Unit 3

Combinational Logic Circuits

Quine-McCluskey minimization technique

Introduction to Combinational Circuits

Multiplexer

Demultiplexer

Decoder

Encoder

Binary adder

Binary adder as subtractor

Carry look ahead adder

Decimal adder

Magnitude Comparator

Problem solving session

Read –only Memory

Arithmetic Logic Unit

Programmable Logic Arrays

HDL Gate and Data Flow modeling

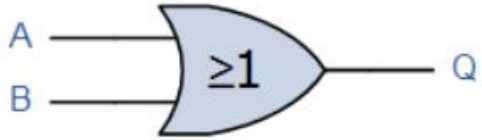
HDL Behavioral modeling

Basic Logic Gates

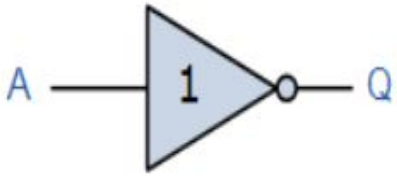
The Logic AND Gate

Symbol	Truth Table		
 2-input AND Gate	A	B	Q
	0	0	0
	0	1	0
	1	0	0
	1	1	1
Boolean Expression Q = A.B	Read as A AND B gives Q		

The Logic OR Gate

Symbol	Truth Table		
 2-input OR Gate	A	B	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	1
Boolean Expression Q = A+B	Read as A OR B gives Q		

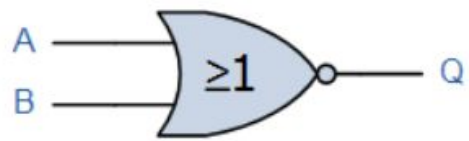
The Logic NOT Gate

Symbol	Truth Table	
 Inverter or NOT Gate	A	Q
	0	1
	1	0
Boolean Expression Q = NOT A or $\overline{A}$	Read as inversion of A gives Q	

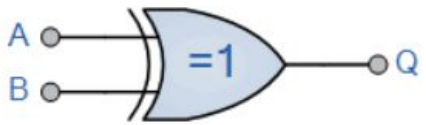
The Logic NAND Gate

Symbol	Truth Table		
 2-input NAND Gate	A	B	Q
	0	0	1
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = $\overline{A.B}$	Read as A AND B gives NOT-Q		

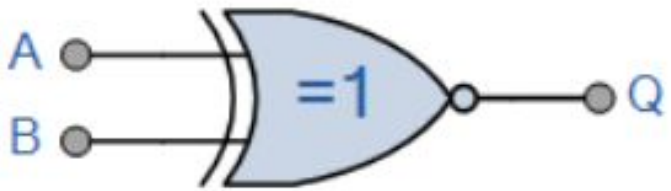
The Logic NOR Gate

Symbol	Truth Table		
 2-input NOR Gate	A	B	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	0
Boolean Expression $Q = \overline{A+B}$	Read as A OR B gives NOT-Q		

The Logic EXOR Gate

Symbol	Truth Table		
	B	A	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0
Boolean Expression $Q = A \oplus B$	Read as A **OR** B but not **BOTH** gives Q (odd)		

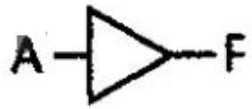
The Logic EXNOR Gate

Symbol	Truth Table		
	B	A	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	1
Boolean Expression $Q = \overline{A \oplus B}$	Read if A **AND** B the **SAME** gives Q (even)		

Consolidation of Logic Gates

Inputs		Truth Table Outputs For Each Gate					
A	B	AND	NAND	OR	NOR	EX-OR	EX-NOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Logic Function	Boolean Notation
AND	$A.B$
OR	$A+B$
NOT	$\overline{A}$
NAND	$\overline{A.B}$
NOR	$\overline{A+B}$
EX-OR	$(A.\overline{B}) + (\overline{A}.B)$ or $A \oplus B$
EX-NOR	$(A.B) + (\overline{A}.\overline{B})$ or $\overline{A \oplus B}$



Buffer
F = A

A	F
0	0
1	1

AND
F = AB

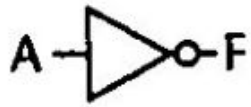
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

OR
F = A+B

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

XOR
F = A⊕B

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0



Inverter
$F = \overline{A}$

A	F
0	1
1	0

NAND
$F = \overline{AB}$

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

NOR
$F = \overline{A+B}$

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

XNOR
$F = \overline{A \oplus B}$

A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

Boolean Rules and Laws

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

Sum of Product (SOP)
and
Product of Sum (POS)

Minterm and Maxterm

A **minterm** is defined as the product term of n variables, in which each of the n variables will appear once either in its complemented or un-complemented form.

A **maxterm** is defined as the sum term of n variables, in which each of the n variables will appear once either in its complemented or un-complemented form.

Variables			Min terms	Max terms
A	B	C	m_i	M_i
0	0	0	A' B' C' = m 0	A + B + C = M 0
0	0	1	A' B' C = m 1	A + B + C' = M 1
0	1	0	A' B C' = m 2	A + B' + C = M 2
0	1	1	A' B C = m 3	A + B' + C' = M 3
1	0	0	A B' C' = m 4	A' + B + C = M 4
1	0	1	A B' C = m 5	A' + B + C' = M 5
1	1	0	A B C' = m 6	A' + B' + C = M 6
1	1	1	A B C = m 7	A' + B' + C' = M 7

Sum of Product (SOP) Form

The Sum of Product form is a form of expression in Boolean algebra in which different product terms of inputs are being summed together.

Example:

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

$$F = \sum(m_1, m_2, m_3, m_5)$$

$$F = \bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}C$$

Canonical or Standard SOP form

Product of Sum (POS) Form

The Product of Sum form is a form in which products of different sum terms of inputs are taken.
Example:

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

$$F = \prod (M_0, M_4, M_6, M_7)$$

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

Canonical or Standard POS form

Conversion from Minimal SOP to Canonical SOP Form

Example:

$$F = \bar{A}B + \bar{B}C$$

The term $\bar{A}B$ is missing input C. So we will multiply $\bar{A}B$ with $(C+\bar{C})$ because $(C+\bar{C} = 1)$.
The term $\bar{B}C$ is missing input A. so it will be multiplied with $(A+\bar{A})$

Missing input variables need to be multiplied

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

$$F = \bar{A}BC + \bar{A}B\bar{C} + A\bar{B}C + \bar{A}\bar{B}C$$

Conversion from Minimal POS to Canonical POS Form

Example:

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

$(\bar{A}+\bar{B})$ term is missing C input so we will add $(C\bar{C})$ with it.
$(B+C)$ term is missing A input so we will add $(A\bar{A})$ with it.

Missing input variables need to be added

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

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

Conversion of SOP to POS

To convert the SOP form into POS form, first we should change the Σ to Π and then write the numeric indexes of missing variables of the given Boolean function.

Example:

$$F = \sum_{A, B, C} (0, 2, 3, 5, 7) = A' B' C' + A B' C' + A B' C + ABC' + ABC$$

Note: writing the missing indexes of the terms i.e., 1 - 001, 4 - 100 and 6 - 110.

001 = (A + B + C'), 100 = (A' + B + C), 110 = (A' + B' + C)

Hence,

$$F = \Pi_{A, B, C} (1, 4, 6) = (A + B + C') * (A' + B + C) * (A' + B' + C)$$

Conversion of POS to SOP

To convert the POS form into SOP form, first we should change the Π to Σ and then write the numeric indexes of missing variables of the given Boolean function.

Example:

$$F = \Pi_{A, B, C}(2, 3, 5) = (A+B'+C) * (A+B'+C') * (A'+B+C')$$

Note: writing the missing indexes of the terms, 0 - 000, 1 - 001, 4 - 100, 6 - 110, and 7 - 111.

000 = A' * B' * C', 001 = A' * B' * C, 100 = A * B' * C', 110 = A * B* C', 111 = A * B * C

Hence,

$$F = \Sigma_{A, B, C}(0, 1, 4, 6, 7) = (A' * B' * C') + (A' * B' * C) + (A * B' * C') +$$

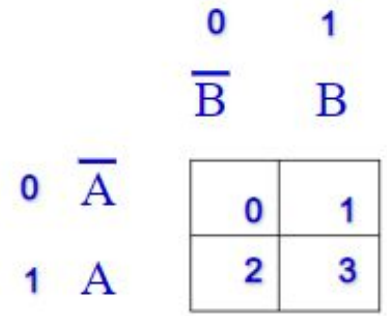
$$(A * B* C') + (A * B * C)$$

Karnaugh Map (K-Map)

K-map

- K-Map is used to minimize the number of logic gates by minimizing the logical expression.
- The minimization will reduce cost, complexity and power consumption.
- An n-variable K-map has 2^n cells with each cell corresponding to an n-variable truth table value.
  - If 2 variable – 4 cells
  - If 3 variable – 8 cells
  - If 4 variable – 16 cells
- K-map cells are arranged such that adjacent cells correspond to truth rows that differ in only one bit position.

2 variables K-Map



3 variables K-Map

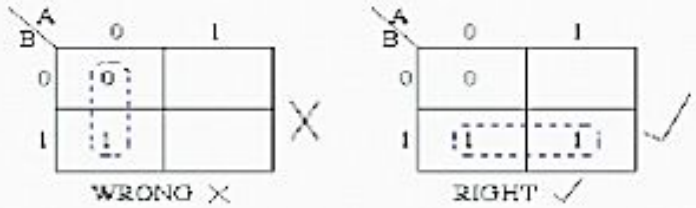
	x \ yz	00 y'z'	01 y'z	11 yz	10 yz'
0	x'	m0 x'y'z' 0	m1 x'y'z 1	m3 x'yz 3	m2 x'yz' 2
1	x	m4 xy'z' 4	m5 xy'z 5	m7 xyz 7	m6 xyz' 6

4 variables K-Map

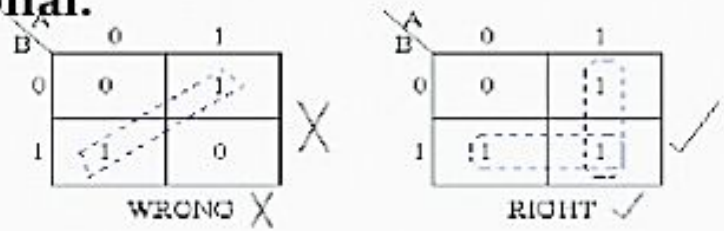
		00	01	11	10
		$\overline{C}\,\overline{D}$	$\overline{C}\,D$	$C\,D$	$C\,\overline{D}$
00	$\overline{A}\,\overline{B}$	0	1	3	2
01	$\overline{A}\,B$	4	5	7	6
11	$A\,B$	12	13	15	14
10	$A\,\overline{B}$	8	9	11	10

K-map Rules

1.Groups may not include any cell containing a zero.

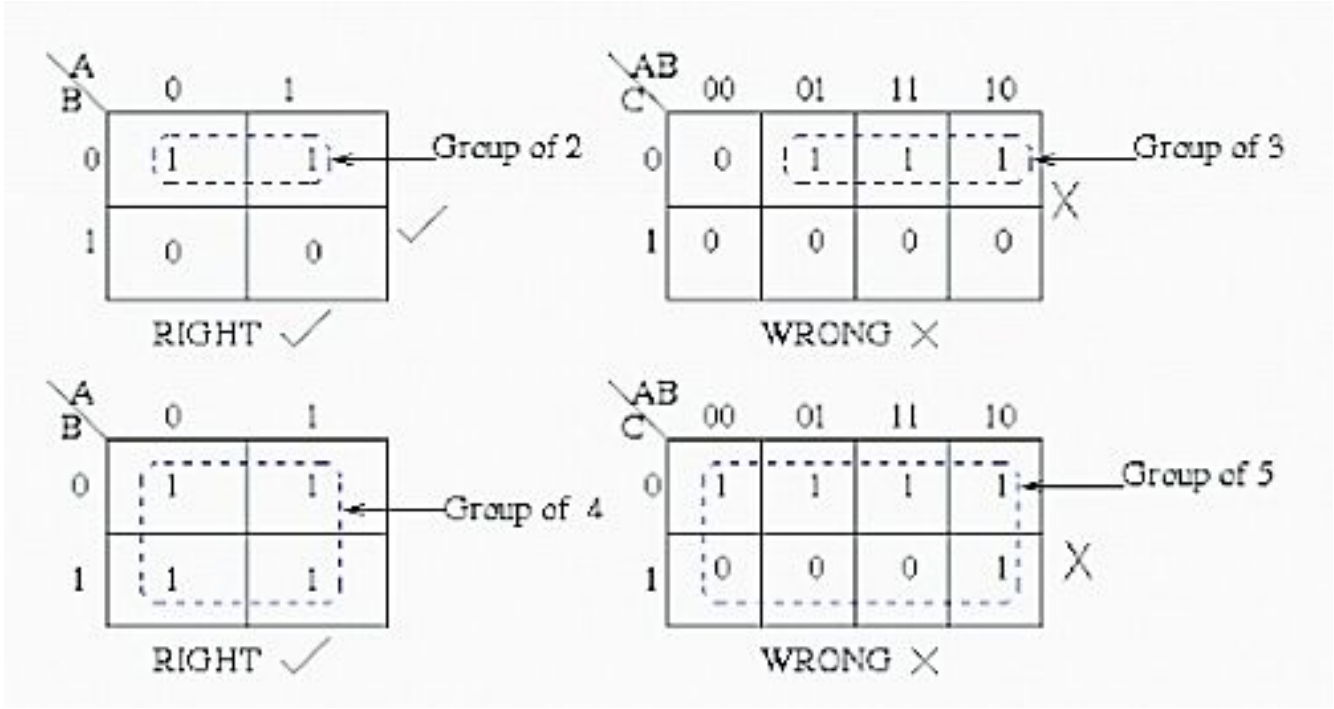


2.Groups may be horizontal or vertical, but not diagonal.



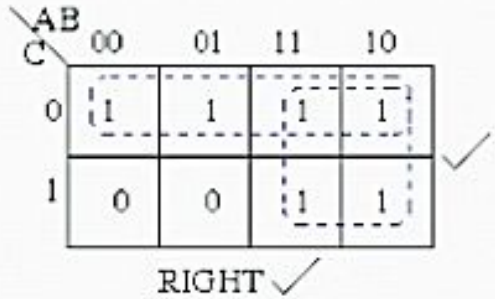
K-map Rules

3. Groups must contain 1, 2, 4, 8, or in general 2^n cells.

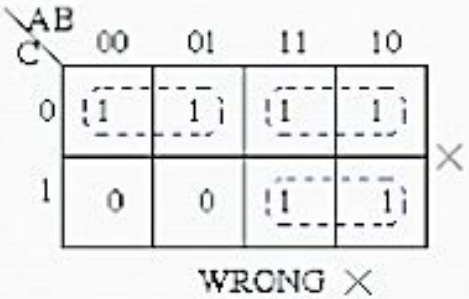


K-map Rules

4. Each group should be as large as possible.



RIGHT ✓

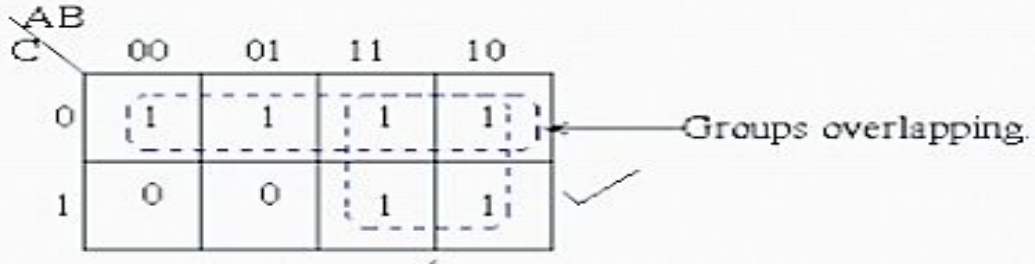


WRONG ×

(Note that no Boolean laws broken, but not sufficiently minimal)

5.Each cell containing a one must be in at least one group.

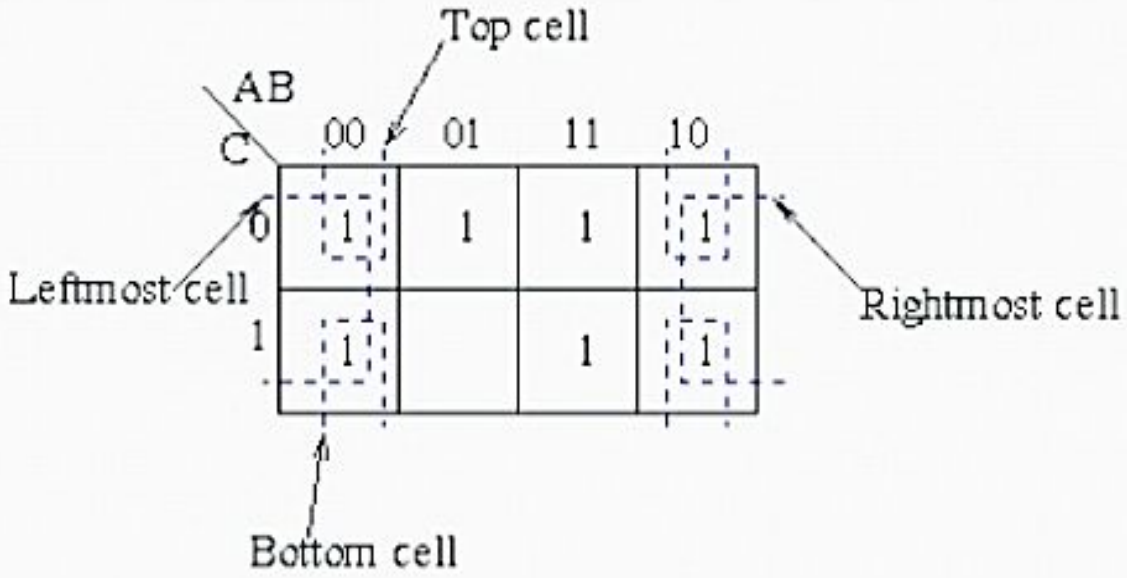
6.Groups may overlap.



K-map Rules

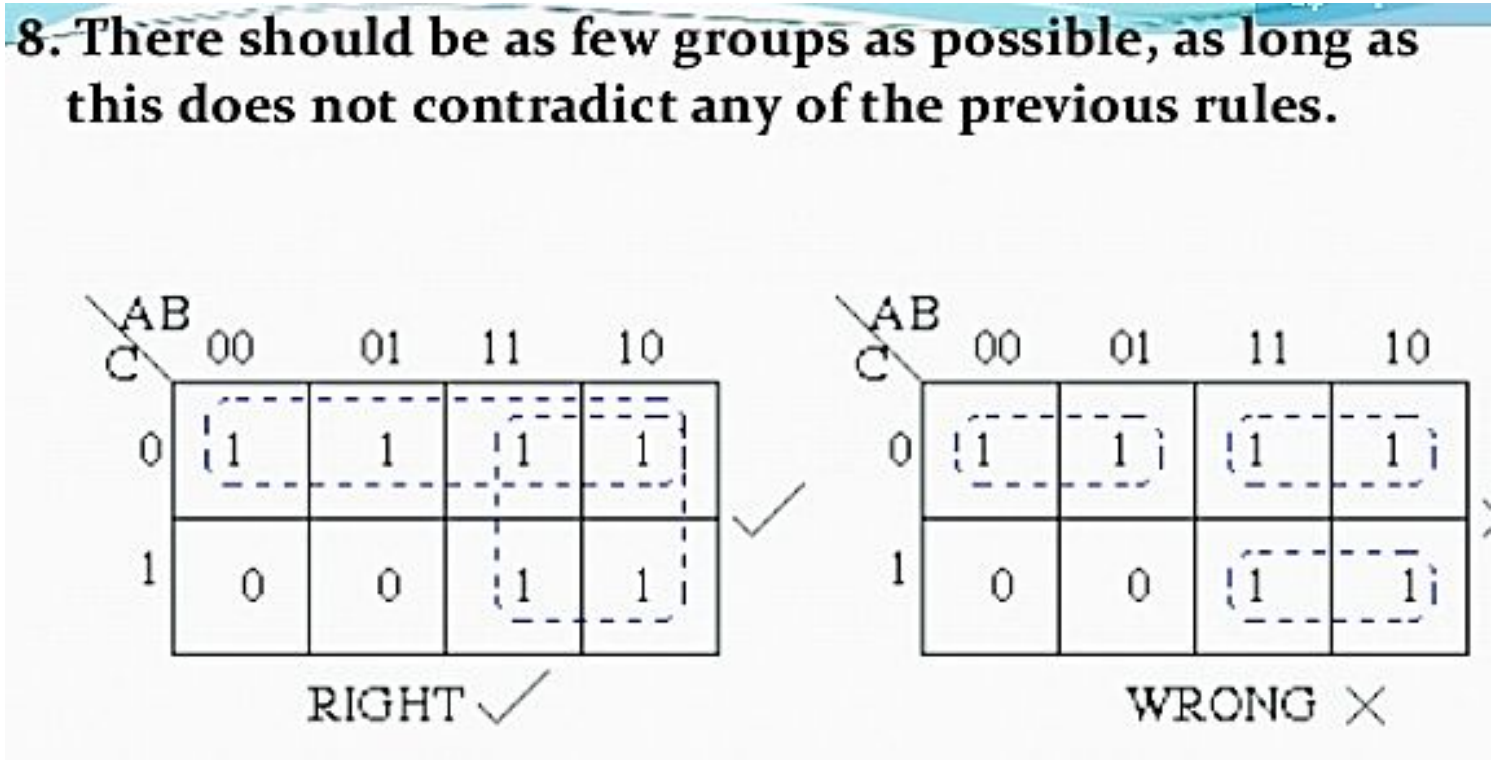
7.Groups may wrap around the table.

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.



