001
- 2 → 010
- 3 → 011
- 5 → 101
- 7 → 111

S.ROHINI

Step 2: Combine (Select differ bit and remove it)

- (001,011) → 0–1 = **A′C**

0	0	1
0	1	1

- (010,011) → 01– = **A′B**

0	1	0
0	1	1

- (101,111) → 1–1 = **AC**

1	0	1
1	1	1

Step 3: Prime Implicants

- 0–1 → $A'C$
- 01– → $A'B$
- 1–1 → AC

0	-	1
0	1	-
1	-	1

Step 4: Final Expression

F=A′C+A′B+AC

4. Given: $F(A,B,C,D) = \Sigma m(0,2,8,10)$

Step 1: Binary form

- 0 → 0000
- 2 → 0010
- 8 → 1000
- 10 → 1010

Step 2: Combine (Select differ bit and remove it)

- (0000,0010) → 00–0

0	0	0	0
0	0	1	0

- (1000,1010) → 10–0

1	0	0	0
1	0	1	0

Step 3: Combine again: Prime Implicants

- (00–0,10–0) → –0–0 = B'D'

S.ROHINI

0	0	-	0
1	0	-	0

Step 4: Final Expression

F=B′D′

2.3. Binary Arithmetic

Binary arithmetic is the process of performing **mathematical operations (addition, subtraction, multiplication, division)** using **binary numbers (0 and 1)**.

2.3.1. Binary Addition

Binary addition is the process of adding binary numbers (0 and 1) using simple rules based on **sum and carry**.

Basic Rules

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Addition with Carry (Important Case)

A	B	Carry-in	Sum	Carry-out
1	1	1	1	1

👉 Because:

$$\mathbf{1 + 1 + 1 = 11}\ (\mathbf{binary})$$

Steps to Perform Binary Addition

1. Start from **rightmost bit (LSB)**
2. Add bits using rules
3. Write **sum** and carry forward
4. Continue till all bits are added

Examples:

1. 1011+1101

```
  1011
+ 1101
---------
 11000
```

2. 111+101

```
   111
 + 101
 ---------
  1100
```

3. Add two binary numbers, 1101 and 1110.

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

4. Add 1010 and 11011.

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

2.3.2 Binary Subtraction

Binary subtraction is the process of subtracting binary numbers using borrow, similar to decimal subtraction.

Basic Rules

A	B	Result	Borrow
0	0	0	0
1	0	1	0
1	1	0	0
0	1	1	1

Examples

1. **Subtract 1100 from 1101**.

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

2. Subtract 101 from 1111.

S.ROHINI

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

3. Subtract 1011 from 1101.

```
   0 10
 1 1 0 1
-1 0 1 1
--------
 0 0 1 0
```

2.3.2.1. Binary Subtraction using two's complement(signed numbers)

2's complement is obtained by **adding 1 to the 1's complement** of a binary number.
Examples:

1. 25−16
Step 1: Convert to binary

$$25 = 11001$$
$$16 = 10000$$

Step 2: Find 2's complement of 16
1's complement:

$$10000 \rightarrow 01111$$

Add 1:

$$01111 + 1 = 10000$$

Step 3: Add
 11001
+10000

 101001
Ignore carry → **01001**
✔ **Answer = 9**

2. −34 − 20
👉 **First convert**:

- $34 = 100010$
- $20 = 010100$

Step 1: 2's complement of 34 (to represent -34)
1's complement:

$$100010 \rightarrow 011101$$

Add 1:

$$011101 + 1 = 011110$$

👉 So, **-34 = 011110**

S.ROHINI

Step 2: Add (-34) + (-20)
Find 2’s complement of 20:

$$010100 \rightarrow 101011 + 1 = 101100$$

Now add:
  011110
+101100

1001010
Ignore carry → **001010**
👉 Take 2’s complement: **001010**

- 1’s → 110101
- +1 → 110110

✔ **Answer = -54**

3. −35+12

Step 1: Convert to binary

$$35 = 100011$$
$$12 = 001100$$

Step 2: 2’s complement of 35
1’s complement:

$$100011 \rightarrow 011100$$

Add 1:

$$011100 + 1 = 011101$$

👉 **-35 = 011101**

Step 3: Add
 011101
+001100

 101001
MSB = 1 → negative result

Step 4: Find magnitude
Take 2’s complement:

- 1’s → 010110
- +1 → 010111

✔ **Answer = -23**

2.3.3. Binary Multiplication

In binary arithmetic, binary multiplication is the process of multiplying two binary numbers and obtain their product.
In binary multiplication, we multiply each bit of one binary number by each bit of another binary number and then add the partial products to obtain the final product.