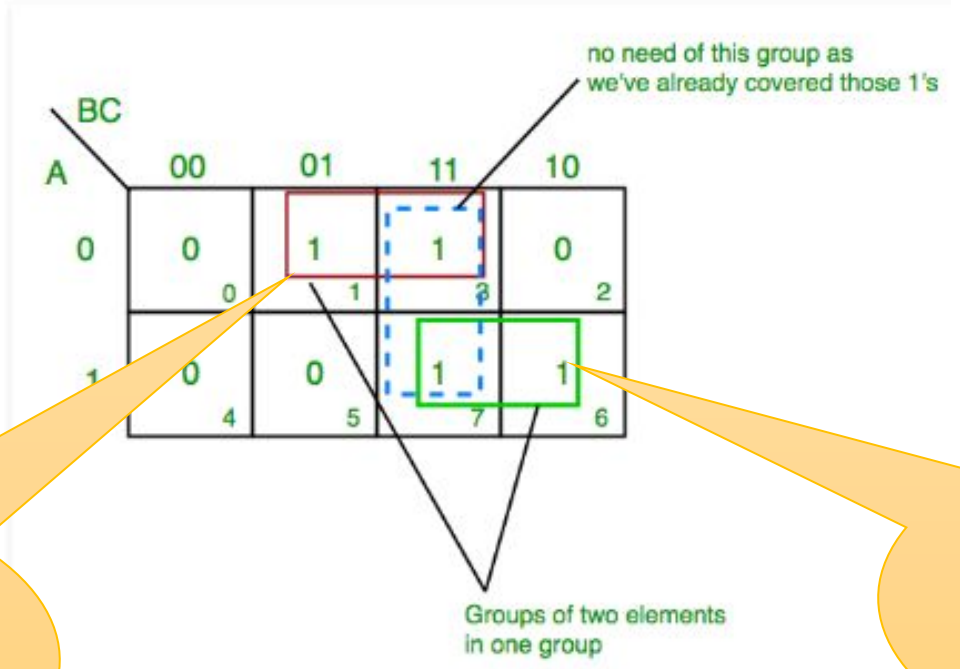
K-map Rules

8. There should be as few groups as possible, as long as this does not contradict any of the previous rules.



Example 1

For the given minterms, find the reduced logical expression using K-Map $Z = \sum_{A,B,C}(1,3,6,7)$



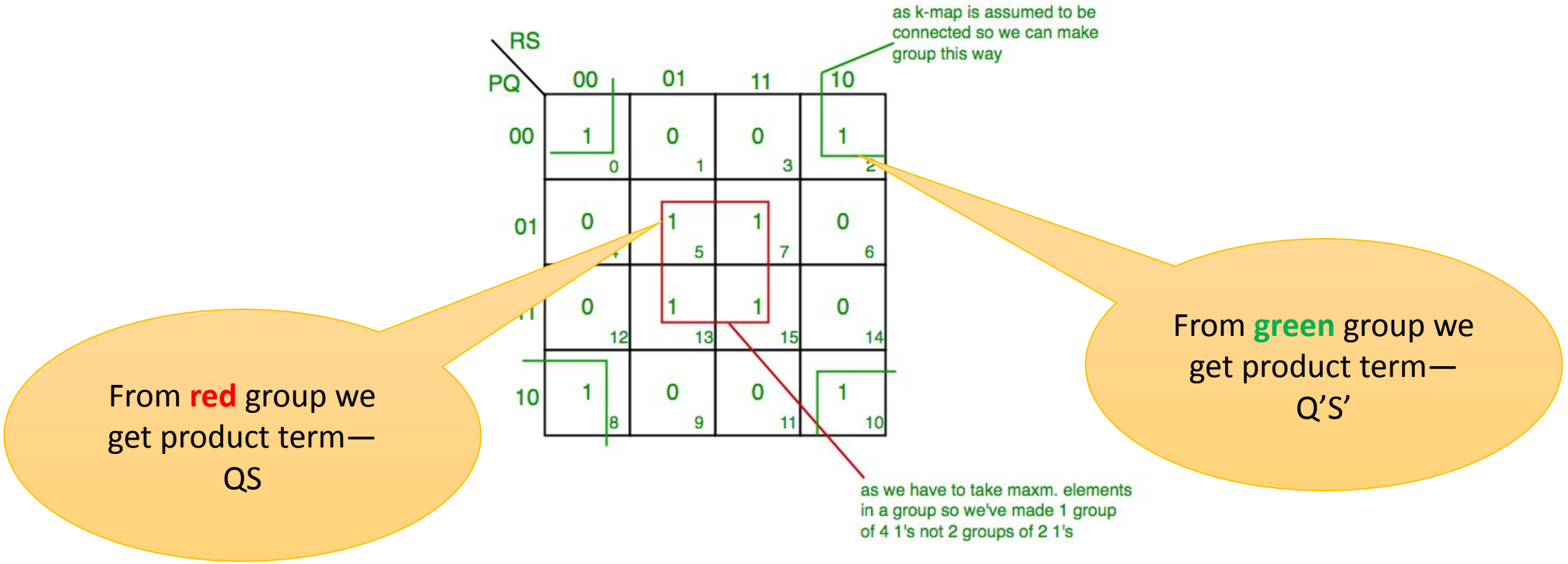
From **red** group we get product term— A'C

From **green** group we get product term— AB

Z= A'C+AB

Example 2

For the given minterms, find the reduced logical expression using K-Map **F(P,Q,R,S)=∑(0,2,5,7,8,10,13,15)**



F = QS+Q'S'

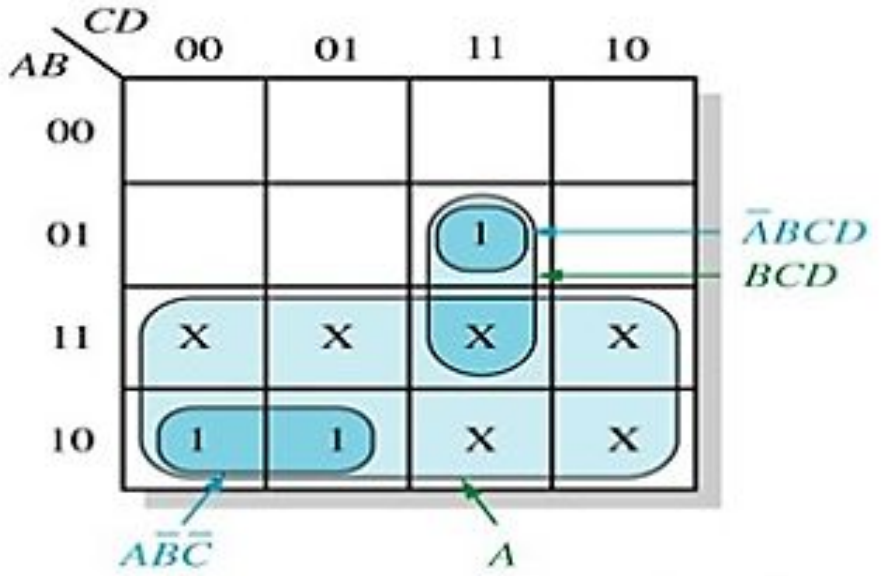
K-map with Don’t cares

- A “don’t care” condition is a combination of inputs for which the output may either ‘1’ or ‘0’.
- Hence “don’t care” conditions may either be included or excluded while grouping is made.
- It is usually denoted as ‘X’ in the K-Map table.

Example 3

For the given minterms, find the reduced logical expression using K-Map

$Z = \sum_{A,B,C}(7, 8, 9) + \sum_{d}(10, 11, 12, 13, 14, 15)$



Z = AB'C'+A'BCD (without don't cares)

✔ **Z = A+BCD (with don't cares)**

Quine-McCluskey Method (Tabulation Method)

- A systematic simplification procedure to reduce a minterm expansion to a minimum sum of products.
- Use XY + XY’ = X to eliminate as many as literals as possible.
  - The resulting terms = prime implicants.
- Use a prime implicant chart to select a minimum set of prime implicants.

Determination of Prime Implicants

√ Eliminate literals

Two terms can be combined if they differ in exactly one variable.

AB'CD' + AB'CD = AB'C

1 0 1 0 + 1 0 1 1 = 1 0 1

X Y X Y' X

A'BC'D + A'BCD' (won't combine)

0 1 0 1 + 0 1 1 0 (check # of 1's)

We need to compare and combine whenever possible.

Sorting to Reduce Comparisons

√ Sort into groups according to the number

F(a,b,c.d) = Σm(0,1,2,5,6,7,8,9,10,14)

- No need for comparisons
  - (1) Terms in nonadjacent group
  - (2) Terms in the same group

Group	0	0	0000
Group	1	1	0001
		2	0010
		8	1000
Group	2	5	0101
		6	0110
		9	1001
		10	1010
Group	3	7	0111
		14	1110

Comparison of adjacent groups

Use X + X = X repeatedly between adjacent groups

Those combined are checked off.

Combine terms that have the same dashes and differ one in the number of 1's. (for column II and column III)

Determination of Prime Implicants

	Column I		Column II		Column III	
group 0	0	0000 ✓	0, 1	000- ✓	0, 1, 8, 9	-00-
			0, 2	00-0 ✓	0, 2, 8, 10	-0-0
group 1	1	0001 ✓	0, 8	-000 ✓	~~0, 8, 1, 9~~	~~-00-~~
	2	0010 ✓			~~0, 8, 2, 10~~	~~-0-0~~
	8	1000 ✓	1, 5	0-01		
			1, 9	-001 ✓	2, 6, 10, 14	--10
group 2	5	0101 ✓	2, 6	0-10 ✓	~~2, 10, 6, 14~~	~~--10~~
	6	0110 ✓	2, 10	-010 ✓		
	9	1001 ✓	8, 9	100- ✓		
	10	1010 ✓	8, 10	10-0 ✓		
			5, 7	01-1		
group 3	7	0111 ✓	6, 7	011-		
	14	1110 ✓	6, 14	-110 ✓		
			10, 14	1-10 ✓		

Prime Implicants

- The terms that have not been checked off are called prime implicants.

  f = 0-01 + 01-1+011- + -00-

  + -0-0 + --10

  = a'c'd + a'bd + a'bc + b'c' +

  b'd' + cd'

- Each term has a minimum number of literals, but minimum SOP for f:

  f = a'bd + b'c' + cd'

  (a'bd, cd' => a'bc)

  (a'bd, b'c' => a'c'd)

  (b'c', cd' => b'd')

Definition of Implicant

- Definition
  - Given a function of F of n variables, a product term P is an implicant of F iff for every combination of values of the n variables for which P = 1, F is also equal to 1.
    - Every minterm of F is an implicant of F.
    - Any term formed by combining two or more minterms is an implicant.
    - If F is written in SOP form, every product term is an implicant.

- Example: f(a,b,c) = a'b'c' + ab'c' + ab'c + abc = b'c' + ac
  - If a'b'c' = 1, then F = 1, if ac = 1, then F = 1. a'b'c' and ac are implicants.
  - If bc = 1, (but a = 0), F = 0, so bc is not an implicant of F.

Definition of Prime Implicant

- Definition
  - A prime implicant of a function F is a product term implicant which is no longer an implicant if any literal is deleted from it.
  - Example: f(a,b,c) = a'b'c' + ab'c' + ab'c + abc = b'c' + ac
    - Implicant a'b'c' is not a prime implicant. Why? If a' is deleted, b'c' is still an implicant of F.
    - b'c' and ac are prime implicants.

- Each prime implicant of a function has a minimum number of literals that no more literals can be eliminated from it or by combining it with other terms.

Quine McClusky Procedure

- QM procedure:
  - Find all product term implicants of a function
  - Combine non-prime implicants.
    - Remaining terms are prime implicants.
  - A minimum SOP expression consists of a sum of some (not necessarily all) of the prime implicants of that function.
    - We need to select a minimum set of prime implicants.
  - If an SOP expression contains a term which is not a prime implicant, the SOP cannot be minimum

Prime Implicant Chart

- Chart layout
  - Top row lists minterms of the function
  - All prime implicants are listed on the left side.
  - Place x into the chart according to the minterms that form the corresponding prime implicant.
- Essential prime implicant
  - If a minterm is covered only by one prime implicant, that prime implicant is called essential prime implicant. (9 & 14).
    - Essential prime implicant must be included in the minimum sum of the function.

		0	1	2	5	6	7	8	9	10	14
(0, 1, 8, 9)	$b'c'$	X	X					X	(X)		
(0, 2, 8, 10)	$b'd'$	X		X				X		X	
(2, 6, 10, 14)	cd'			X		X				X	(X)
(1, 5)	$a'c'd$		X		X						
(5, 7)	$a'bd$				X		X				
(6, 7)	$a'bc$					X	X				

First List

	A	B	C	D	
0	0	0	0	0	✓
1	0	0	0	1	✓
2	0	0	1	0	✓
8	1	0	0	0	✓
3	0	0	1	1	✓
5	0	1	0	1	✓
10	1	0	1	0	✓
12	1	1	0	0	✓
7	0	1	1	1	✓
13	1	1	1	1	✓
15	1	1	1	1	✓

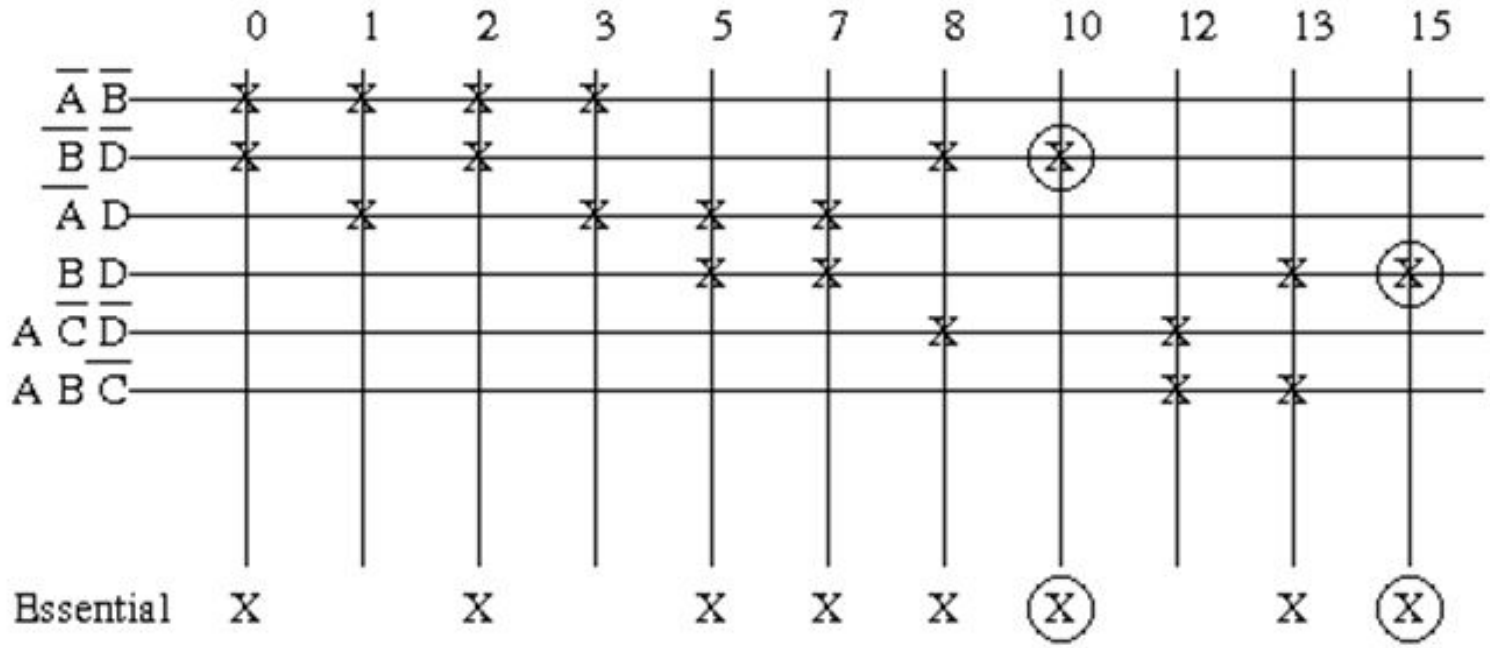
Second List

	A	B	C	D	
0,1	0	0	0	–	✓
0,2	0	0	–	0	✓
0,8	–	0	0	0	✓
1,3	0	0	–	1	✓
1,5	0	–	0	1	✓
2,3	0	0	1	–	✓
2,10	–	0	1	0	✓
8,10	1	0	–	0	✓
8,12	1	–	0	0	$A\overline{C}\overline{D}$
3,7	0	–	1	1	✓
5,7	0	1	–	1	✓
5,13	–	1	0	1	✓
12,13	1	1	0	–	$AB\overline{C}$
7,15	–	1	1	1	✓
13,15	1	1	–	1	✓

Third List

	A	B	C	D	
0, 1 2 3	0	0	–	–	$\overline{A}\,\overline{B}$
0, 2 1 3	0	0	–	–	
0, 2, 8, 10	–	0	–	0	$\overline{B}\,\overline{D}$
0, 8, 2, 10	–	0	–	0	
1, 3, 5, 7	0	–	–	1	$\overline{A}D$
1, 5, 3, 7	0	–	–	1	
5, 7, 13, 15	–	1	–	1	BD
5, 13, 7, 15	–	1	–	1	

The prime implicants are: $\overline{A}\,\overline{B} + \overline{B}\,\overline{D} + \overline{A}D + BD + A\overline{C}\,\overline{D} + AB\overline{C}$



Thus, one minimal solution is: $Z = \overline{B}\,\overline{D} + BD + \overline{A}\,\overline{B} + A\overline{C}\,\overline{D}$

$F(A,B,C,D) = \Sigma m(2,6,8,9,10,1,14,15)$

Group Name	Min terms	W	X	Y	Z
GA1	2	0	0	1	0
	8	1	0	0	0
GA2	6	0	1	1	0
	9	1	0	0	1
	10	1	0	1	0
GA3	11	1	0	1	1
	14	1	1	1	0
GA4	15	1	1	1	1

Group Name	Min terms	W	X	Y	Z
GB1	2,6	0	-	1	0
	2,10	-	0	1	0
	8,9	1	0	0	-
	8,10	1	0	-	0
GB2	6,14	-	1	1	0
	9,11	1	0	-	1
	10,11	1	0	1	-
	10,14	1	-	1	0
GB3	11,15	1	-	1	1
	14,15	1	1	1	-

Group Name	Min terms	W	X	Y	Z
GB1	2,6,10,14	-	-	1	0
	2,10,6,14	-	-	1	0
	8,9,10,11	1	0	-	-
	8,10,9,11	1	0	-	-
GB2	10,11,14,15	1	-	1	-
	10,14,11,15	1	-	1	-

Group Name	Min terms	W	X	Y	Z
GC1	2,6,10,14	-	-	1	0
	8,9,10,11	1	0	-	-
GC2	10,11,14,15	1	-	1	-

Min terms / Prime Implicants	2	6	8	9	10	11	14	15
YZ’	1	1			1		1	
WX’			1	1	1	1		
WY					1	1	1	1

f(W,X,Y,Z) = YZ’ + WX’ + WY.

$F(A,B,C,D) = \Sigma m(0,1,3,7,8,9,11,15)$

Group	Minterms	Variables A B C D	Remark
0	0	0000	✓
1	1	0001	✓
	8	1000	✓
2	3	0011	✓
	9	1001	✓
3	7	0111	✓
	11	1011	✓
4	15	1111	✓

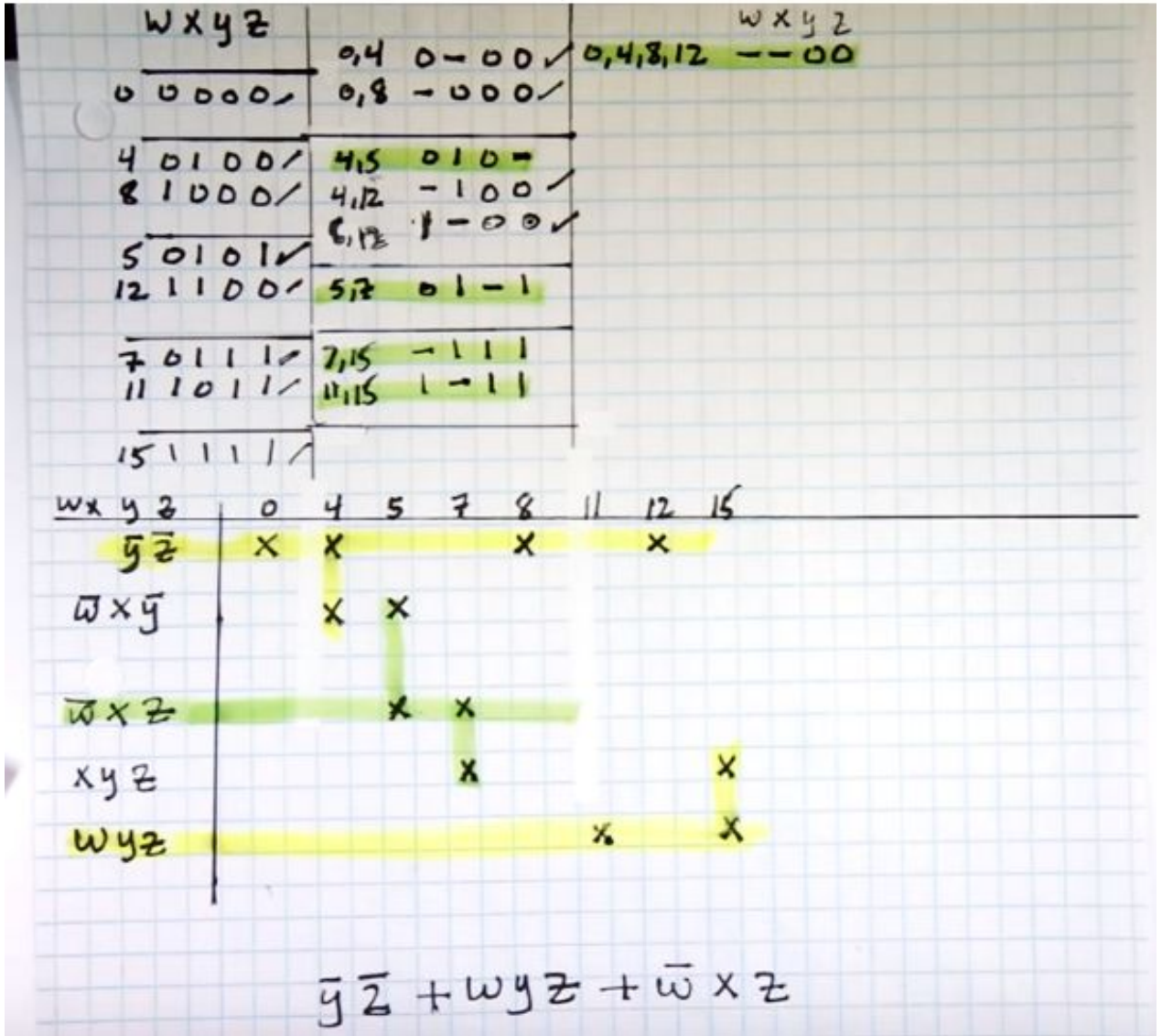
Group	Minterms	Variables A B C D	Remark
0	0,1	000-	✓
	0,8	-000	✓
1	1,3	00-1	✓
	1,9	-001	✓
	8,9	100-	✓
2	3,7	0-11	✓
	3,11	-011	✓
	9,11	10-1	✓
3	7,15	-111	✓
	11,15	1-11	✓

Group	Minterms	Variables ABCD	Remark
0	(0,1),(8,9)	-00-	✓
	(0,8),(1,9)	-00-	✓
1	(1,3),(9,11)	-0-1	✓
	(1,9),(3,11)	-0-1	✓
2	(3,7),(11,15)	--11	✓
	(3,11),(7,15)	--11	✓

PI terms	Group of Minterms	Minterms 0 1 3 7 8 9 11 15
B'C'	0,1,8,9	(X) X (X) X
B'D	1,3,9,11	X X X X
CD	3,7,11,15	X (X) X (X)

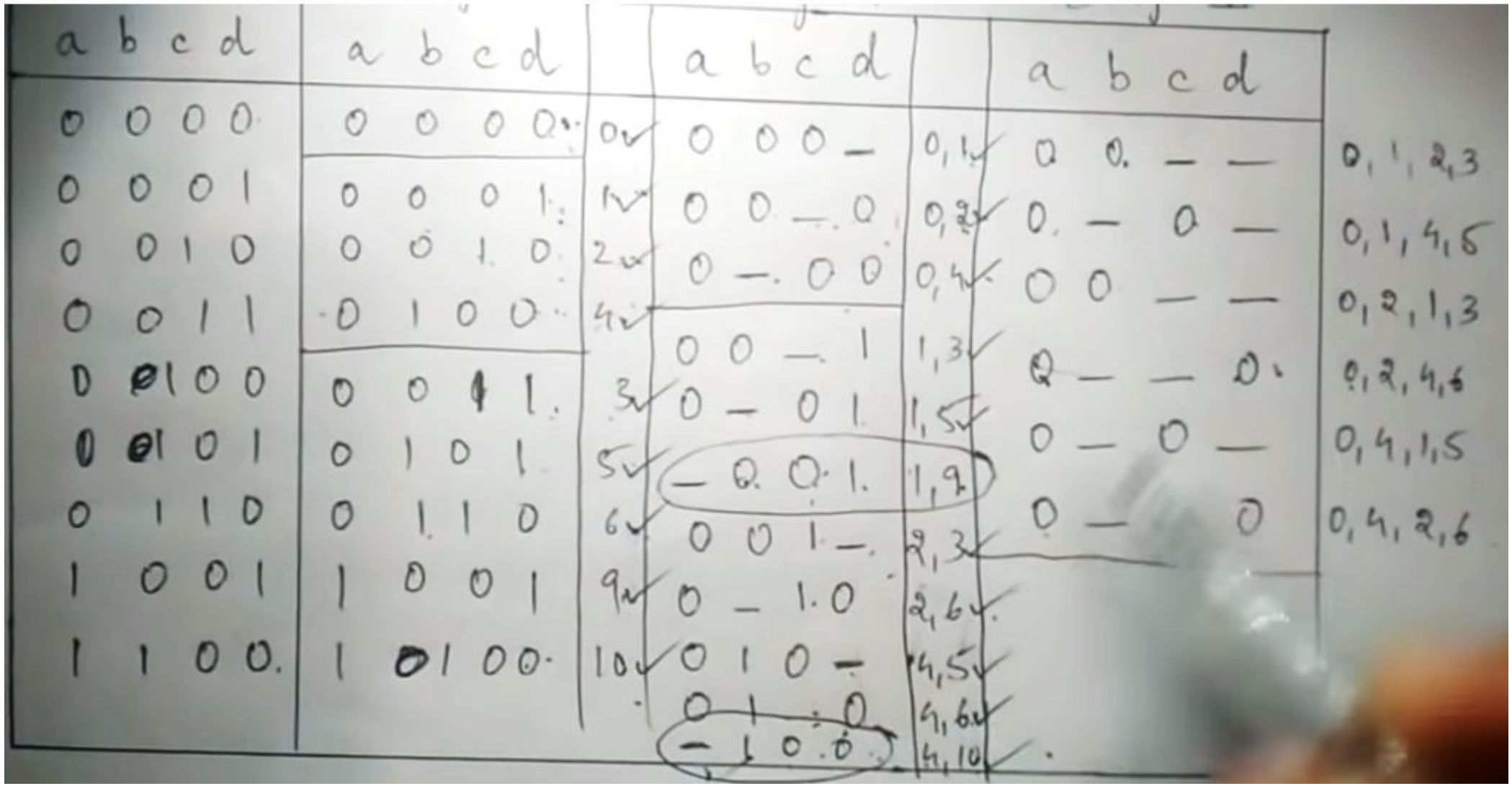
(B'C'+CD).

$F(w,x,y,z) = \Sigma m(0,4,5,7,8,11,12,15)$



•f(a,b,c,d) = Σm(0,1,2,3,9,10) + Σd(4,5,6)

$$f(a,b,c,d) = \sum m(0,1,2,3,9,10) + \sum dc(4,5,6)$$



PI	Minterms	0	1	2	3	9	10
$\bar{b}\bar{c}d$ ✓	1, 9		×			⊗	
$b\bar{c}d$ ✓	4, 10						⊗
$\bar{a}\bar{b}$ ✓	0, 1, 2, 3	×	×	×	⊗		
$\bar{a}\bar{c}$	0, 1, 4, 5	×	×				
$\bar{a}\bar{d}$	0, 2, 4, 6	×		×			
		✓	✓	✓	✓	✓	✓

$$f(a,b,c,d) = \bar{b}\bar{c}d + b\bar{c}d + \bar{a}\bar{b}$$

$$F(A,B,C,D) = \sum m(2,3,7,9,11,13) + \sum d(1,10,15)$$

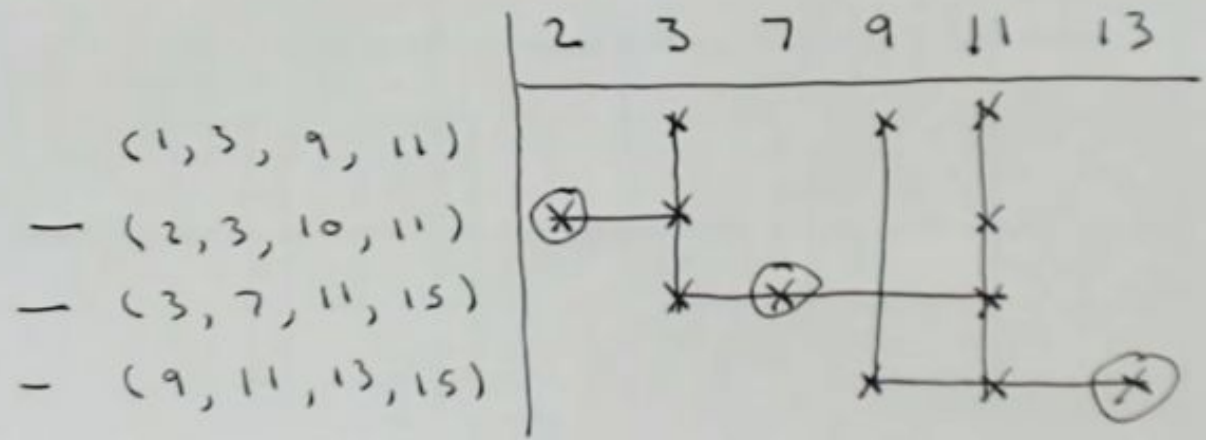
$$F(A,B,C,D) = \sum m(2,3,7,9,11,13) + \sum d(1,10,15)$$

1	0	0	0	1	✓
2	0	0	1	0	✓
3	0	0	1	1	✓
9	1	0	0	1	✓
10	1	0	1	0	✓
7	0	1	1	1	✓
11	1	0	1	1	✓
13	1	1	0	1	✓
15	1	1	1	1	✓

(1,3)	0	0	-	1	✓
(1,9)	-	0	0	1	
(2,3)	0	0	1	-	✓
(2,10)	-	0	1	0	
(3,7)	0	-	1	1	
(3,11)	-	0	1	1	
(9,11)	1	0	-	1	✓
(9,13)	1	-	0	1	
(10,11)	1	0	1	-	✓
(7,15)	-	1	1	1	
(11,15)	1	-	1	1	
(13,15)	1	1	-	1	

(1,3,9,11)	-	0	-	1
(2,3,10,11)	-	0	1	-
(3,7,11,15)	-	-	1	1
(9,11,13,15)	1	-	-	1

	A B C D
(1, 3, 9, 11)	- 0 - 1
(2, 3, 10, 11)	- 0 1 -
(3, 7, 11, 15)	- - 1 1
(9, 11, 13, 15)	1 - - 1

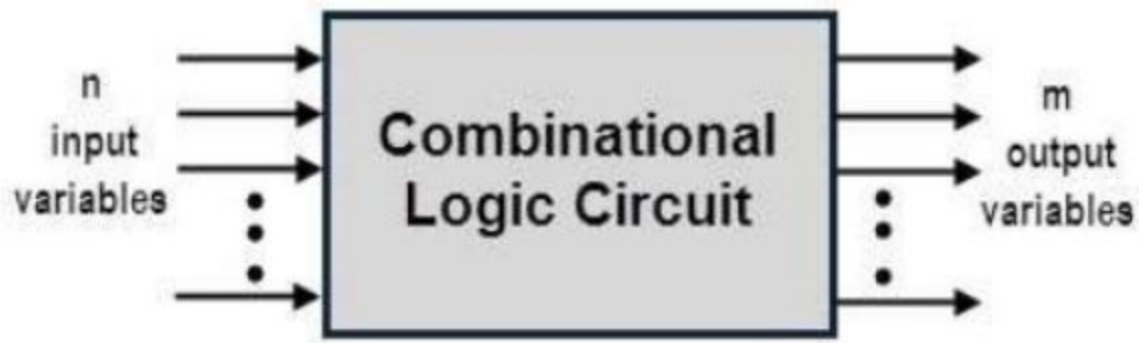


$F(A,B,C,D) : \bar{B}C + CD + AD$

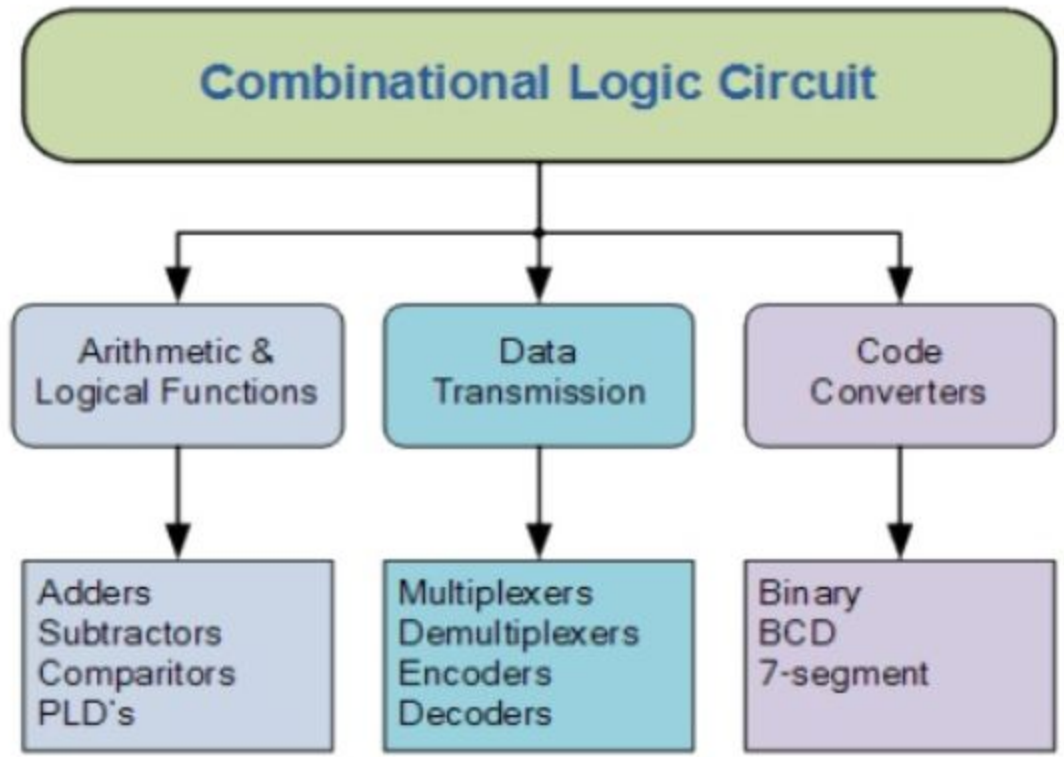
Combinational Circuits

- A combinational logic circuit is one in which the present state of the combination of the logic inputs decides the output .
- The term combination logic means combining of two or more logic gates to form a required function where the output at a given time depends only on the input.

- The required output data is obtained from this process by transforming the binary information given at the input.



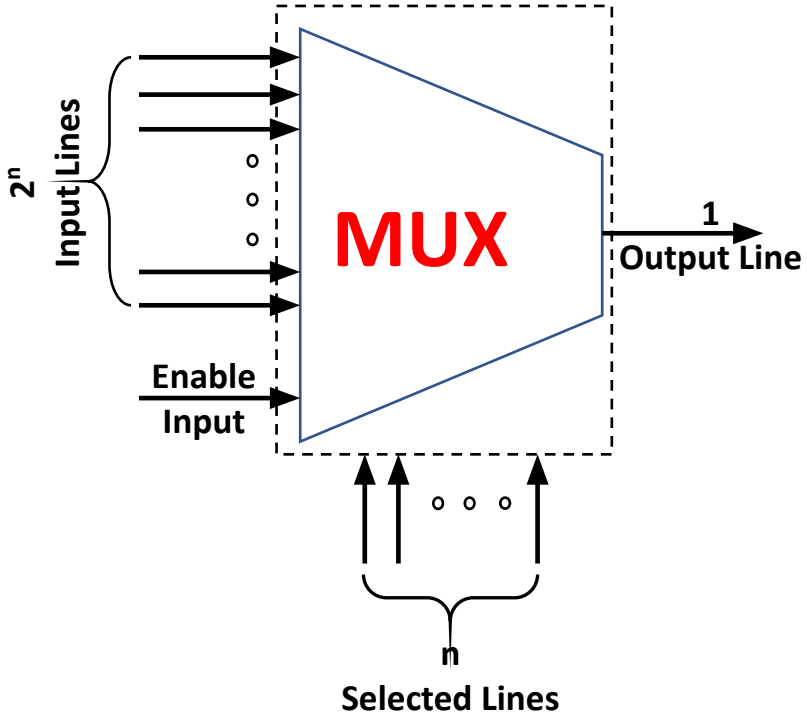
Classification of Combinational Logic



MULTIPLEXER

Multiplexer

- It is a combinational logic circuit.
- It is a device that selects between several analog or digital input signals and forwards it to a single output line.
- Input – 2^n
- Output – 1
- Multiplexer – $2^n : 1$
- The selection of particular input line is controlled by a set of selection lines.
- It is also called Data Selector.
- There is a Enable input, which is required to connect two or three multiplexer in parallel.



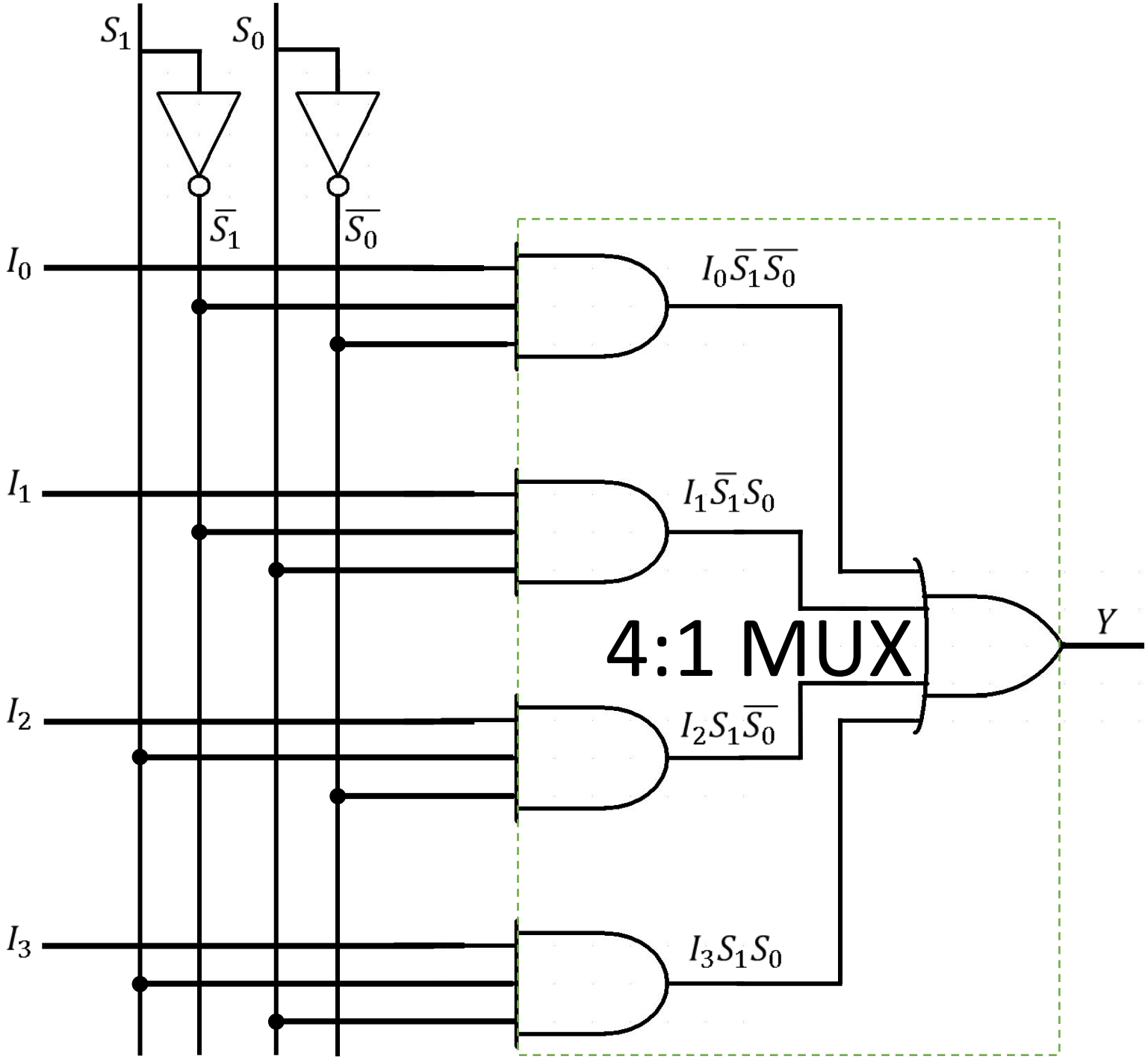
Design of 4:1 Mux:

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

- Input: 4 (I_0, I_1, I_2 & I_3)
- Output: 1 (Y)
- Selection Line: 2 (S_0 & S_1)
- Output is,

$$Y = I_0\overline{S_1}\,\overline{S_0} + I_1\overline{S_1}S_0 + I_2S_1\overline{S_0} + I_3S_1S_0$$

$$Y = I_0\overline{S_1}\,\overline{S_0} + I_1\overline{S_1}S_0 + I_2S_1\overline{S_0} + I_3S_1S_0$$

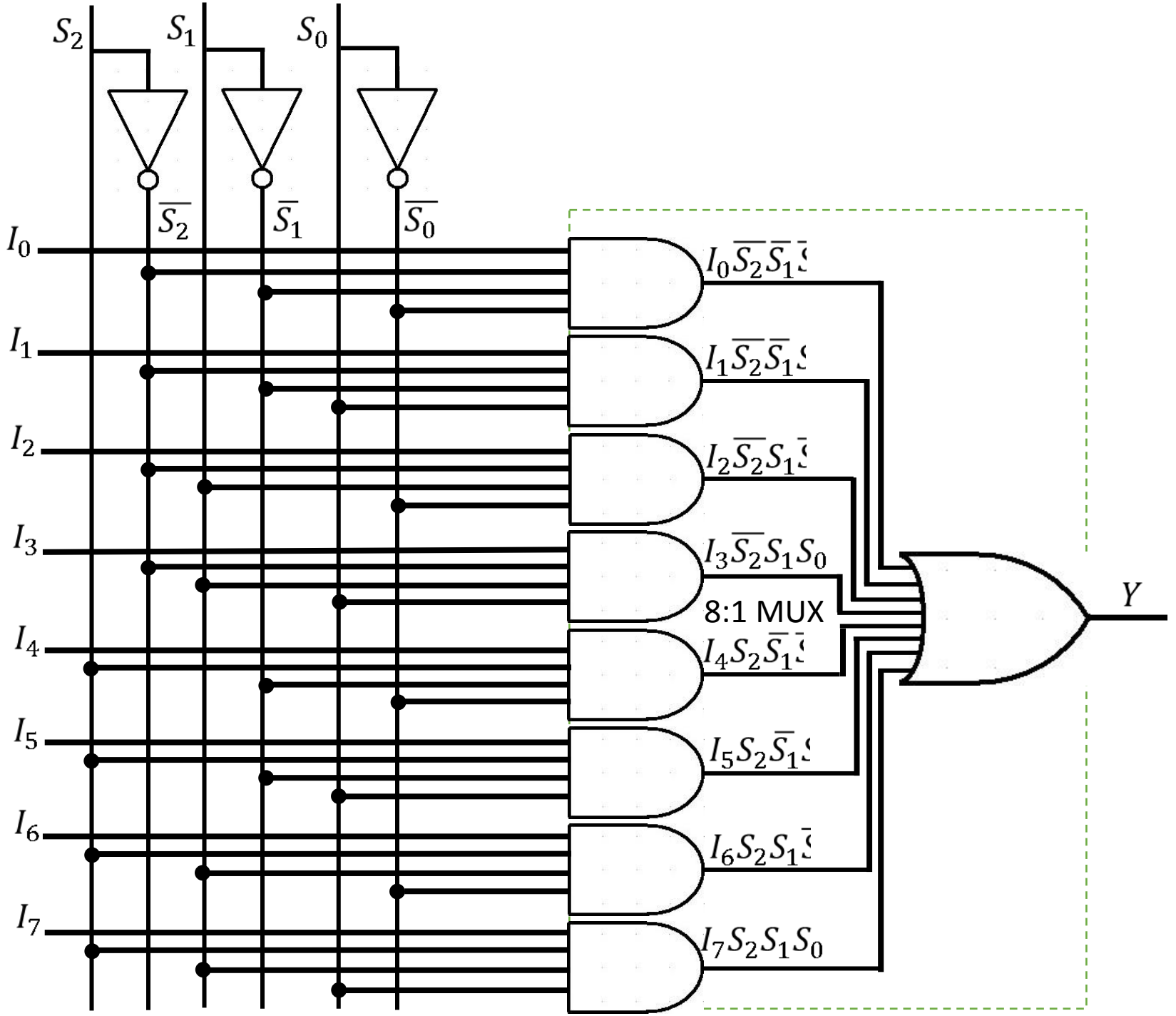


Design of 8:1 Mux:

S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

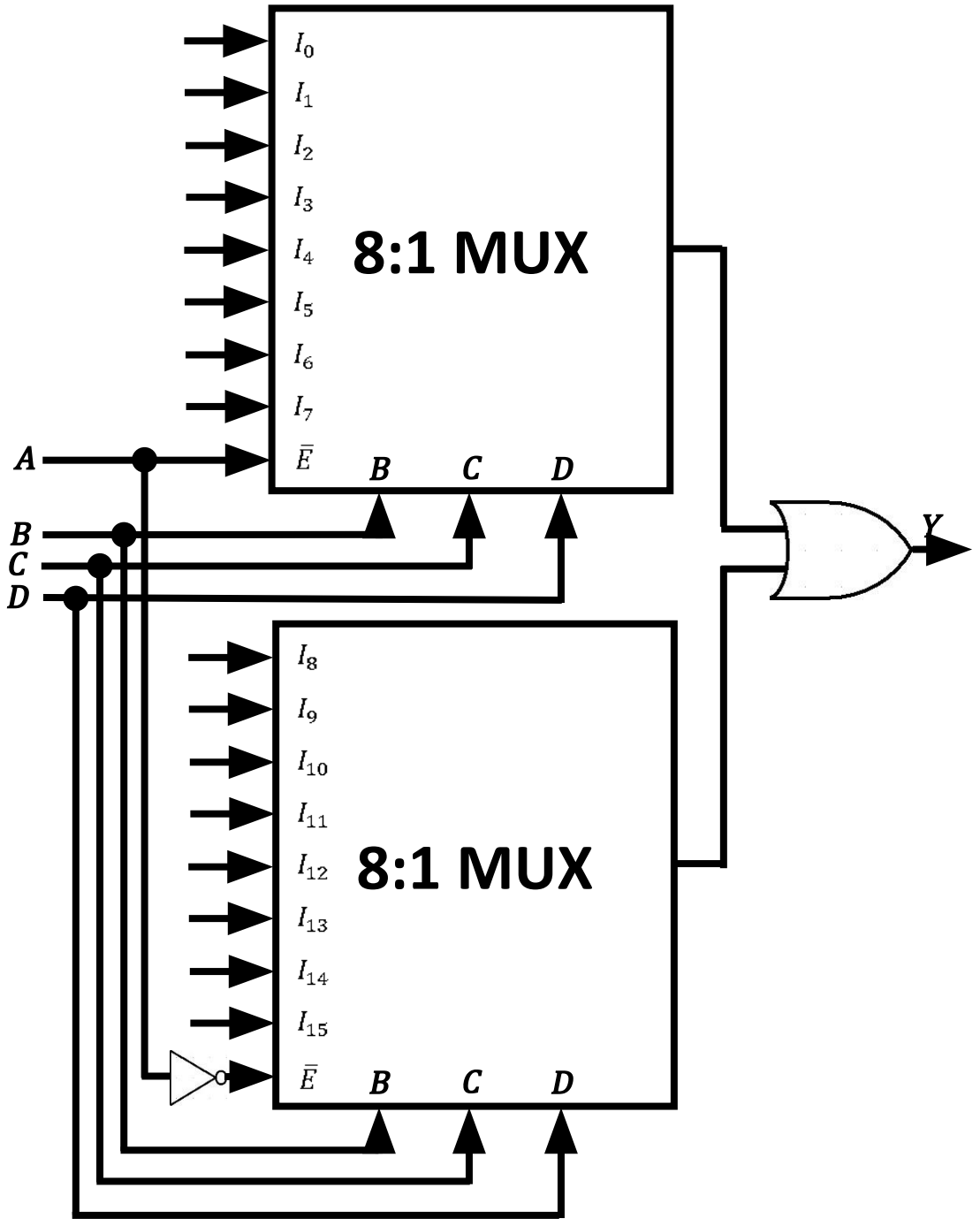
- Input: 8 (I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7 & I_8)
- Output: 1 (Y)
- Selection Line: 3 (S_0, S_1 & S_2)
- Output is,

$$Y = I_0\overline{S_2}\overline{S_1}\overline{S} + I_1\overline{S_2}\overline{S} + I_2\overline{S_2}\overline{S} + I_3\overline{S_2}S_1\overline{S} + I_4S_2\overline{S} + I_5S_2\overline{S} + I_6S_2\overline{S} + I_7S_2S_1\overline{S}$$