Rules of Binary Multiplication

SROHINI

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

Examples:

1. Multiply 1101 and 11.

```
     1 1 0 1
   x     1 1
  -----------
   1 1 1 0 1
   1 1 0 1
  -----------
 1 0 0 1 1 1
```

2. Multiply 11011 and 110.

```
        1 1 0 1 1
    x 1 1   1 1 0
   ---------------
      1 0 0 0 0 0
    1 1 1 0 1 1
    1 1 0 1 1
  -----------------
  1 0 1 0 0 0 1 0
```

2.3.4. Binary Division

Binary division is one of the basic arithmetic operations used to find the quotient and remainder when dividing one binary number by another.

Rules of Binary Division

$$0 \div 0 = \text{Undefined}$$

$$0 \div 1 = 0 \text{ with Remainder } = 0$$

$$1 \div 0 = \text{Undefined}$$

$$1 \div 1 = 1 \text{ with Remainder } = 0$$

Binary Division Procedure

- Start dividing from the leftmost bits of the dividend by the divisor.
- Multiply the quotient obtained by the divisor and subtract from the dividend.
- Bring down the next bits of the dividend and repeat the division process until all the bits of given divided are used.

Examples

1. Divide 110011 by 11.

ROHINI

```
       1 0 0 0 1
11 ) 1 1 0 0 1 1
     1 1 ↓ ↓ ↓ ↓
       0 0 0 1 1
             1 1
               0
```

2. Divide 11011 by 10.

```
      1 1 0 1
10 ) 1 1 0 1 1
     1 0 ↓ | |
       1 0 | |
       1 0 ↓ ↓
         0 1 1
           1 0
             1
```

2.4. Arithmetic Building Blocks

Arithmetic Building Blocks are basic combinational circuits used to perform **arithmetic operations** such as **addition and subtraction** in digital systems.

They are the foundation for designing complex units like **ALU (Arithmetic Logic Unit)** in computers.

Main Arithmetic Building Blocks

2.4.1. Half Adder

- A **Half Adder** adds two single-bit binary numbers.
- The half adder provides the output along with a carry value (if any).
- The half adder circuit is designed by connecting an EX-OR gate and one AND gate.
- It has two input terminals and two output terminals for sum and carry.

S.ROHINI

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

Table 3.11.1 Truth table for half-adder

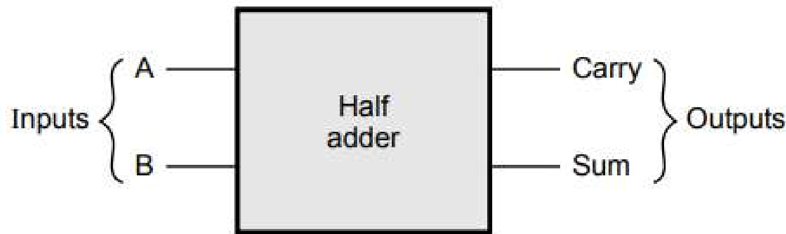


Fig. 3.11.1 Block schematic of half-adder

K-map simplification for carry and sum

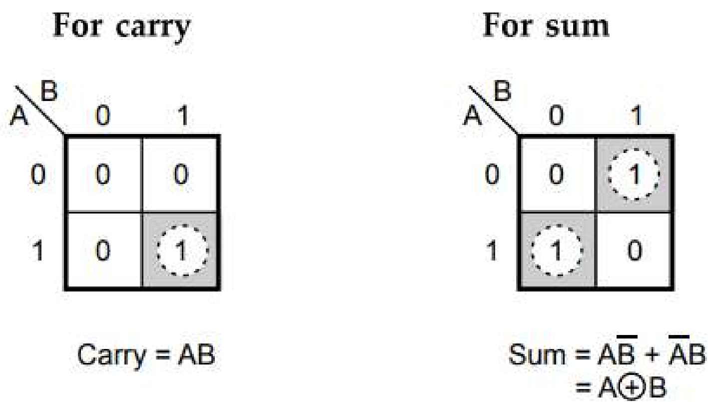


Fig. 3.11.2 Maps for half-adder

Logic diagram

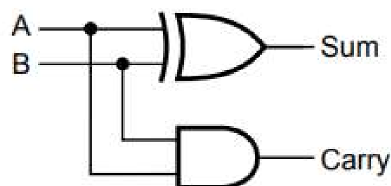


Fig. 3.11.3 Logic diagram for half-adder

2.4.2 Full Adder

- A combinational logic circuit that can add two binary digits (bits) and a carry bit, and produces a sum bit and a carry bit as output is known as a full-adder.

- In other words, a combinational circuit which is designed to add three binary digits and produces two outputs (sum and carry) is known as a full adder.
- Thus, a full adder circuit adds three binary digits, where two are the inputs and one is the carry forwarded from the previous addition.

Inputs			Outputs	
A	**B**	**C_{in}**	**Carry**	**Sum**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3.11.2 Truth table for full-adder

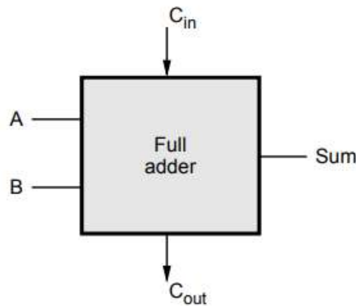


Fig. 3.11.5 Block schematic of full-adder

K-map simplification for carry and sum

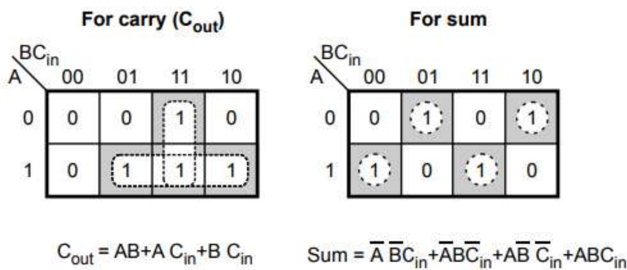


Fig. 3.11.6 Maps for full-adder

$$\text{Sum} = \bar{A}\,\bar{B}\,C_{in} + \bar{A}\,B\,\bar{C}_{in} + A\,\bar{B}\,\bar{C}_{in} + A\,B\,C_{in} = C_{in}\,(\bar{A}\,\bar{B} + AB) + \bar{C}_{in}\,(\bar{A}\,B + A\,\bar{B})$$

$$= C_{in}\,(A \odot B) + \bar{C}_{in}\,(A \oplus B)$$

$$= C_{in}\,(\overline{A \oplus B}) + \bar{C}_{in}\,(A \oplus B)$$

$$= C_{in} \oplus (A \oplus B)$$

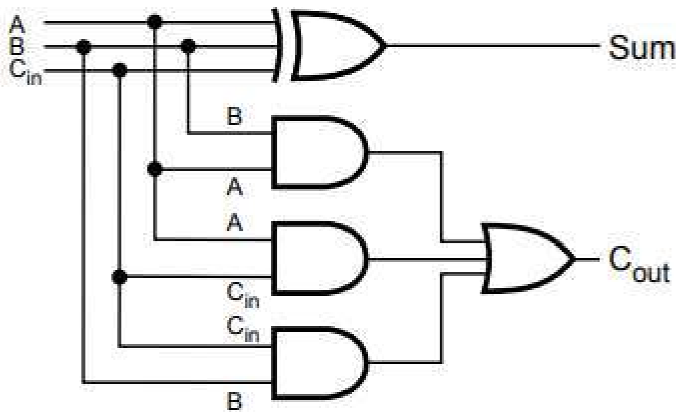


Fig. 3.11.8 Implementation of full-adder

2.4.3 Half Subtractor

- A **half-subtractor** is a combinational logic circuit that have two inputs and two outputs (i.e. difference and borrow).
- The half subtractor produces the difference between the two binary bits at the input and also produces a borrow output (if any).
- In the subtraction (A-B), A is called as **Minuend bit** and B is called as **Subtrahend bit**.

Inputs		Outputs	
A	**B**	**Difference**	**Borrow**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Table 3.12.1 Truth table for half-subtractor