Design of 16:1 MUX using 8:1 MUX

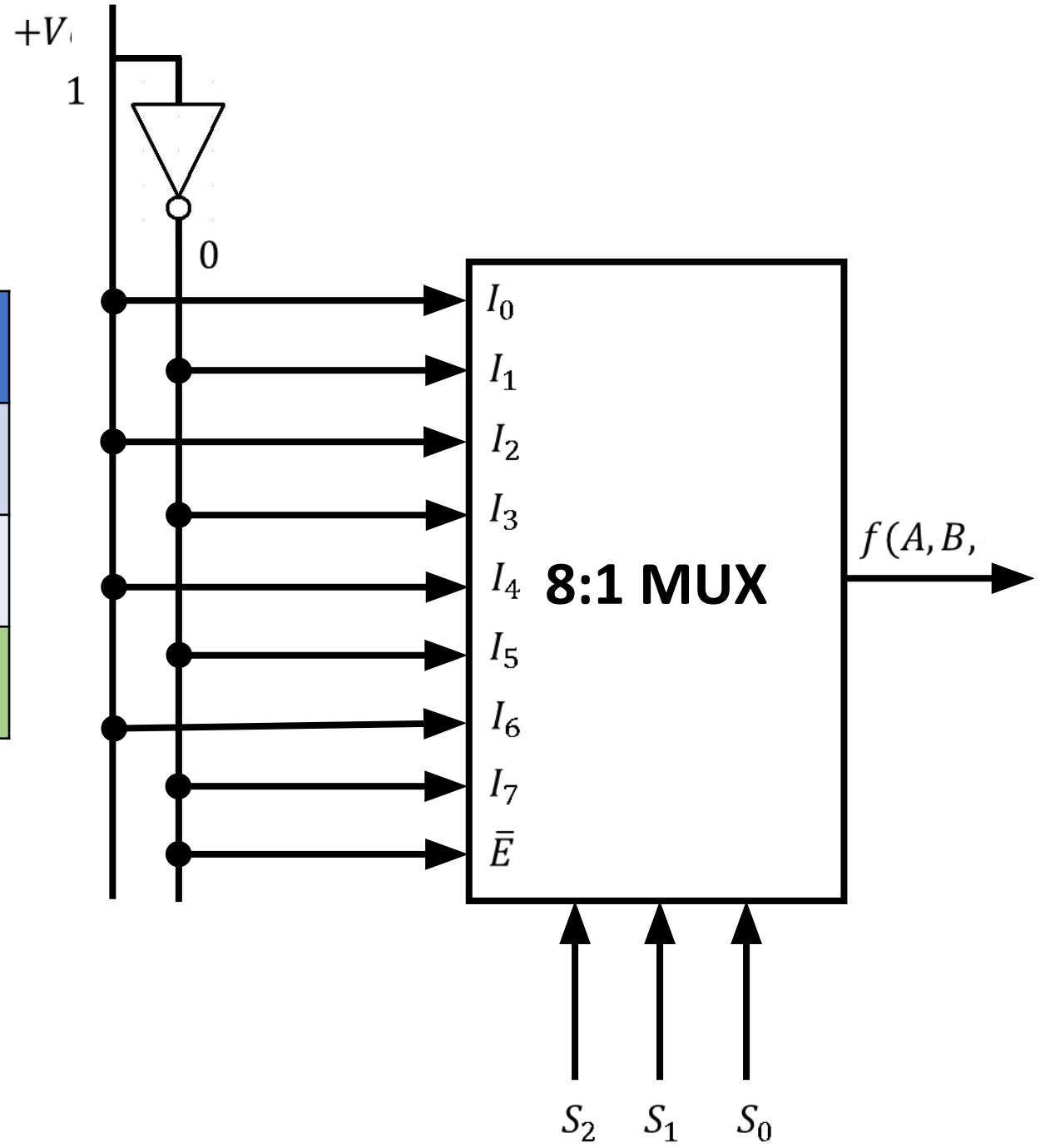
S_3	S_2	S_1	S_0	Y
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	



Implement the following function using 8:1 MUX

$$f(A,B,C,D) = \sum_m(0,2,4,6,8,10,12,14)$$

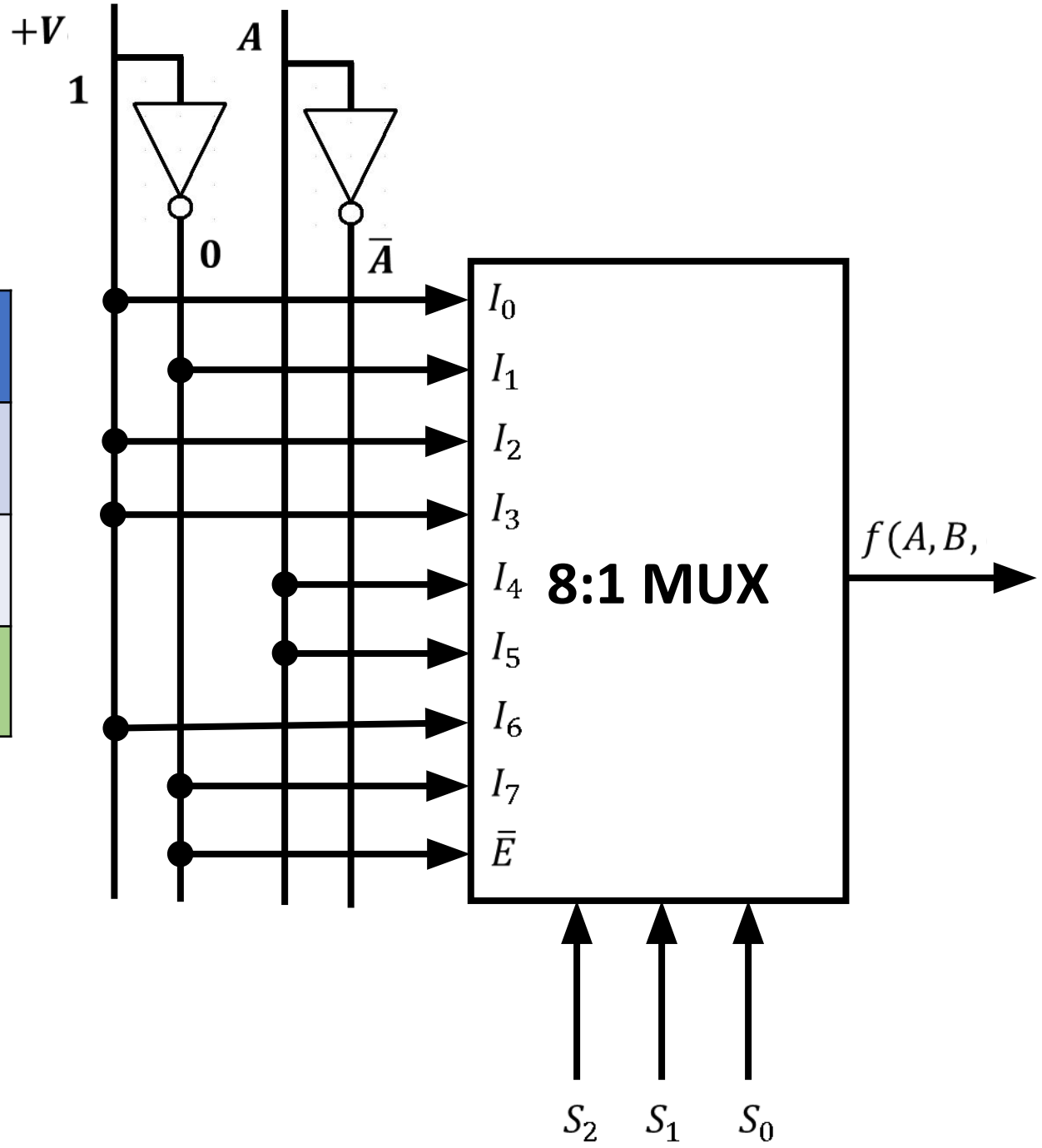
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
1	**0**	**1**	**0**	**1**	**0**	**1**	**0**



Implement the following function using 8:1 MUX

$$f(A,B,C,D) = \sum_m(0,2,6,10,11,12,13) + d(3,8,14)$$

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
1	0	1	1	A	A	1	0



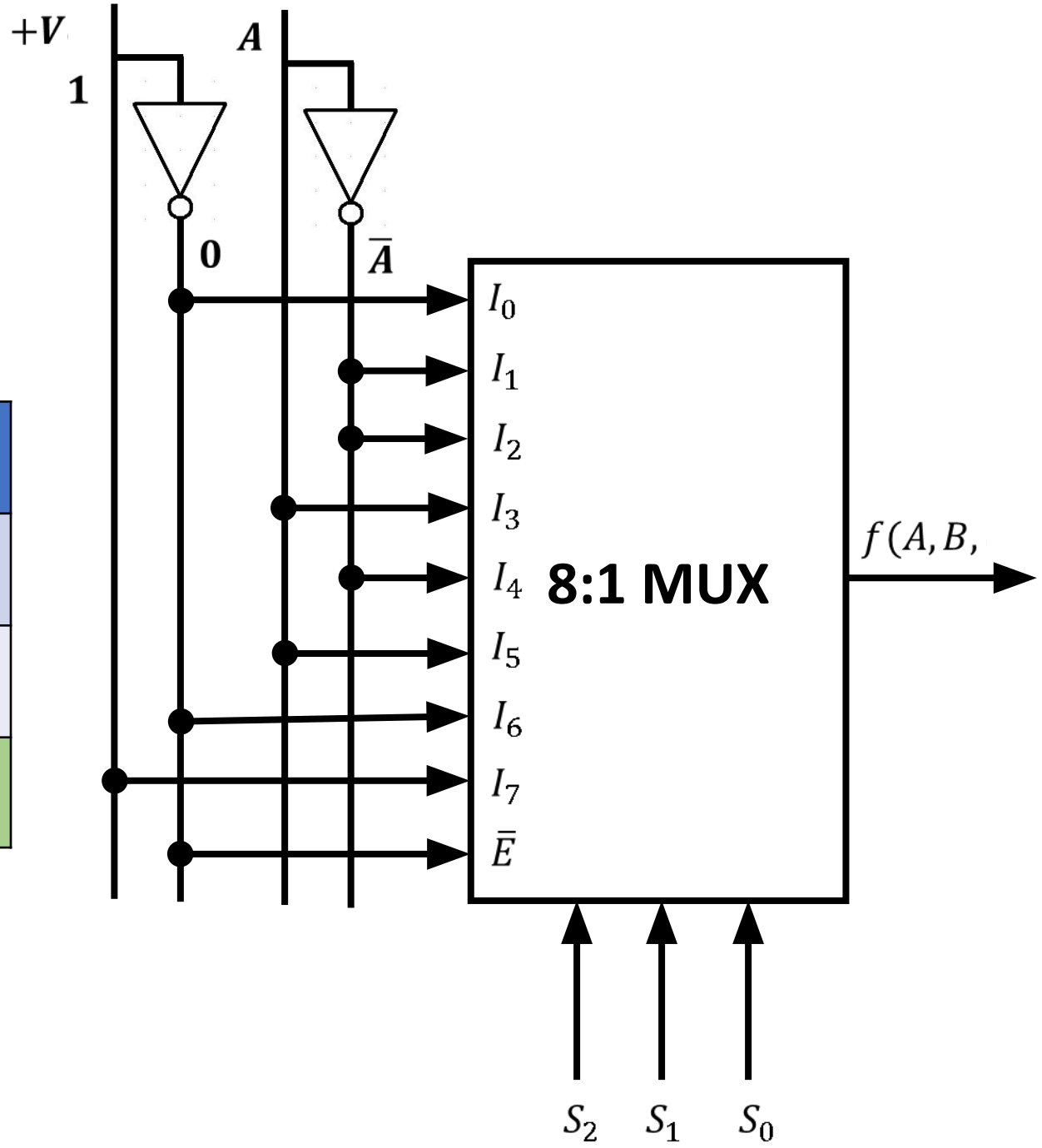
Implement the following function using 8:1 MUX

$$f(A,B,C,D) = \prod_M(0,3,5,6,8,9,10,12,14)$$

PoS function should be converted into SoP function,

$$f(A,B,C,D) = \sum_m(1,2,4,7,11,13,15)$$

	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
	0			A		A	0	1

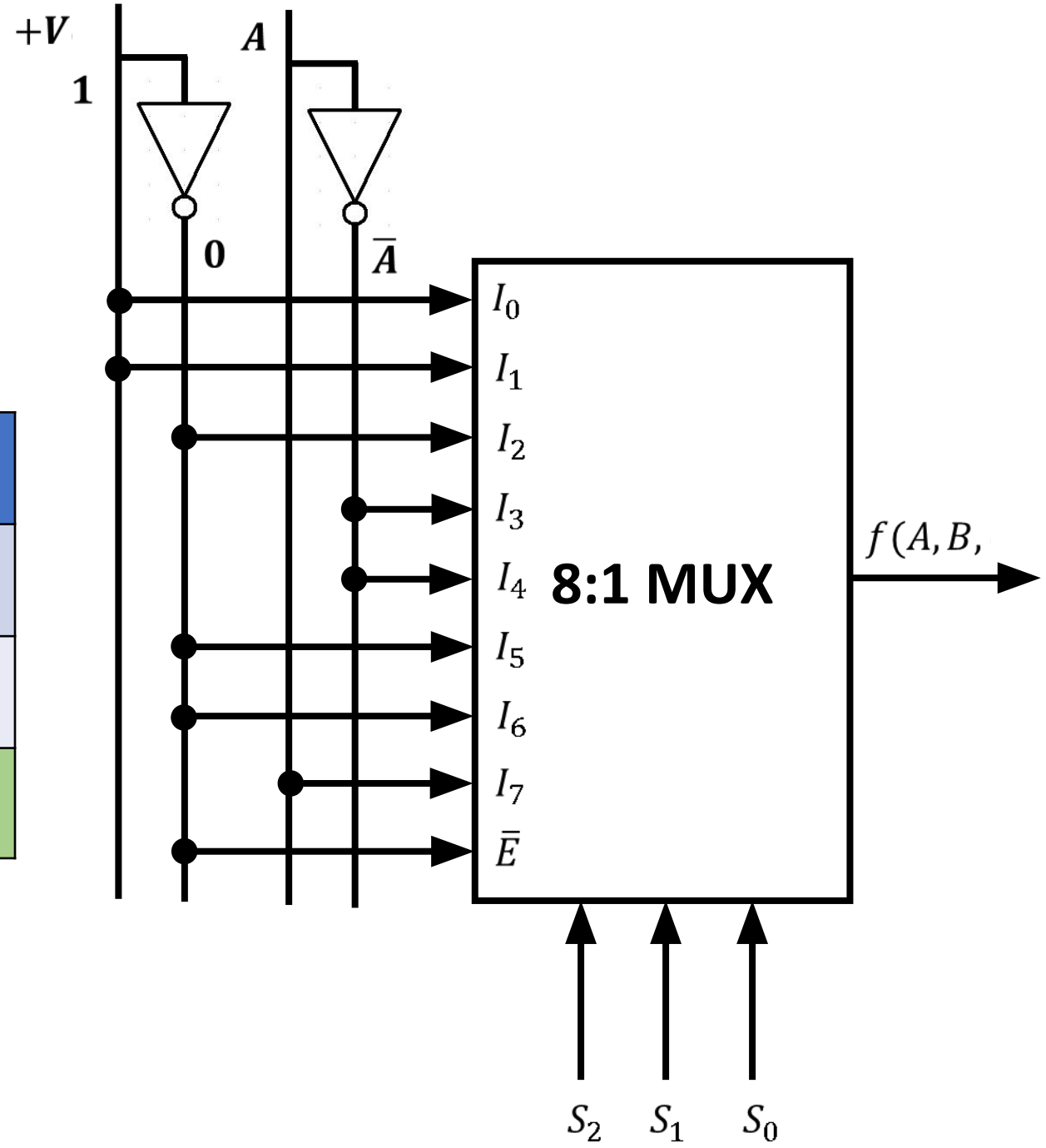


Implement the following function using 8:1 MUX and 4:1 MUX

$$f(A,B,C,D) = \sum_m(0,1,3,4,8,9,15)$$

Using 8:1 Mux

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
1	1	0			0	0	A

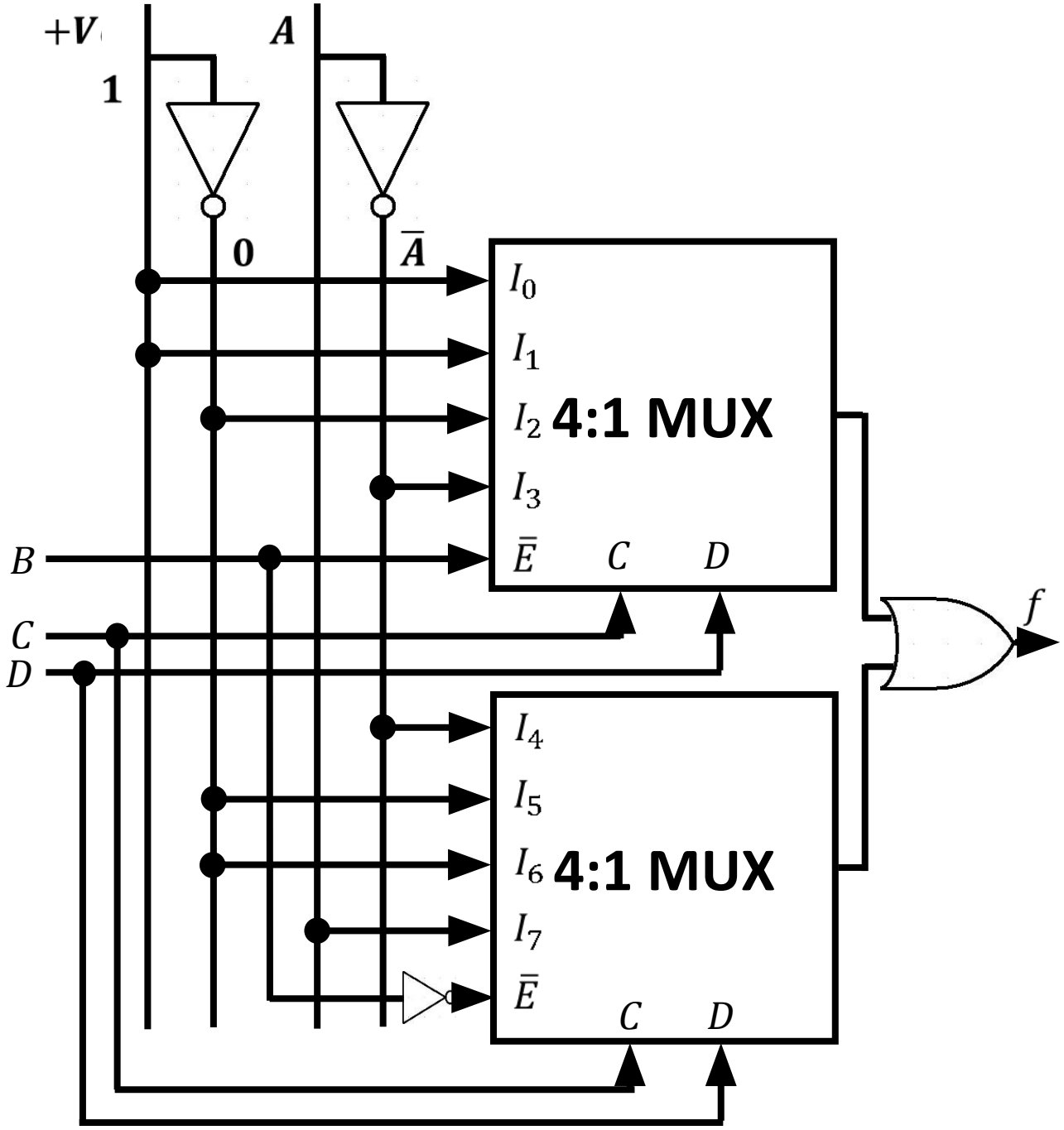


Implement the following function using 8:1 MUX and 4:1 MUX

$$f(A,B,C,D) = \sum_m(0,1,3,4,8,9,15)$$

Using 4:1 Mux

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
1	1	0			0	0	A



Application of Multiplexer

- It is used as a data selector to select one out of many data inputs.
- It is used to implement combinational logic circuit.
- It is used in time multiplexing systems.
- It is used in ADC and DAC converter.
- It is used in data acquisition systems.

THANK YOU…

ANY QUERIEES???...

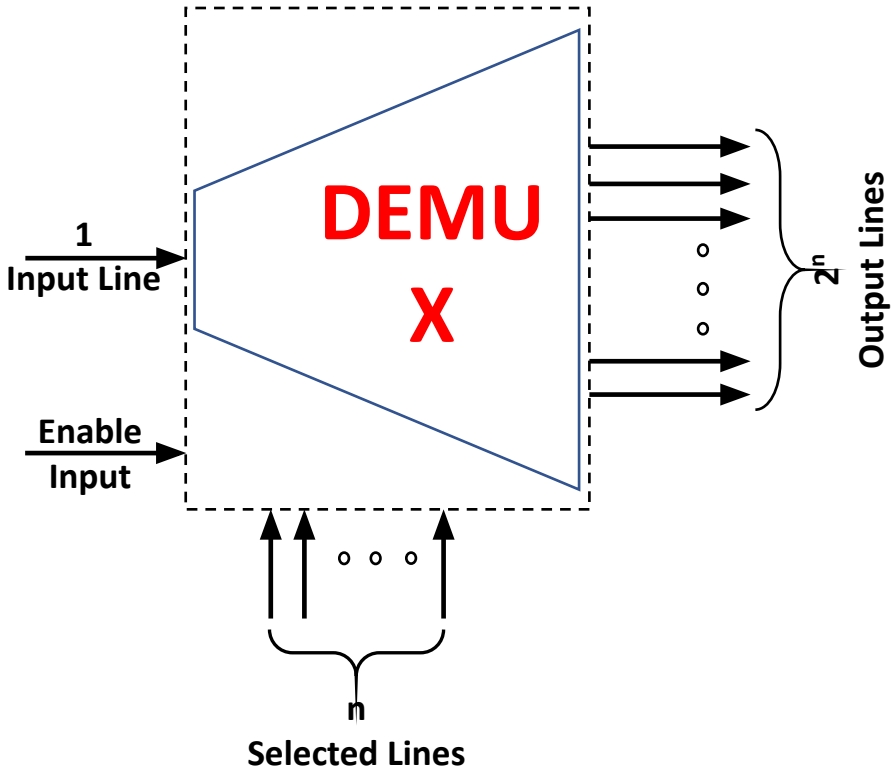
DEMULTIPLEXER

PREPARED BY
Mr.P.KANAKARAJ
AP/EEE
SRMIST-KTR

Demultiplexer

:

- It is a combinational logic circuit.
- It is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines.
- The selection of specific output line is controlled by the values of n selection lines.
- Input – 1
- Output – 2^n
- Selection lines – n
- De-Multiplexer – 1 : 2^n
- There is a Enable input, which is required to connect two or three multiplexer in parallel.