K-map simplification for half-subtractor

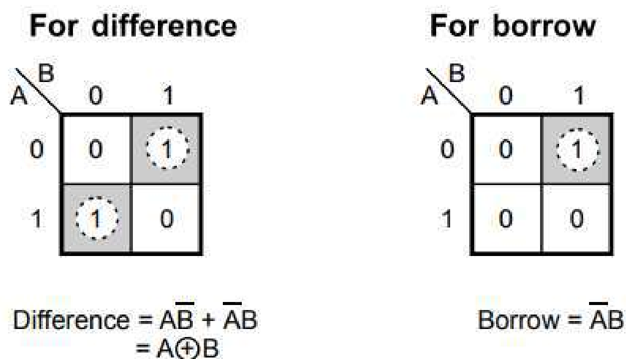


Fig. 3.12.1

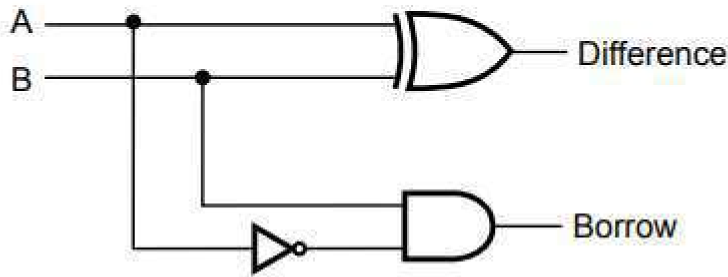


Fig. 3.12.2 Implementation of half-subtractor

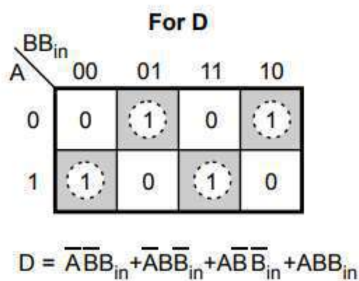
2.3.4. Full-Subtractor

A **full-subtractor** is a combinational circuit that has three inputs A, B, bin and two outputs d and b. Where, A is the minuend, B is subtrahend, b_{in} is borrow produced by the previous stage, d is the difference output and b is the borrow output.

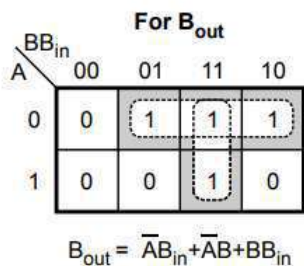
Inputs			Outputs	
A	B	B_{in}	D	B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Table 3.12.2 Truth table for full-subtractor

K-map simplification of D and B_{out}



$D = \bar{A}\bar{B}B_{in} + \bar{A}B\bar{B}_{in} + A\bar{B}\bar{B}_{in} + ABB_{in}$



$B_{out} = \bar{A}B_{in} + \bar{A}B + BB_{in}$

Fig. 3.12.4 Maps for full-subtractor

$$D = \overline{A}\,\overline{B}\,B_{in} + \overline{A}\,B\,\overline{B}_{in} + A\,\overline{B}\,\overline{B}_{in} + A\,B\,B_{in}$$

$$= B_{in}\,(\overline{A}\,\overline{B} + AB) + \overline{B}_{in}\,(\overline{A}\,B + A\,\overline{B})$$

$$= B_{in}\,(A \odot B) + \overline{B}_{in}\,(A \oplus B)$$

$$= B_{in}\,(\overline{A \oplus B}) + \overline{B}_{in}\,(A \oplus B)$$

$$= B_{in} \oplus (A \oplus B)$$

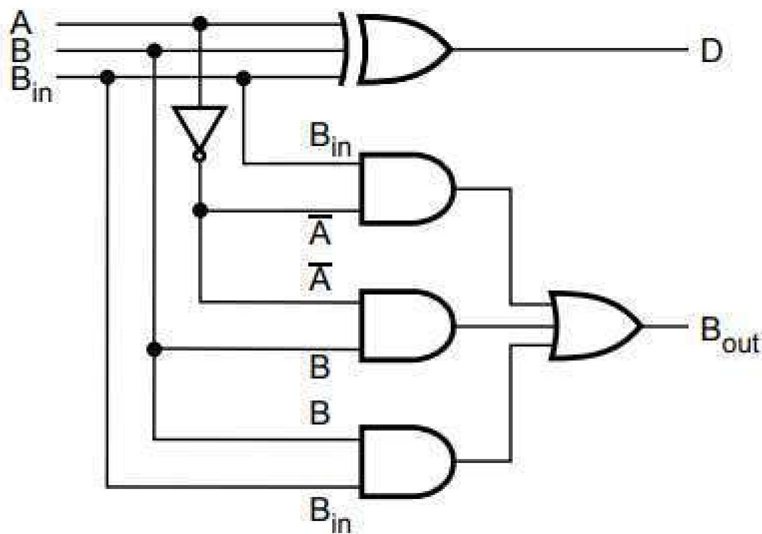


Fig. 3.12.6 Implementation of full-subtractor