Design of 1:4 DEMUX:

Truth Table

Selection Lines		Outputs			
0	0	I	0	0	0
0	1	0	I	0	0
1	0	0	0	I	0
1	1	0	0	0	I

- Input: 1 (I)
- Output: 4 (Y_0, Y_1, Y_2 & Y_3)
- Selection Line: 2 (S_0 & S_1)
- Outputs are,

$$Y_0 = I\overline{S_1}\,\overline{S_0} = I$$
$$Y_1 = I\overline{S_1}S_0 = I$$
$$Y_2 = IS_1\overline{S_0} = I$$
$$Y_3 = IS_1S_0 = I$$

Logic Diagram of 1:4 DEMUX:

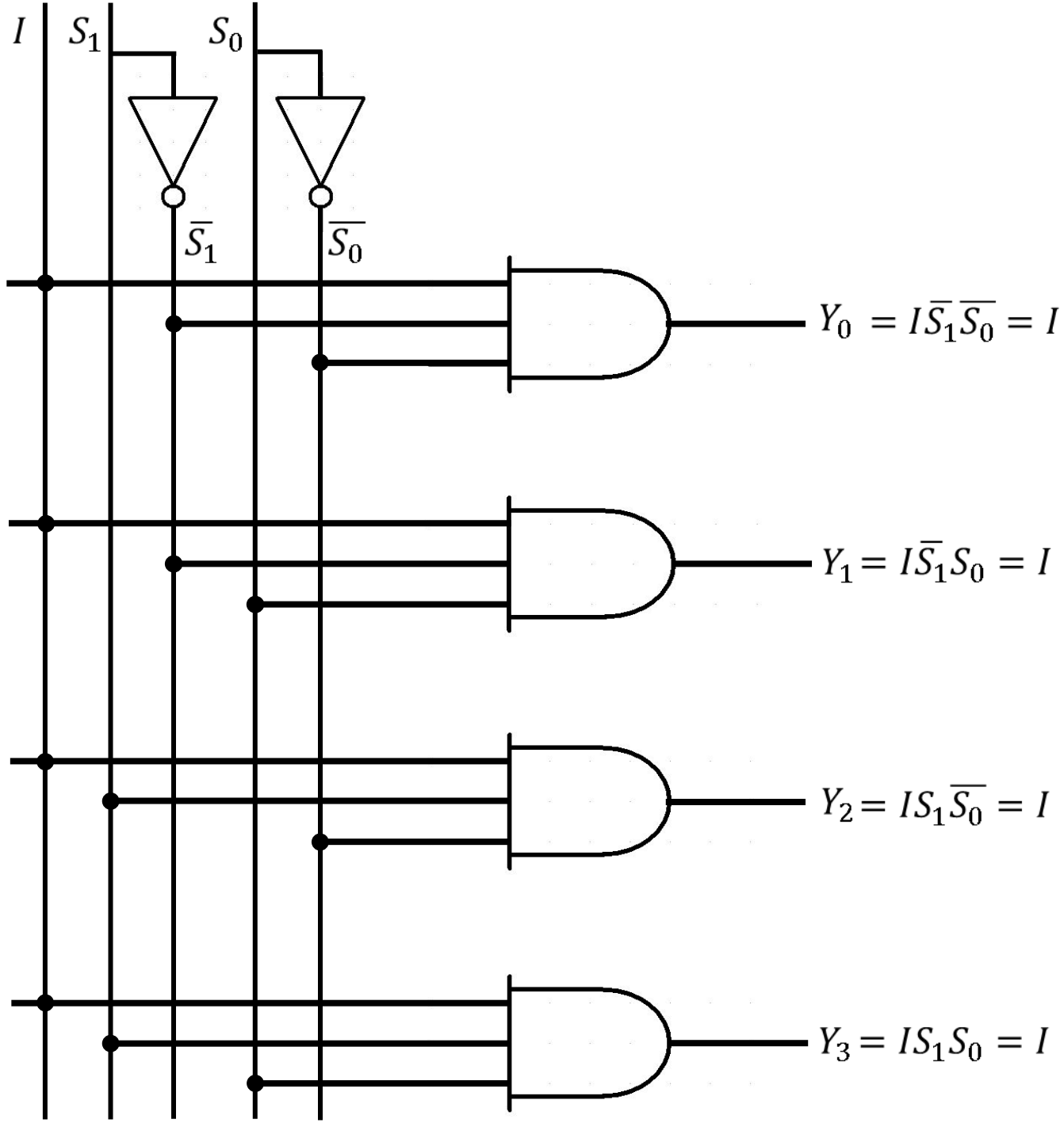
Outputs are,

$$Y_0 = I\overline{S_1}\,\overline{S_0} = I$$

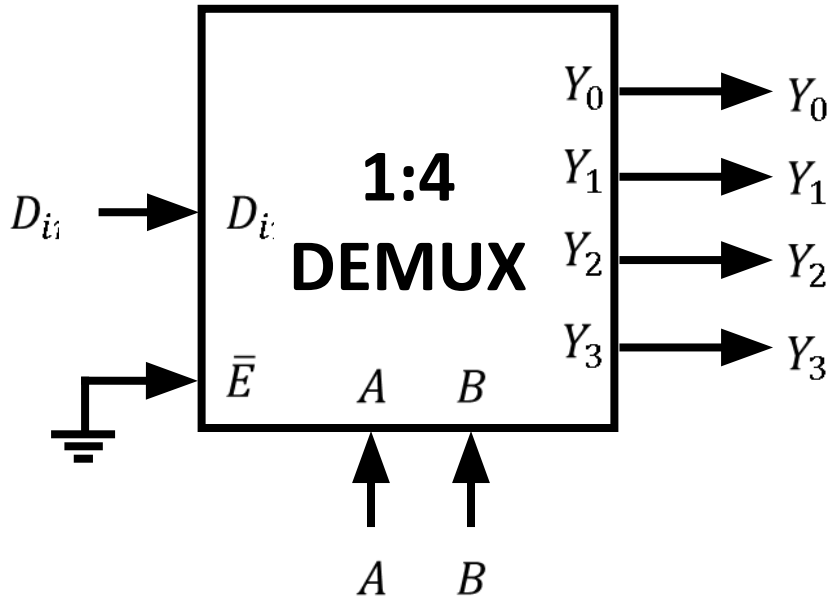
$$Y_1 = I\overline{S_1}S_0 = I$$

$$Y_2 = IS_1\overline{S_0} = I$$

$$Y_3 = IS_1S_0 = I$$



1:4 DEMUX



Design of 1:8 DEMUX

Truth Table

Selection Lines			*Outputs*							
0	*0*	*0*	*I*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0	*0*	*1*	*0*	*I*	*0*	*0*	*0*	*0*	*0*	*0*
0	*1*	*0*	*0*	*0*	*I*	*0*	*0*	*0*	*0*	*0*
0	*1*	*1*	*0*	*0*	*0*	*I*	*0*	*0*	*0*	*0*
1	*0*	*0*	*0*	*0*	*0*	*0*	*I*	*0*	*0*	*0*
1	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*I*	*0*	*0*
1	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*I*	*0*
1	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*I*

- Input: 1 (I)
- Output: 8 (Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7)
- Selection Line: 3 (S_0, S_1 & S_2)
- Outputs are,

$$Y_0 = I\overline{S_2}\,\overline{S_1}\,\overline{S_0} \qquad Y_4 = IS_2\overline{S_1}\,\overline{S_0}$$

$$Y_1 = I\overline{S_2}\,\overline{S_1}S_0 \qquad Y_5 = IS_2\overline{S_1}S_0$$

$$Y_2 = I\overline{S_2}S_1\overline{S_0} \qquad Y_6 = IS_2S_1\overline{S_0}$$

$$Y_3 = I\overline{S_2}S_1S_0 \qquad Y_7 = IS_2S_1S_0$$

Logic Diagram of 1:8 DEMUX:

Outputs are,

$Y_0 = I\overline{S_2}\,\overline{S_1}\,\overline{S_0}$

$Y_1 = I\overline{S_2}\,\overline{S_1}S_0$

$Y_2 = I\overline{S_2}S_1\overline{S_0}$

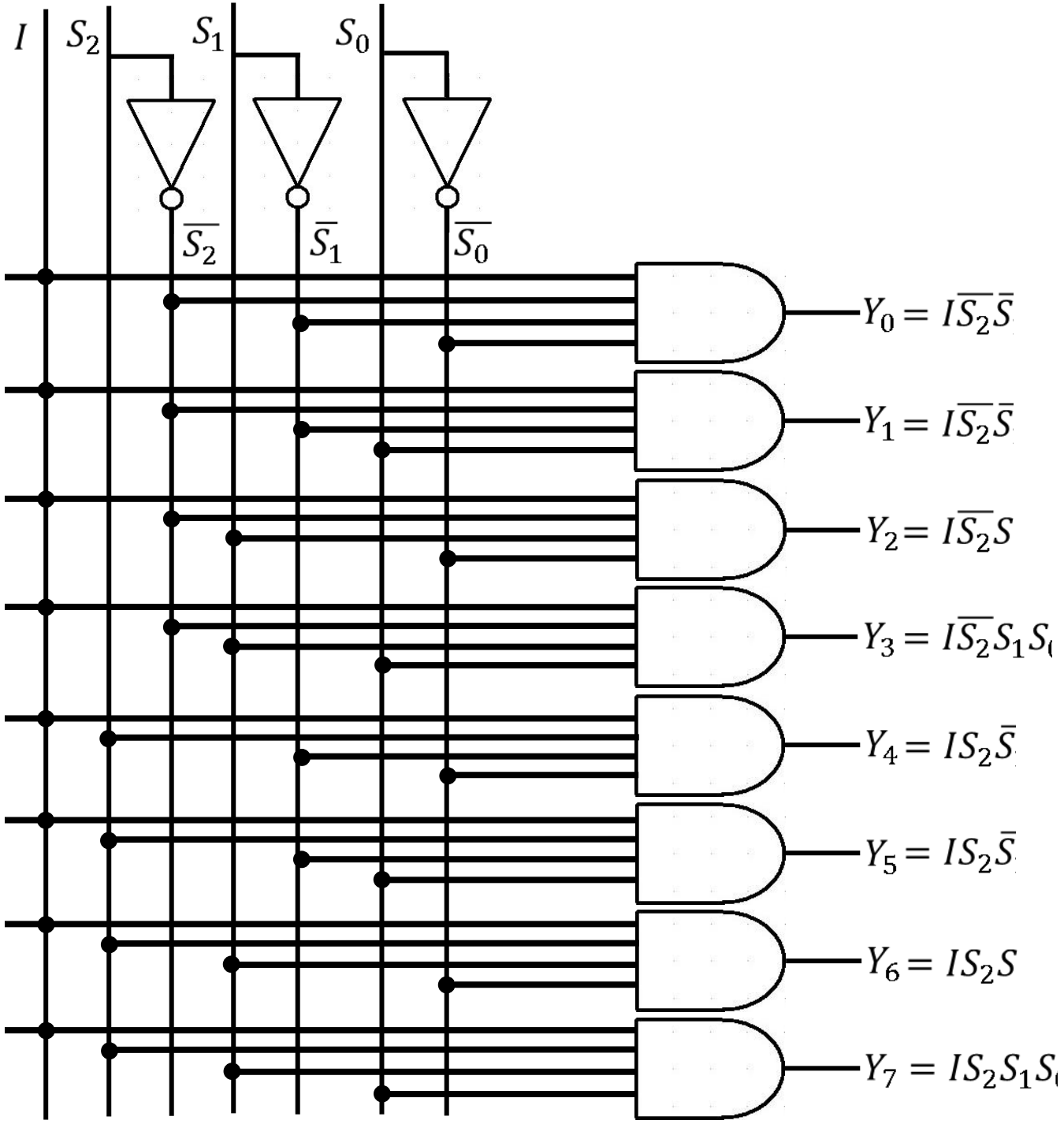
$Y_3 = I\overline{S_2}S_1S_0$

$Y_4 = IS_2\overline{S_1}\,\overline{S_0}$

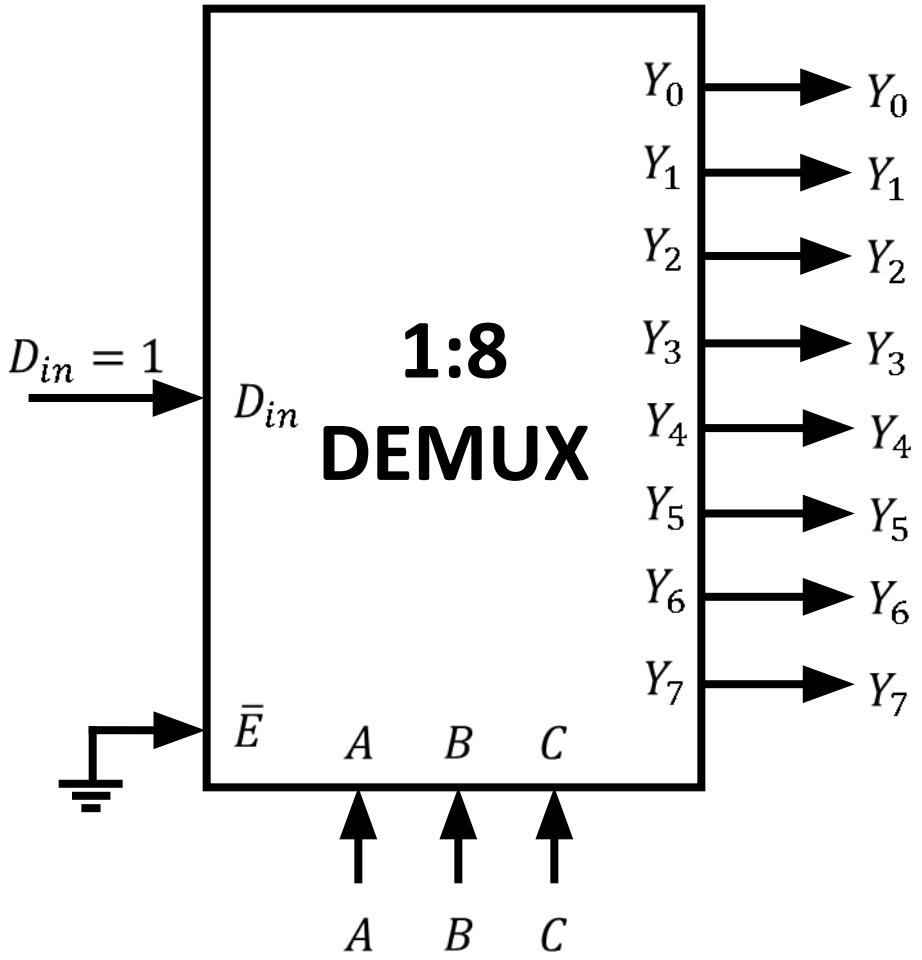
$Y_5 = IS_2\overline{S_1}S_0$

$Y_6 = IS_2S_1\overline{S_0}$

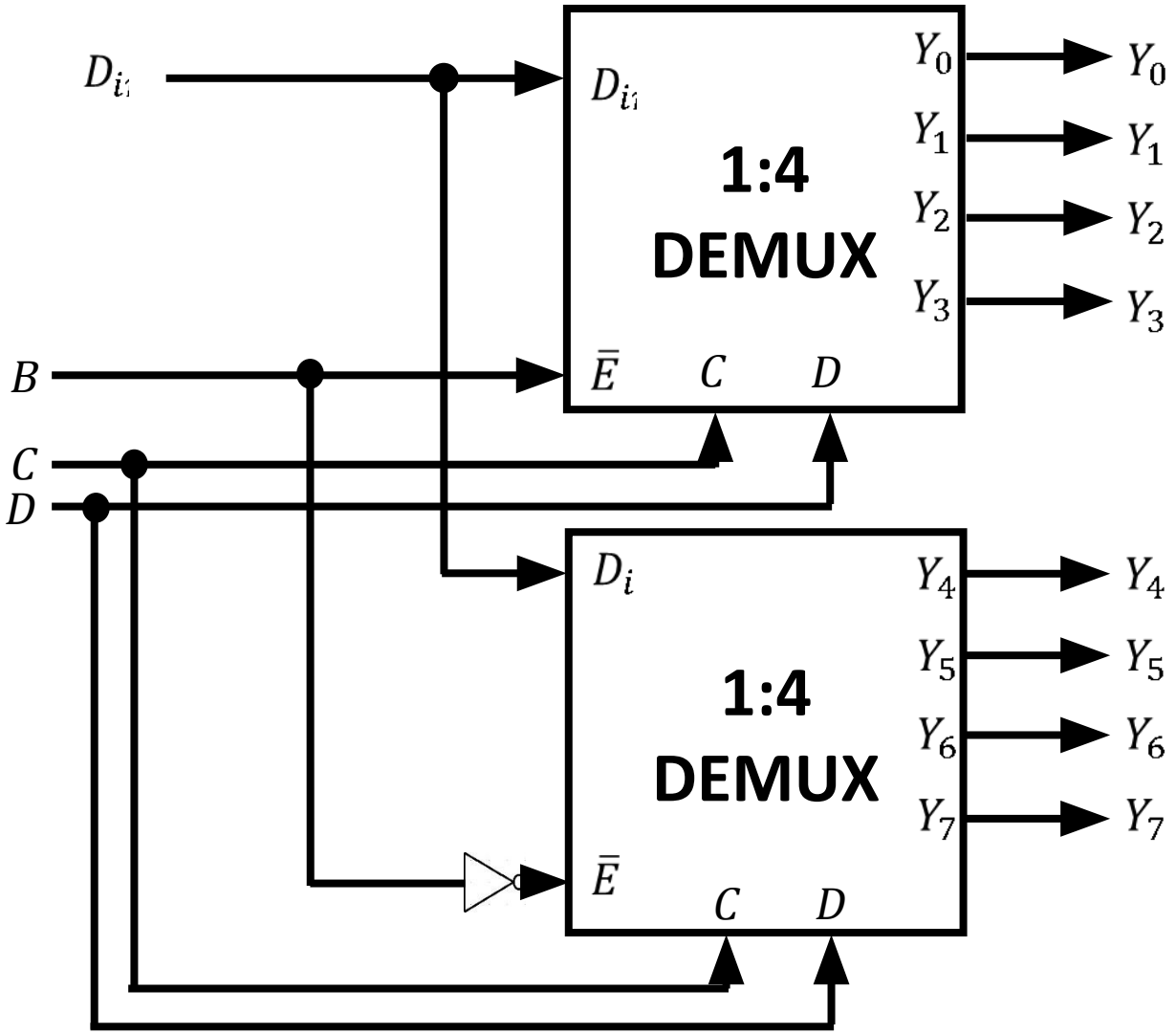
$Y_7 = IS_2S_1S_0$



1:8 DEMUX


$D_{in} = 1$
D_{in}
1:8
DEMUX
$\bar{E}$
A
B
C
Y_0
Y_1
Y_2
Y_3
Y_4
Y_5
Y_6
Y_7
Y_0
Y_1
Y_2
Y_3
Y_4
Y_5
Y_6
Y_7
A
B
C

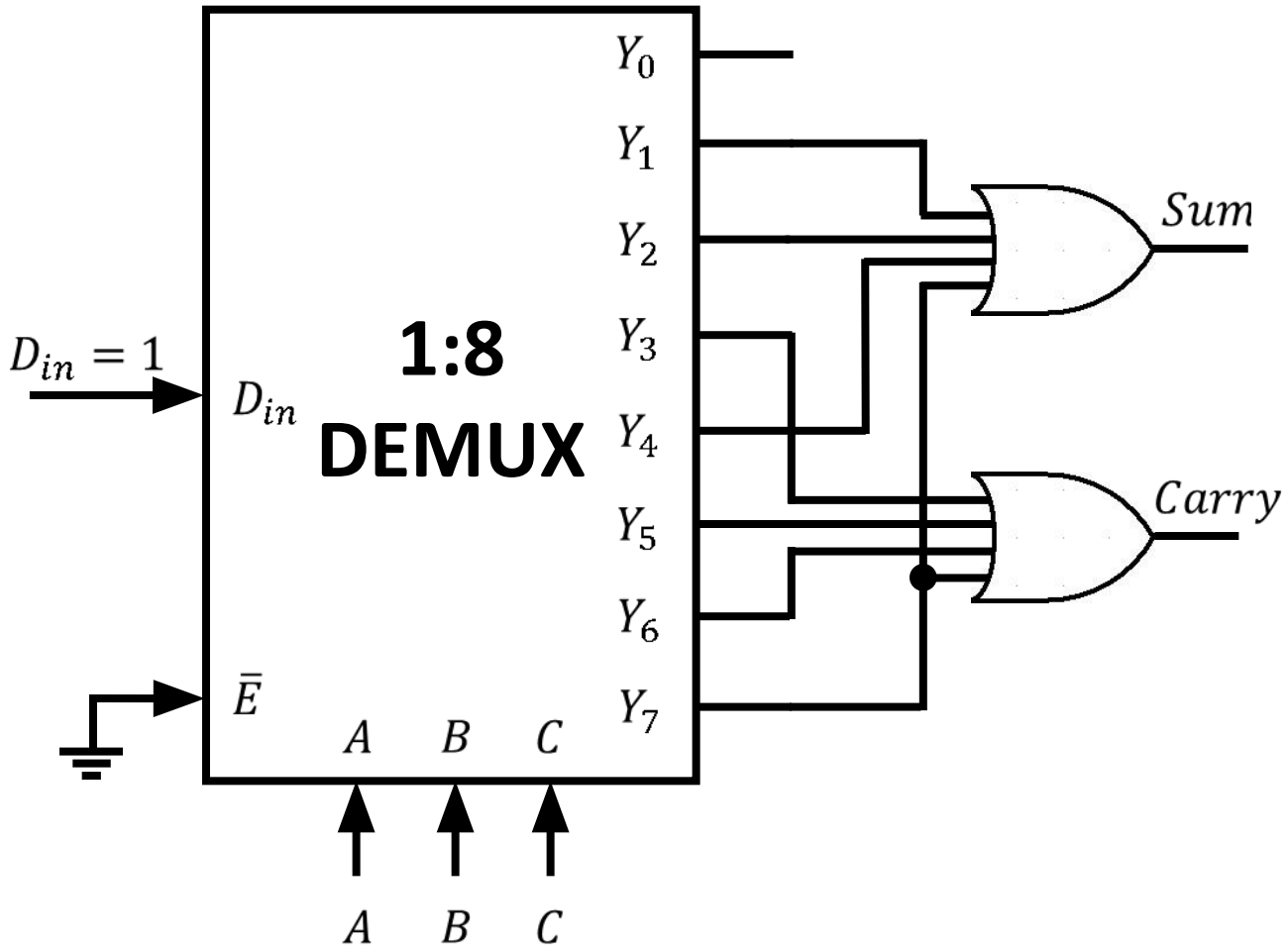
Design of 1:8 DEMUX Using 1:4 DEMUX



Implementation of Full Adder using Demultiplexer

Truth Table

Decimal	Inputs			Outputs	
	A	B	C	Sum	Carry
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1



Output Expressions are,

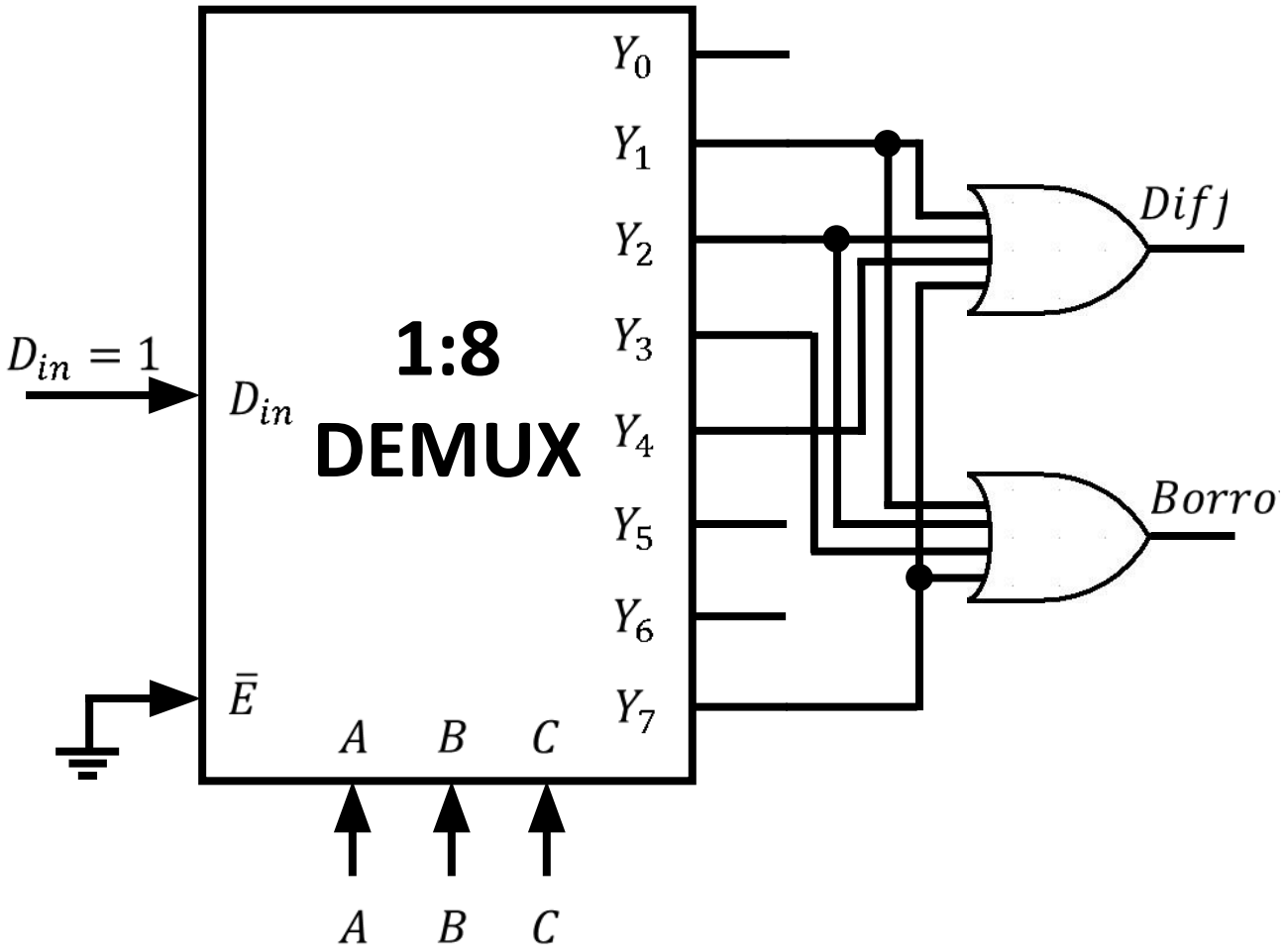
$$\text{Sum} = \sum_m(1,2,4,7)$$

$$\text{Carry} = \sum_m(3,5,6,7)$$

Implementation of Full Subtractor using Demultiplexer

Truth Table

Decimal	Inputs			Outputs	
	A	B	C	Difference	Borrow
0	0	0	0	0	0
1	0	0	1	1	1
2	0	1	0	1	1
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	0
6	1	1	0	0	0
7	1	1	1	1	1



Output Expressions are,

$$\text{Difference} = \sum_m(1,2,4,7)$$

$$\text{Borrow} = \sum_m(1,2,3,7)$$

Applications of Demultiplexer:

- Clock demultiplexer
- Security monitoring system
- Synchronous data transmission system

THANK YOU…

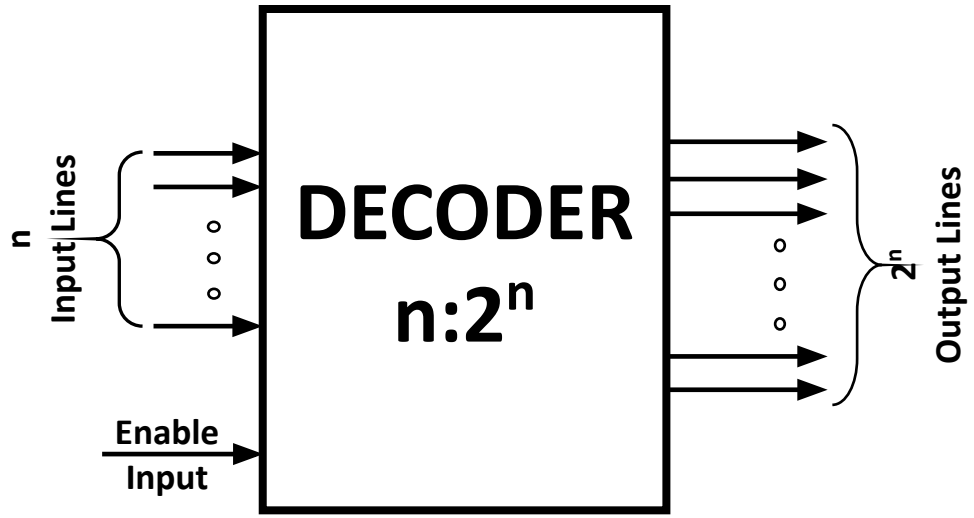
ANY QUERIEES???...

DECODER

PREPARED BY
Mr.P.KANAKARAJ
AP/EEE
SRMIST-KTR

Decoder:

- It is a combinational logic circuit.
- It has an n-bit binary input code and a one activated output out of 2^n output code is called binary decoder.
- A binary decoder is used when it is necessary to activate exactly one of 2^n outputs based on an n-bit input value.
- It is similar to demultiplexer, with only one exception that it has no data input.
- Input – n
- Output – 2^n
- Decoder – n : 2^n
- There is a Enable input, which is required to connect two or three decoder in parallel.



Decoder Vs Demultiplexer

- **Decoder**
  - Decoder is a many inputs to many outputs device.
  - There are no selection lines.

- **Demultiplexer**
  - Demultiplexer is a one input to many outputs device.
  - The selection of specific output line is controlled by the value of selection lines.

Design of 2:4 DECODER

Truth Table

Input			Output			
1	X	X	0	0	0	0
0	0	0	*1*	0	0	0
0	0	1	0	*1*	0	0
0	1	0	0	0	*1*	0
0	1	1	0	0	0	*1*

- Input: 1 (I)
- Output: 4 (Y_0, Y_1, Y_2 & Y_3)
- Selection Line: 2 (S_0 & S_1)
- Outputs are,

$$Y_0 = \bar{E}\overline{S_1}\,\overline{S_0}$$

$$Y_1 = \bar{E}\overline{S_1}S_0$$

$$Y_2 = \bar{E}S_1\overline{S_0}$$

$$Y_3 = \bar{E}S_1S_0$$

Logic Diagram of 2:4 Decoder

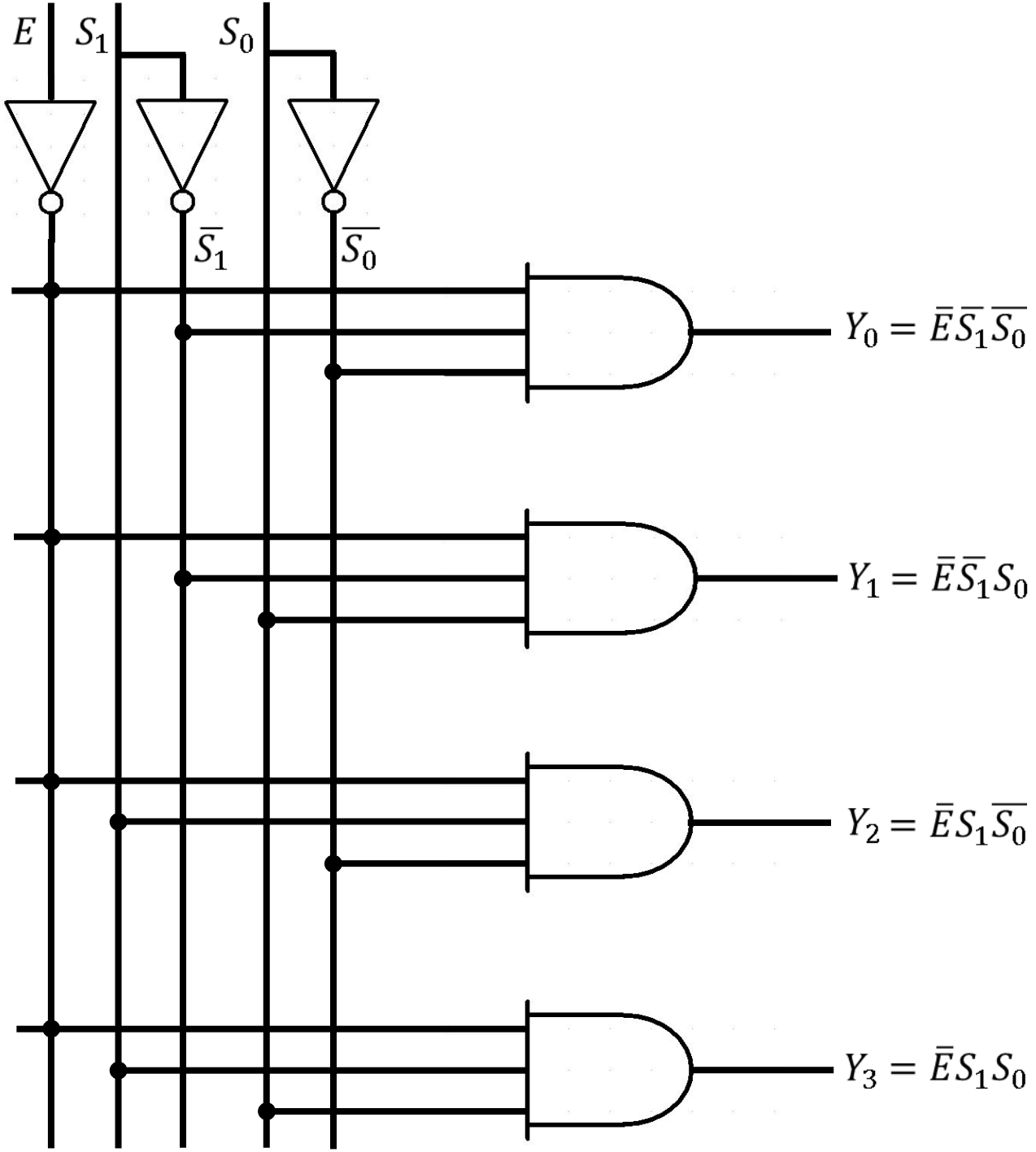
Outputs are,

$$Y_0 = \bar{E}\bar{S_1}\bar{S_0}$$

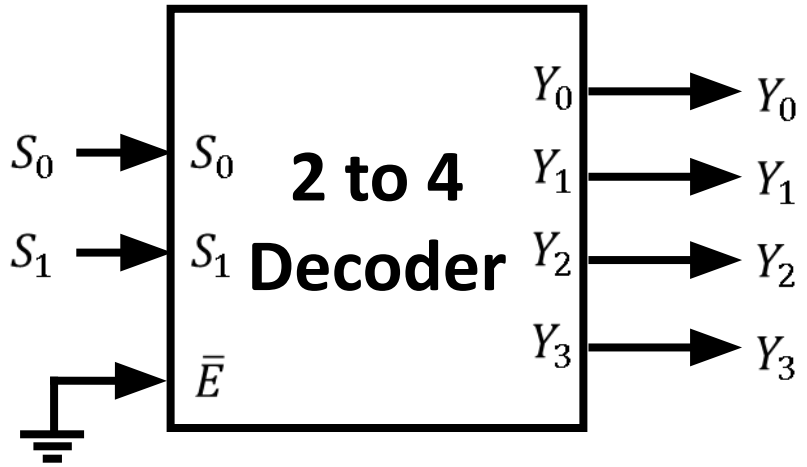
$$Y_1 = \bar{E}\bar{S_1}S_0$$

$$Y_2 = \bar{E}S_1\bar{S_0}$$

$$Y_3 = \bar{E}S_1S_0$$



2 to 4 Line Decoder



Three to Eight line Decoder

Truth Table

Input			Output							
0	0	0	I	0	0	0	0	0	0	0
0	0	1	0	I	0	0	0	0	0	0
0	1	0	0	0	I	0	0	0	0	0
0	1	1	0	0	0	I	0	0	0	0
1	0	0	0	0	0	0	I	0	0	0
1	0	1	0	0	0	0	0	I	0	0
1	1	0	0	0	0	0	0	0	I	0
1	1	1	0	0	0	0	0	0	0	I

- Input: 3 (S_0, S_1 , S_2)
- Output: 8 (Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7)
- Outputs are,

$$Y_0 = \overline{S_2}\,\overline{S_1}\,\overline{S_0} \qquad Y_4 = S_2\overline{S_1}\,\overline{S_0}$$

$$Y_1 = \overline{S_2}\,\overline{S_1}S_0 \qquad Y_5 = S_2\overline{S_1}S_0$$

$$Y_2 = \overline{S_2}S_1\overline{S_0} \qquad Y_6 = S_2S_1\overline{S_0}$$

$$Y_3 = \overline{S_2}S_1S_0 \qquad Y_7 = S_2S_1S_0$$

Logic Diagram of 3 to 8 line Decoder

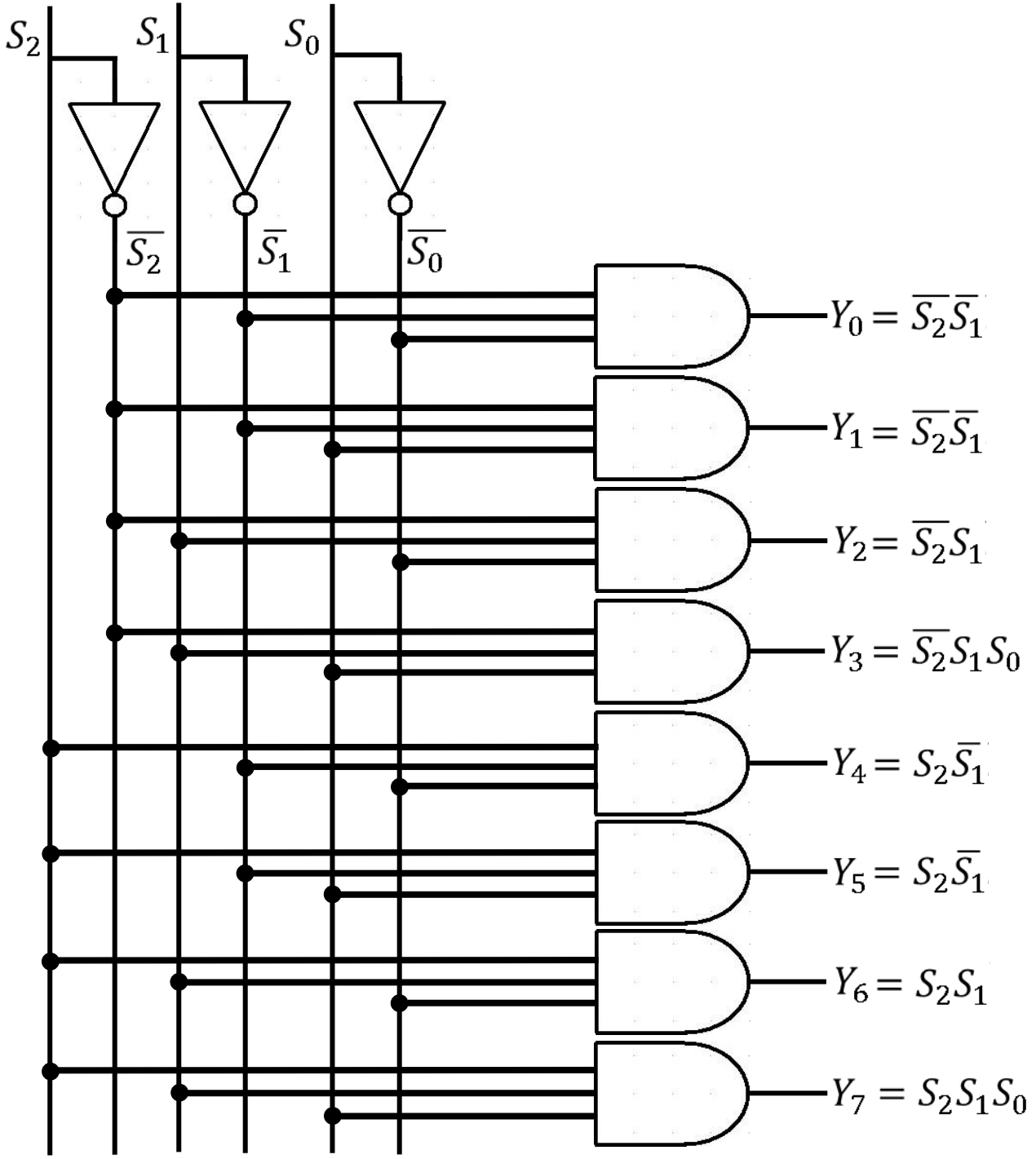
Outputs are,

$$Y_0 = \overline{S_2}\,\overline{S_1}\,\overline{S_0} \qquad Y_4 = S_2\overline{S_1}\,\overline{S_0}$$

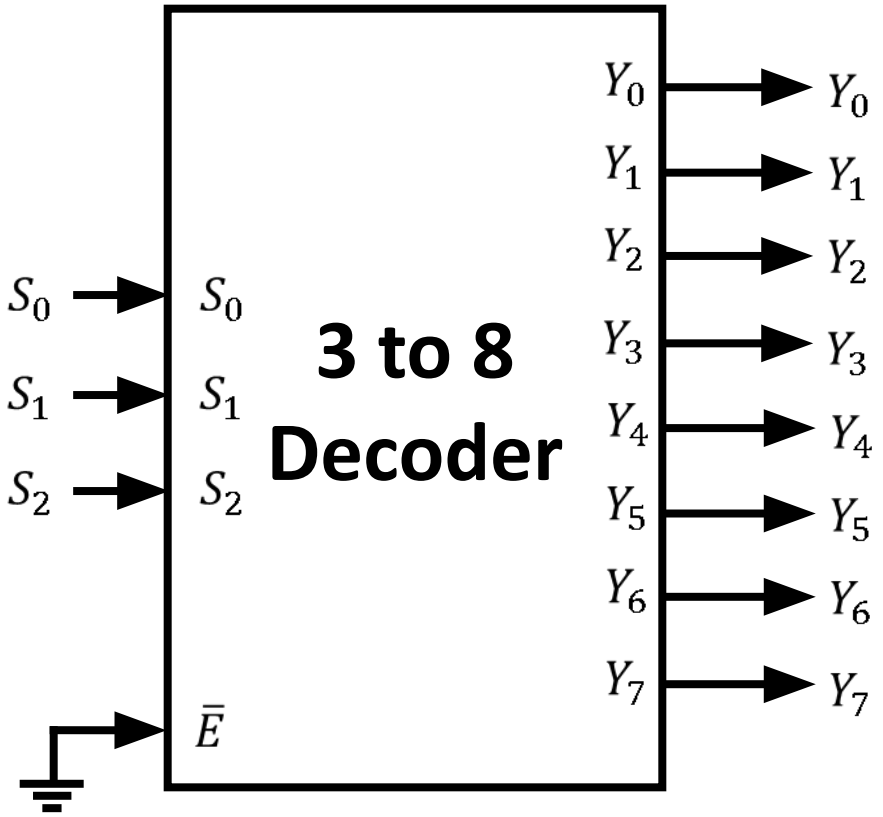
$$Y_1 = \overline{S_2}\,\overline{S_1}S_0 \qquad Y_5 = S_2\overline{S_1}S_0$$

$$Y_2 = \overline{S_2}S_1\overline{S_0} \qquad Y_6 = S_2S_1\overline{S_0}$$

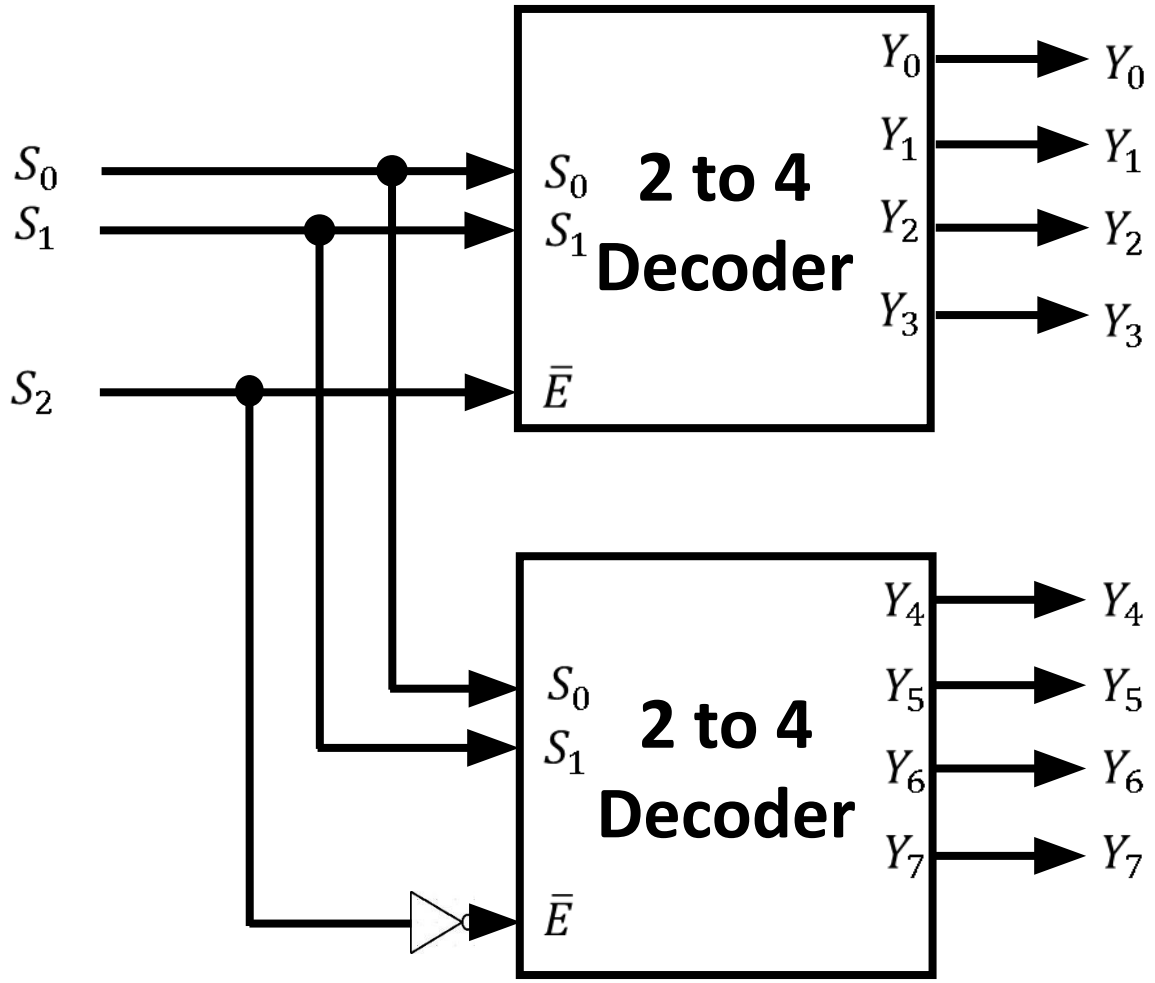
$$Y_3 = \overline{S_2}S_1S_0 \qquad Y_7 = S_2S_1S_0$$



3 to 8 Line Decoder



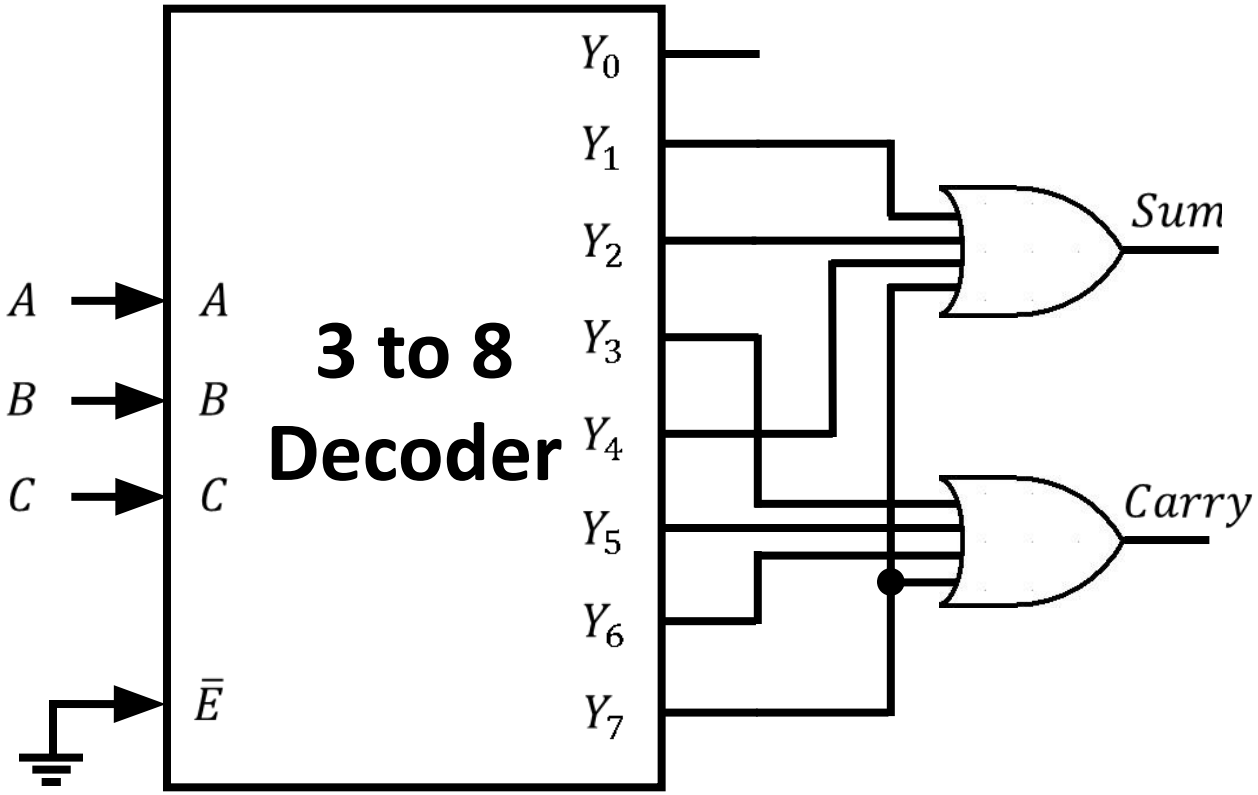
Design of 3 to 8 line Decoder Using 2 to 4 line Decoder



Implementation of Full Adder using Decoder

Truth Table

Decimal	Inputs			Outputs	
	A	B	C	Sum	Carry
0	0	0	0	0	0
1	0	0	1	1	0
2	0	1	0	1	0
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	1
6	1	1	0	0	1
7	1	1	1	1	1



Output Expressions are,

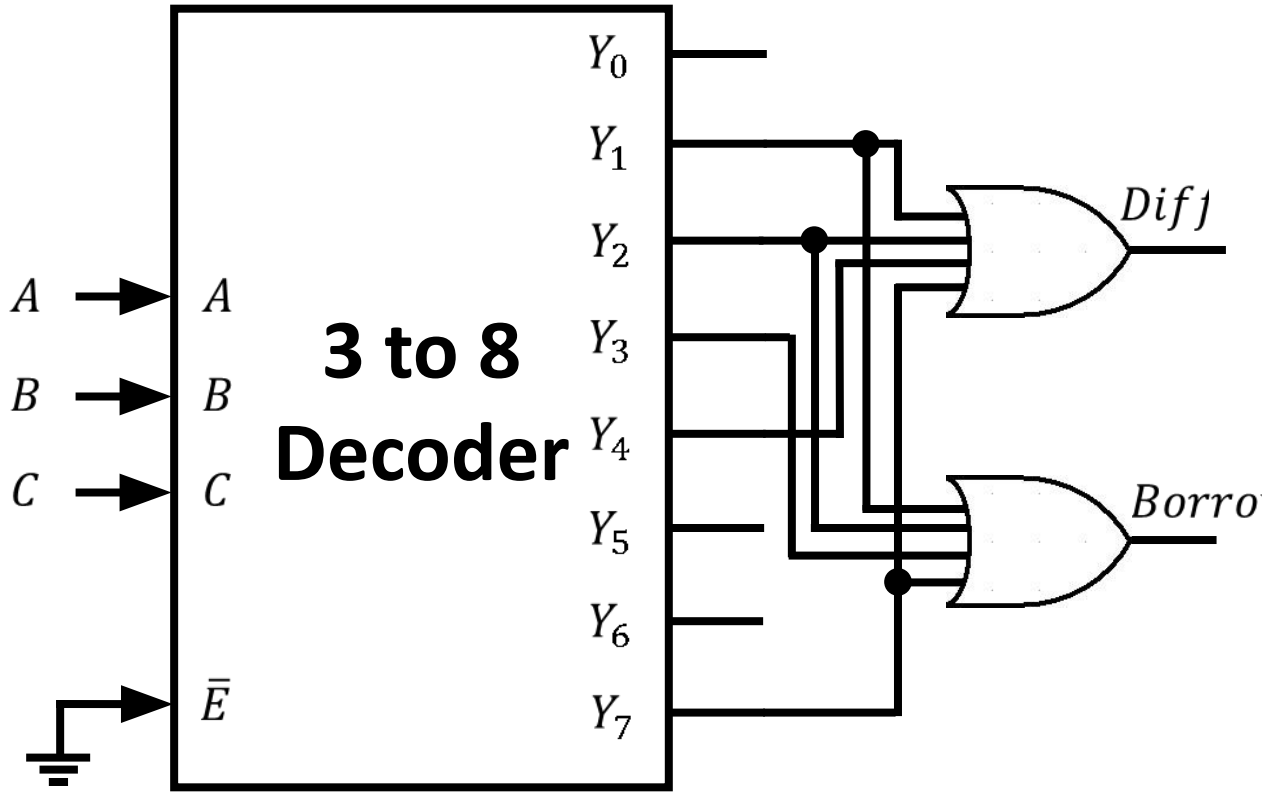
$$\text{Sum} = \sum_m(1,2,4,7)$$

$$\text{Carry} = \sum_m(3,5,6,7)$$

Implementation of Full Subtractor using Demultiplexer

Truth Table

Decimal	Inputs			Outputs	
	A	B	C	Difference	Borrow
0	0	0	0	0	0
1	0	0	1	1	1
2	0	1	0	1	1
3	0	1	1	0	1
4	1	0	0	1	0
5	1	0	1	0	0
6	1	1	0	0	0
7	1	1	1	1	1



Output Expressions are,

$$\text{Difference} = \sum_m(1,2,4,7)$$

$$\text{Borrow} = \sum_m(1,2,3,7)$$

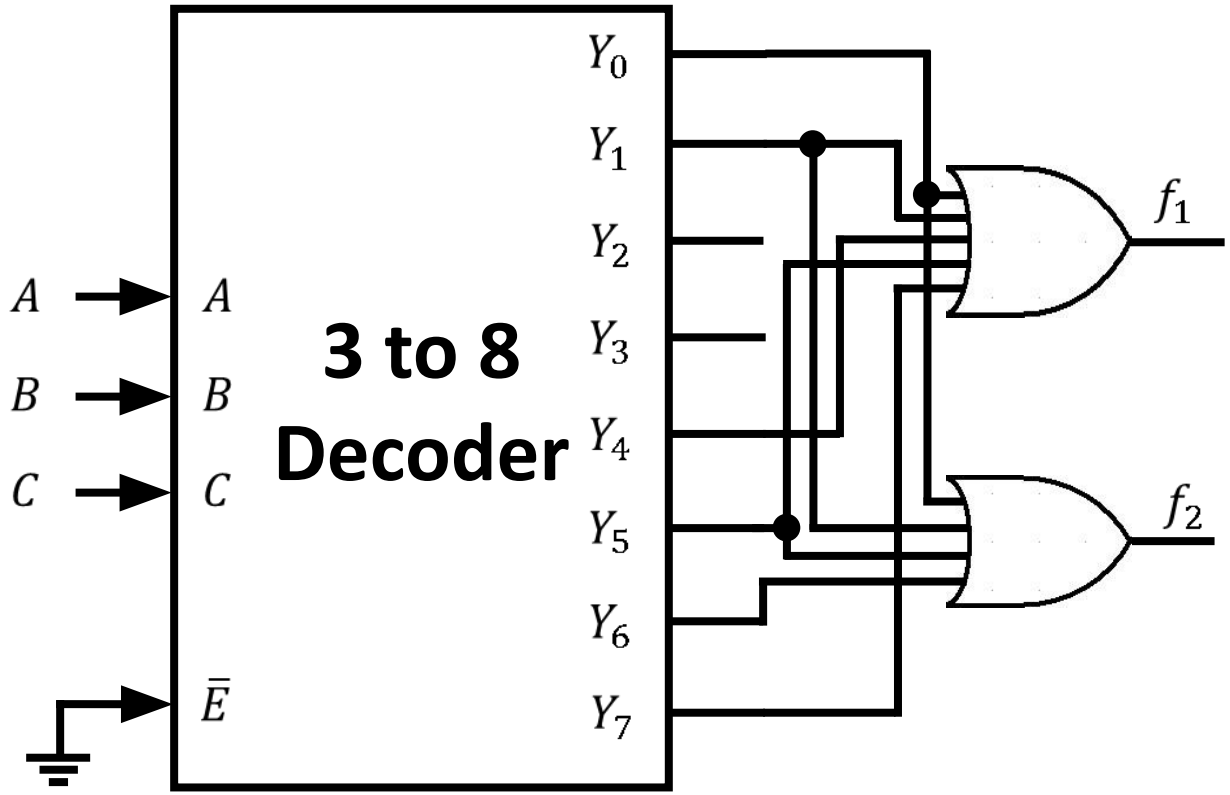
Implementation of given Boolean function using Decoder

$$f_1 = \sum_m(0,1,4,5,7) \qquad f_2 = \sum_m(0,1,5,6)$$

Given Expressions are,

$$f_1 = \sum_m(0,1,4,5,7)$$

$$f_2 = \sum_m(0,1,5,6)$$



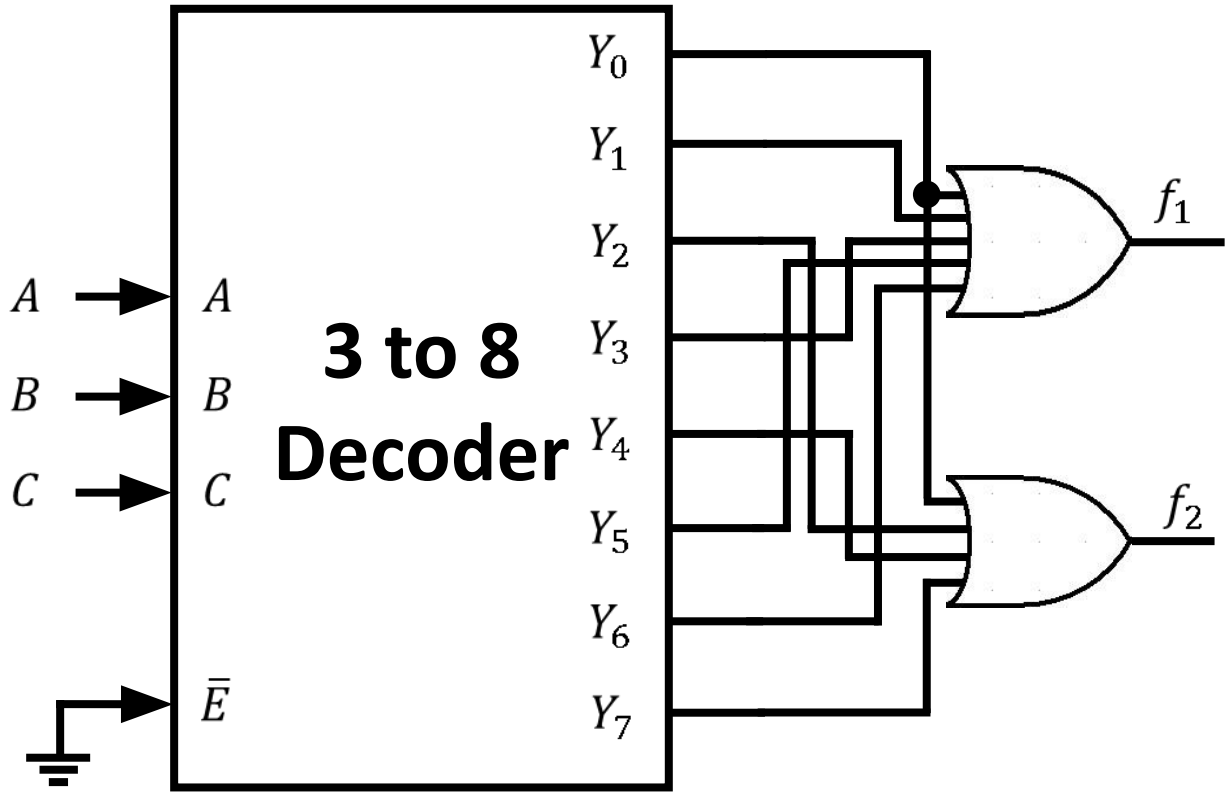
Implementation of given Boolean function using Decoder

$$f_1 = \sum_m(0,1,3,5,6) \qquad f_2 = \sum_m(0,2,4,7)$$

Given Expressions are,

$$f_1 = \sum_m(0,1,3,5,6)$$

$$f_2 = \sum_m(0,2,4,7)$$



Applications of Decoder

- It can be used to implement combinational circuit
- It can be used to convert BCD into 7-segment code
- It is used in memories to select particular register

THANK YOU…

ANY QUERIEES???...

ENCODER

Prepared By

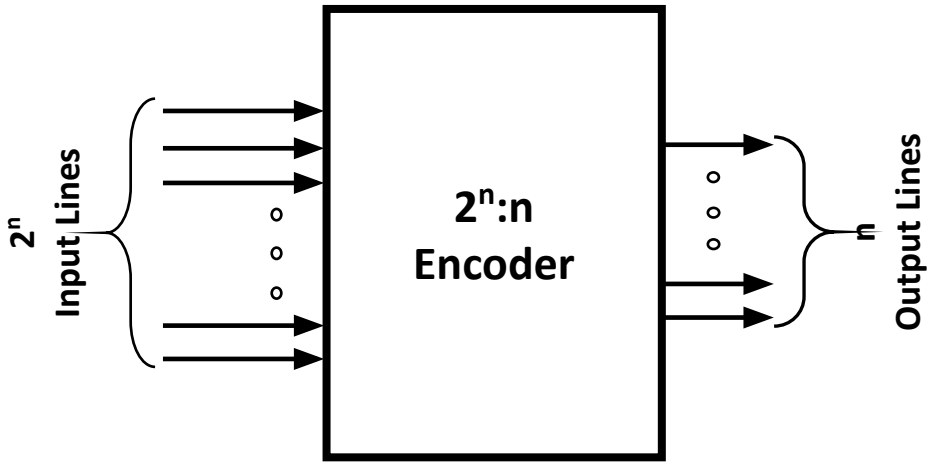
Mr.P.KANAKARAJ

AP/EEE

SRMIST-KTR

Encoder:

- It is a combinational logic circuit.
- An encoder has 2^n input lines and n output lines.
- The n output lines generate the binary code for the possible 2^n input lines.
- Input – 2^n
- Output – n
- Encoder – 2^n : n



Four to Two Line Binary Encoder

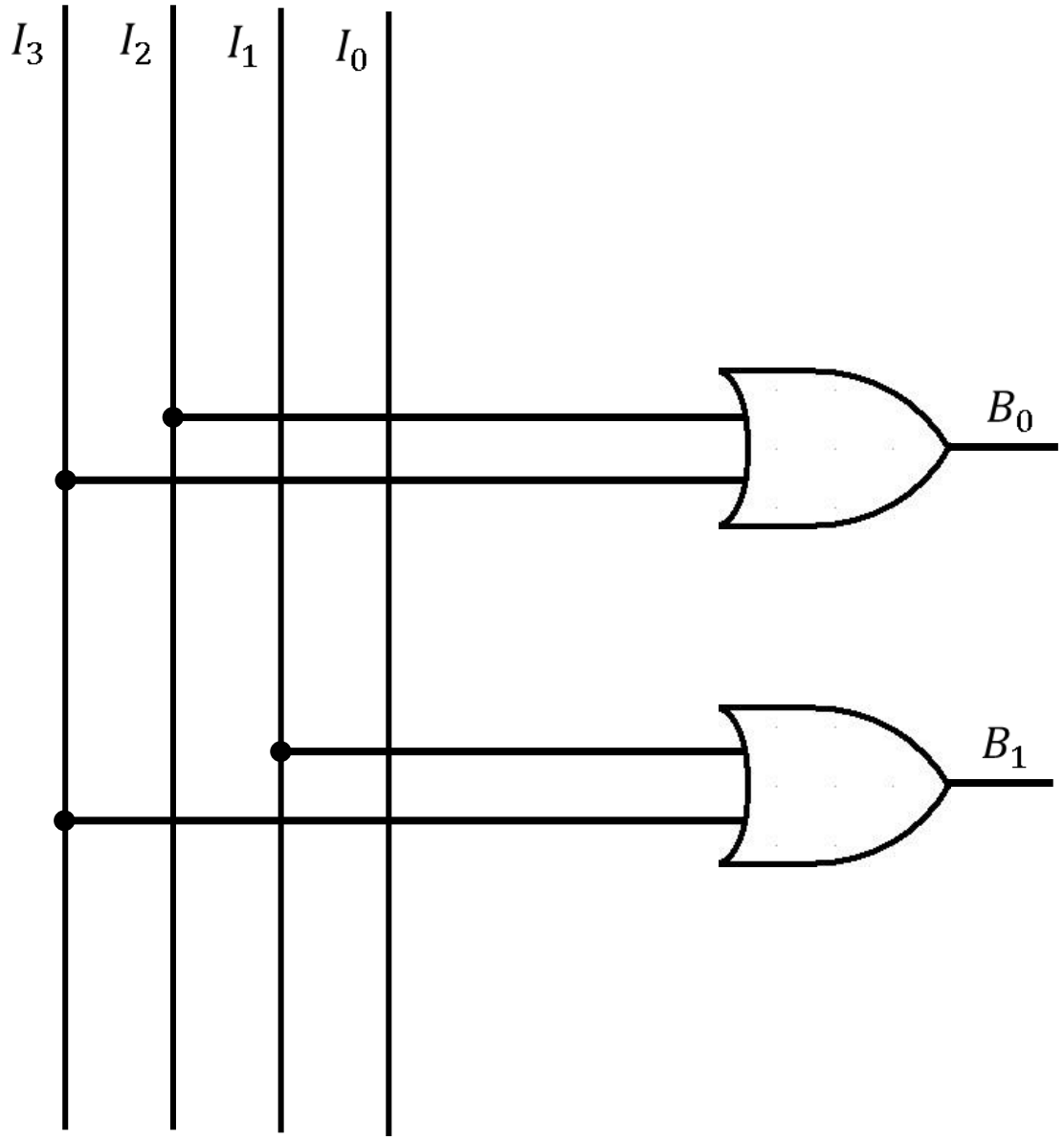
Truth Table

Decimel Value	Input				Output	
0	0	0	0	1	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	0
3	1	0	0	0	1	1

- Input – 4
- Output – 2
- Outputs are,

$$B_0 = I_2 + I_3$$

$$B_1 = I_1 + I_3$$



Eight to Three Line Binary Encoder (Octal to Binary Encoder

Truth Table

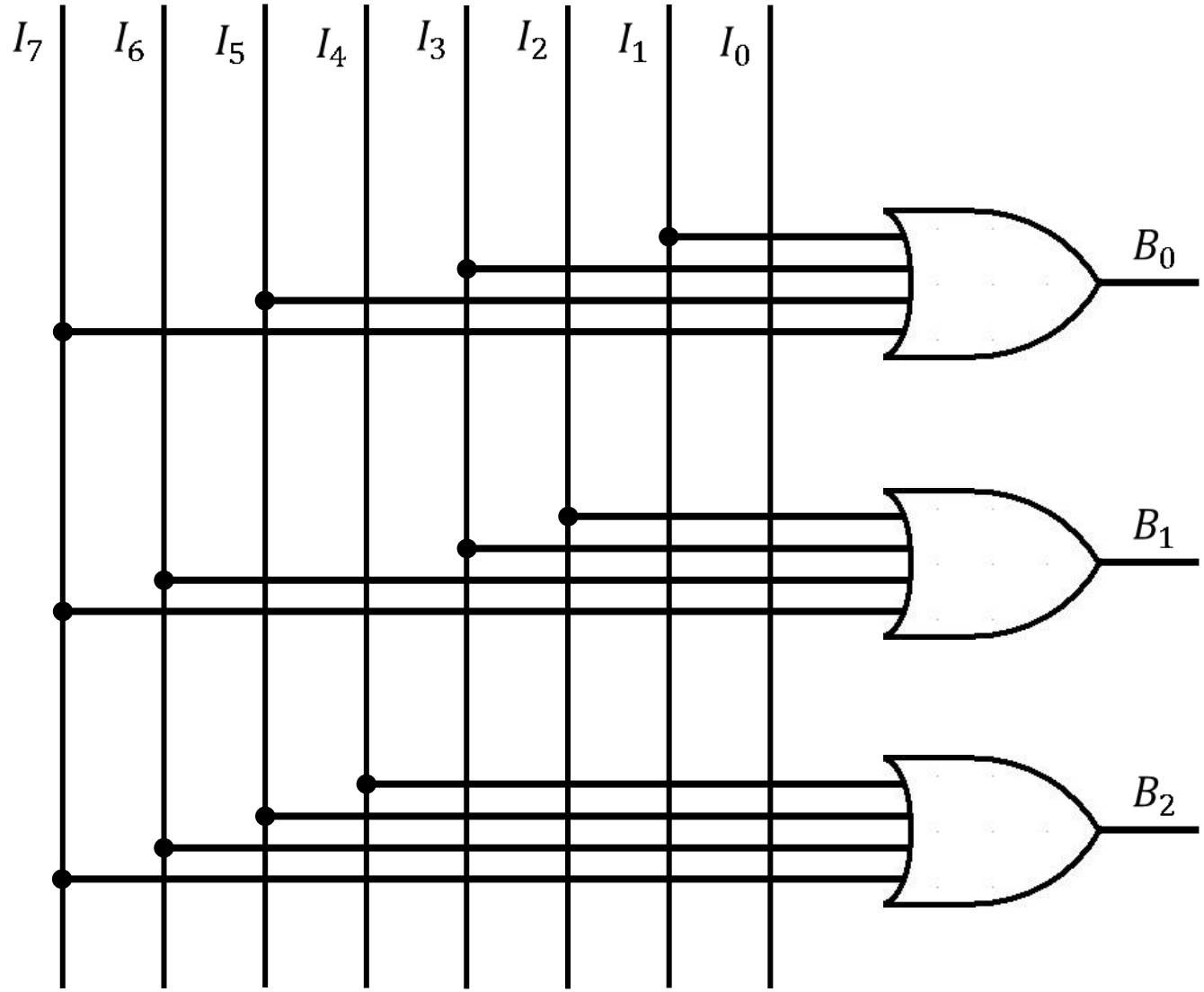
Decimal Value	Input								Output		
0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	(1)
2	0	0	0	0	0	1	0	0	0	(1)	0
3	0	0	0	0	1	0	0	0	0	(1)	(1)
4	0	0	0	1	0	0	0	0	(1)	0	0
5	0	0	1	0	0	0	0	0	(1)	0	(1)
6	0	1	0	0	0	0	0	0	(1)	(1)	0
7	1	0	0	0	0	0	0	0	(1)	(1)	(1)

- Input – 8
- Output – 3
- Outputs are,

$$B_0 = I_1 + I_3 + I_5 + I_7$$

$$B_1 = I_2 + I_3 + I_6 + I_7$$

$$B_2 = I_4 + I_5 + I_6 + I_7$$



Priority Encoder:

- It is an encoder circuit that includes the priority function.
- In that priority encoder, if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

Truth Table

Input				Output		
0	0	0	0			0
1	0	0	0	0	0	1
	1	0	0	0	1	1
		1	0	1	0	1
			1	1	1	1

K-Map Simplification

B_1:

D_0D_1 \ D_2D_3	00	01	11	10
00	X	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

$$B_1 = D_2 + D_3$$

B_0:

D_0D_1 \ D_2D_3	00	01	11	10
00	X	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

$$B_0 = D_3 + D_1\overline{D_2}$$

V:

D_0D_1 \ D_2D_3	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

$$V = D_0 + D_1 + D_2 + D_3$$

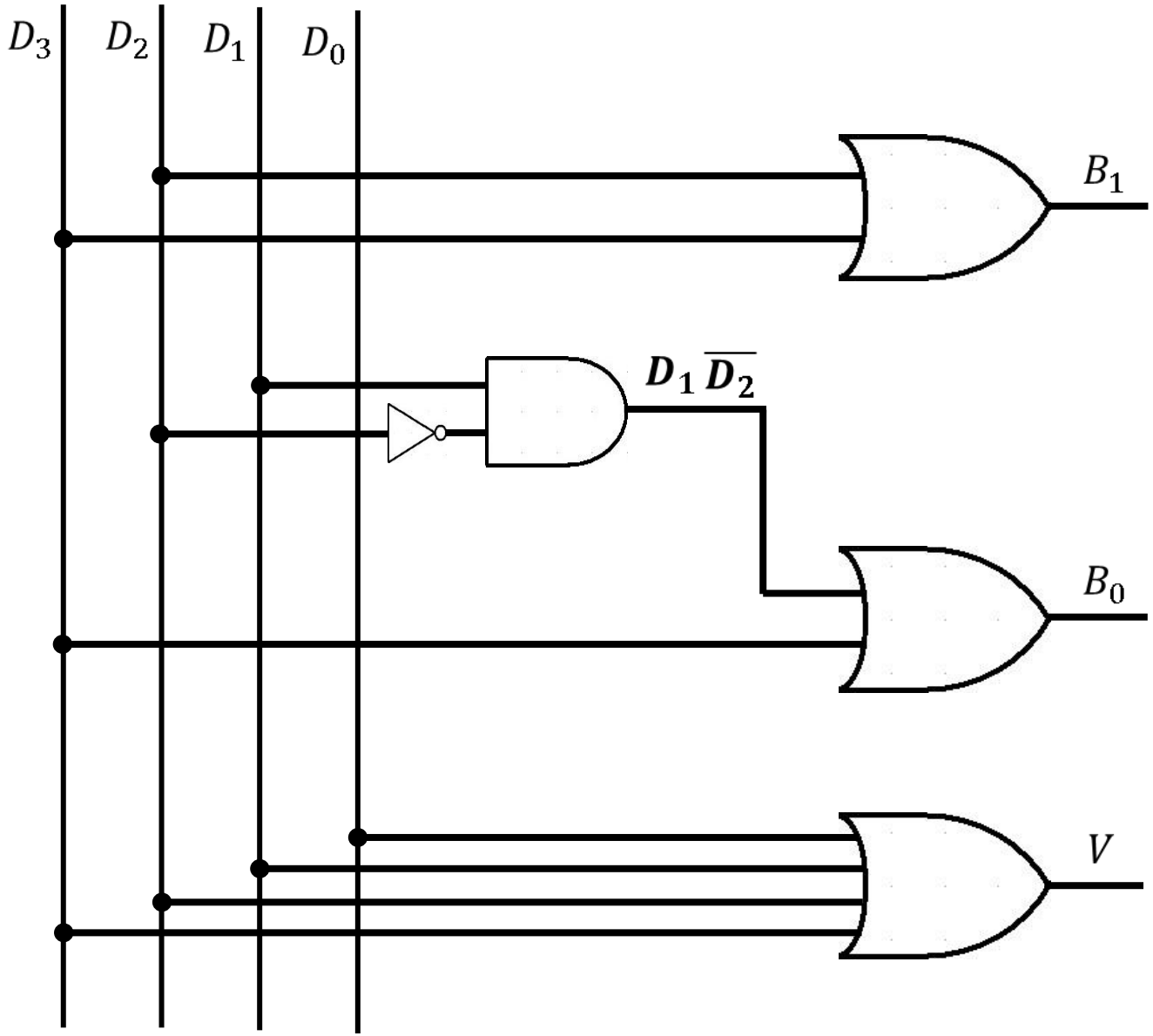
Logic Diagram for Priority Encoder

- Expressions are,

$$B_1 = D_2 + D_3$$

$$B_0 = D_3 + D_1\overline{D_2}$$

$$V = D_0 + D_1 + D_2 + D_3$$



THANK YOU…

ANY QUERIEES???...

Unit - 3

Combinational Logic Circuits

Contents

- Binary adder
- Binary adder as subtractor
- Carry look ahead adder
- Decimal adder
- Magnitude Comparator

Binary adder

Half Adder: is a combinational circuit that performs the addition of two bits, this circuit needs two binary inputs and two binary outputs.

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0
Truth table			

The simplified Boolean function from the truth table:

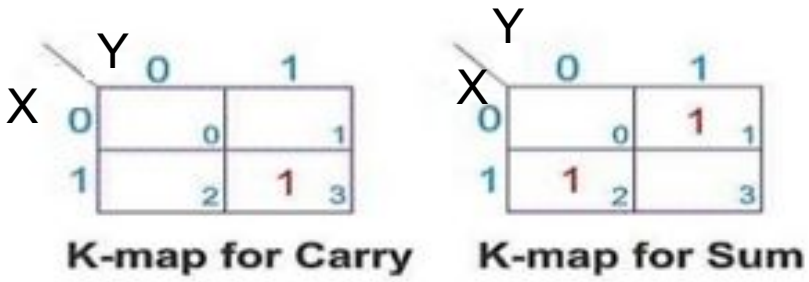
$$\begin{cases} S = \bar{X}Y + X\bar{Y} \\ C = XY \end{cases} \quad 1$$ (Using sum of product form)

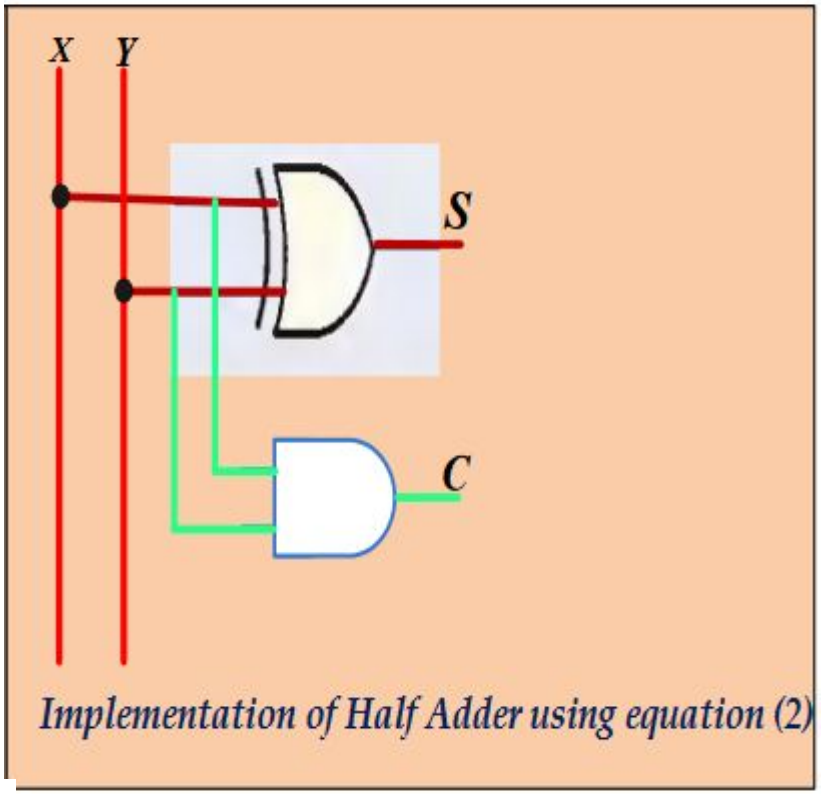
Where S is the sum and C is the carry.

$$\begin{cases} S = X \oplus Y \\ C = XY \end{cases} \quad 2$$ (Using XOR and AND Gates)

Half adder

K-map for half adder

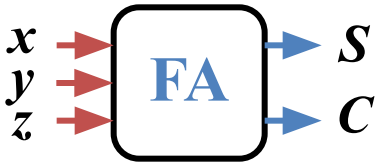




$$\begin{cases} S = X \oplus Y \\ C = XY \end{cases} \quad 2 \} \text{ (Using XOR and AND Gates)}$$

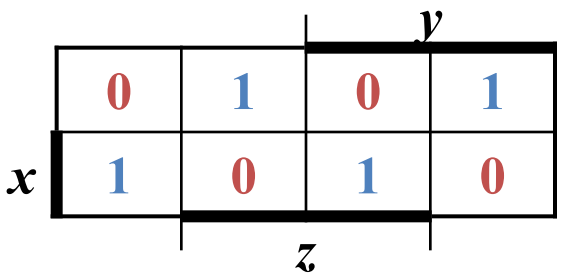
Binary Adder

- Full Adder
  - Adds 1-bit plus 1-bit plus 1-bit
  - Produces Sum and Carry

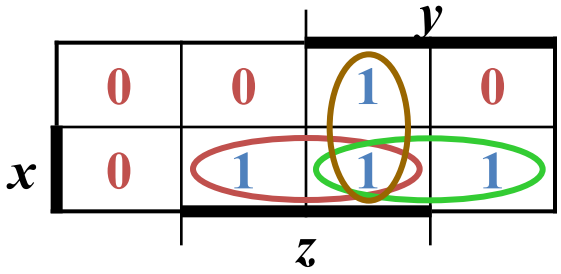


$$\begin{array}{rr} & x \\ + & y \\ + & z \\ \hline C & S \end{array}$$

x y z	C S
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



$$S = xy'z' + x'yz' + x'y'z + xyz = x \oplus y \oplus z$$



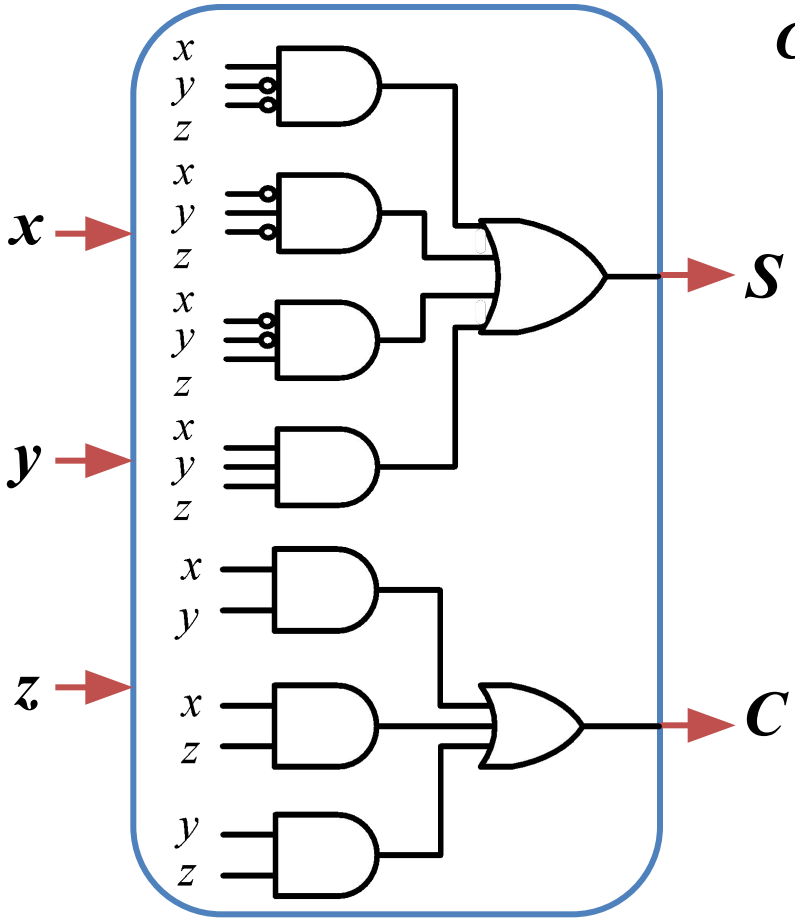
$$C = xy + xz + yz$$

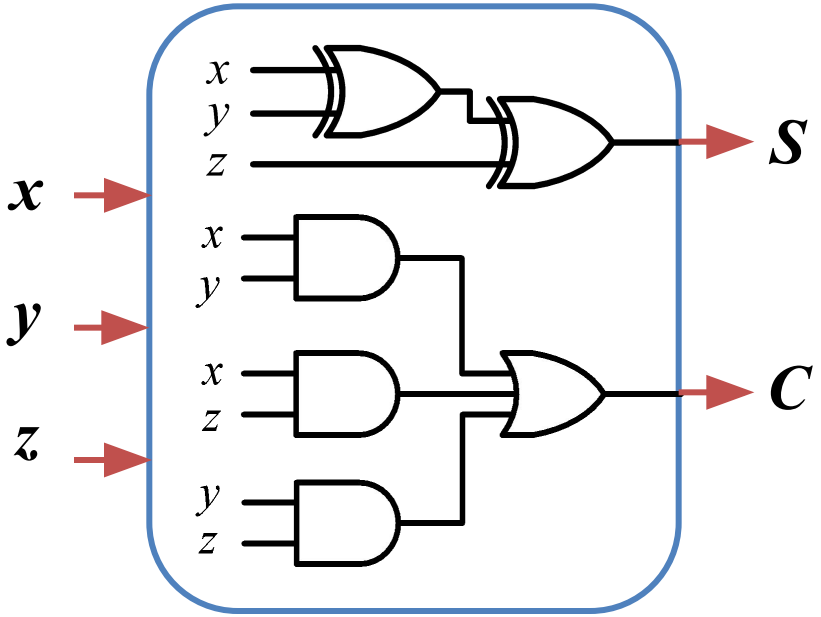
Binary Adder

- Full Adder

$$S = xy'z' + x'yz' + x'y'z + xyz = x \oplus y \oplus z$$

$$C = xy + xz + yz$$





Binary Adder

Full adder design from two half adder

The sum:

$$S = \bar{X}\bar{Y}C_{in} + \bar{X}Y\overline{C_{in}} + X\bar{Y}\,\overline{C_{in}} + XYC_{in}$$

$$= \overline{C_{in}}(\bar{X}Y + X\bar{Y}) + C_{in}(\bar{X}\,\bar{Y} + XY)$$

$$= \overline{C_{in}}(\bar{X}Y + X\bar{Y}) + C_{in}\overline{(\bar{X}Y + X\bar{Y})}$$

$$S = C_{in} \oplus (X \oplus Y)$$

The carry output:

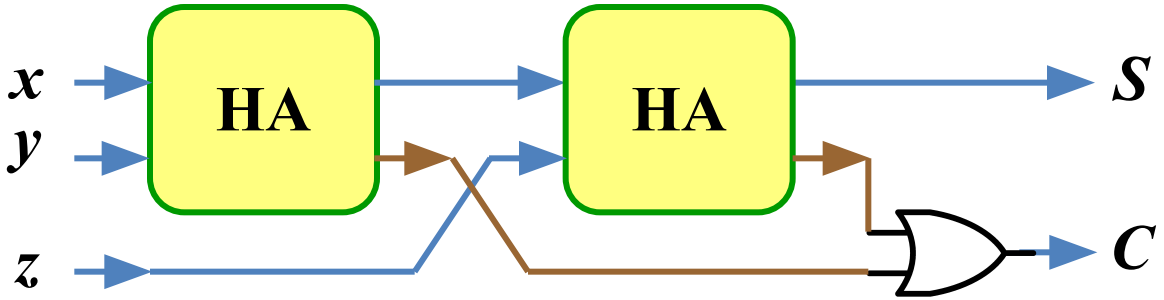
$$C_{out} = \bar{X}YC_{in} + X\bar{Y}C_{in} + XYC_{in} + XY\,\overline{C_{in}}$$

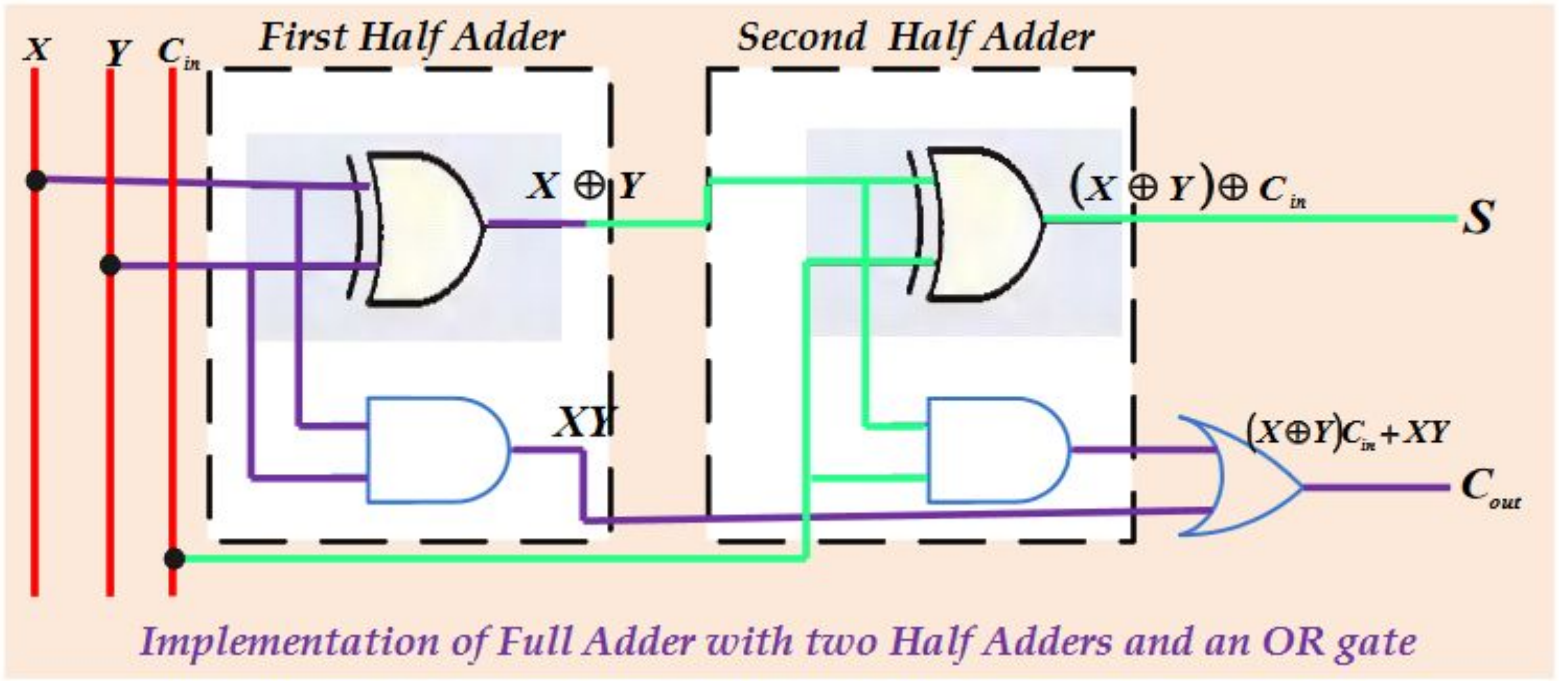
$$= C_{in}(\bar{X}Y + X\bar{Y}) + XY(C_{in} + \overline{C_{in}})$$

$$C_{out} = C_{in} \cdot (X \oplus Y) + XY$$

Binary Adder

- Full Adder



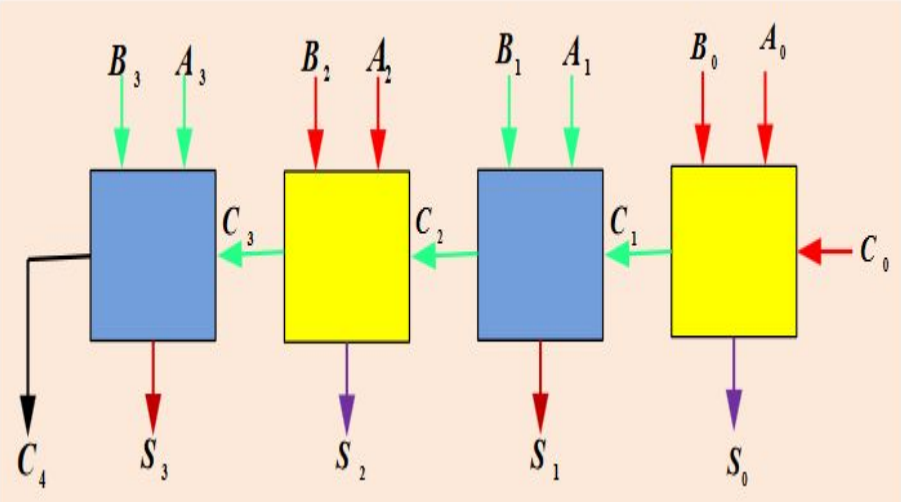


Implementation of Full Adder with two Half Adders and an OR gate

Binary Adder

Binary Adder (Asynchronous Ripple-Carry Adder)

- A binary adder is a digital circuit that produces the ***arithmetic sum of two binary numbers***.
- A binary adder can be constructed with ***full adders connected in cascade*** with the output carry form each full adder connected to the input carry of the next full adder in the chain.
- The ***four-bit adder*** is a typical example of a ***standard component*** .It can be used in many application involving arithmetic operations.



$$\begin{array}{llllll} & c_3 & c_2 & c_1 & & \\ + & x_3 & x_2 & x_1 & x_0 & \\ + & y_3 & y_2 & y_1 & y_0 & \\ \hline Cy & S_3 & S_2 & S_1 & S_0 & \end{array}$$

Example:

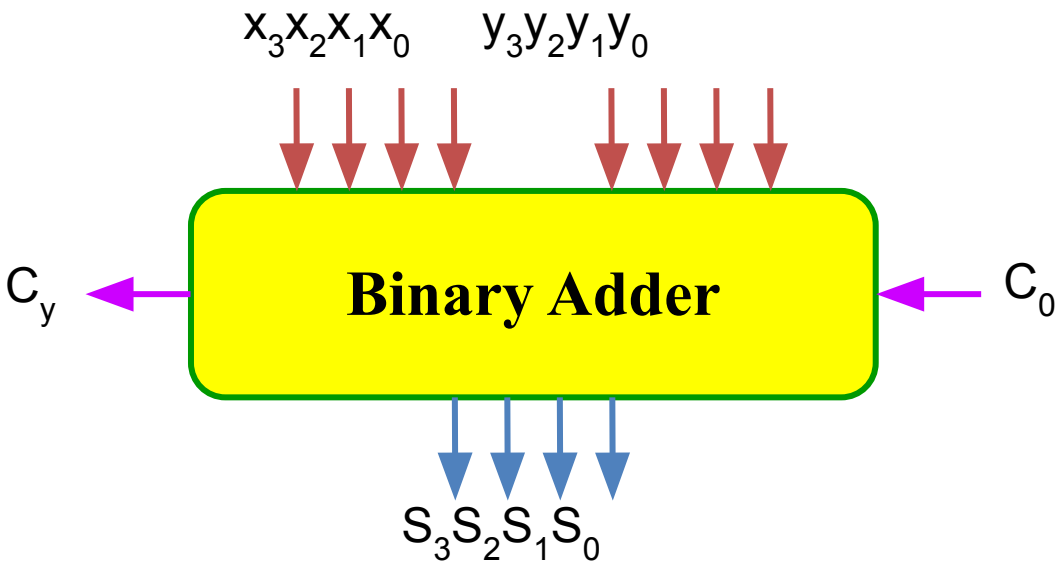
$A + B$ $(A = 1011)$ and $(B = 0011)$

Subscript i	3	2	1	0	
Input Carry	0	1	1	0	C_i
A	1	0	1	1	A_i
+					
B	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output Carry	0	0	1	1	C_{i+1}

$C_0 = 0$

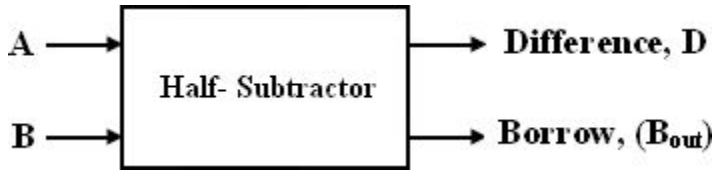
Binary Adder

- The input carry to the adder is C_0 and it ripples through the full adders to the output carry C_4.
- n-bit binary adder requires n full adders.



Binary Subtractor

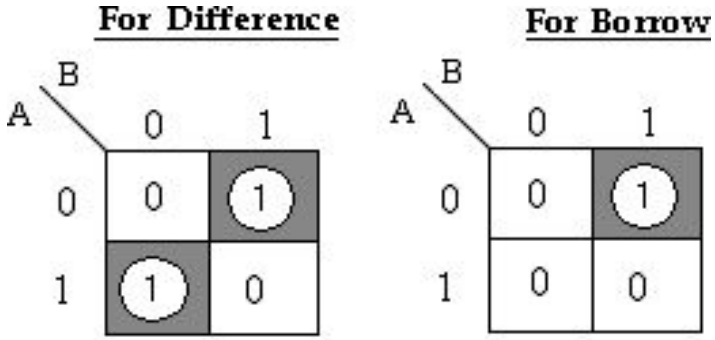
Block schematic of half-subtractor



Truth table

Input		Output	
A	B	Difference (D)	Borrow (Bout)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

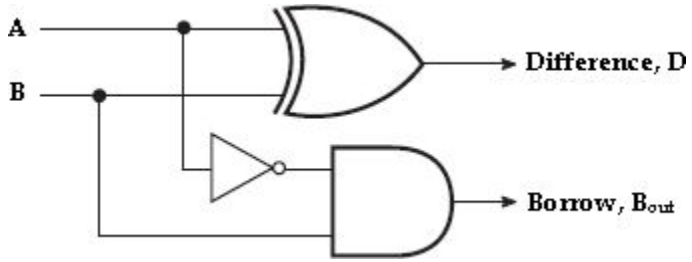
Logical expression



Difference = AB'+ A'B
= A⊕B

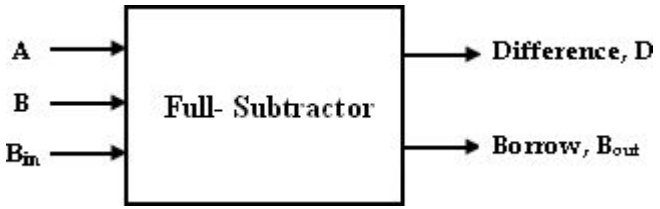
Borrow = $\overline{A}$.B

Logical design



Binary Subtractor

Block schematic of full-subtractor

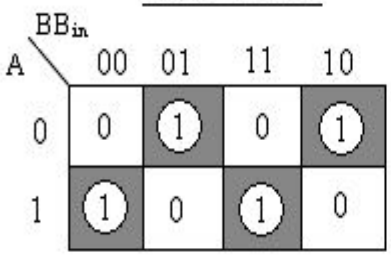


Truth table

Inputs			Outputs	
A	B	B_{in}	Difference(D)	Borrow(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**

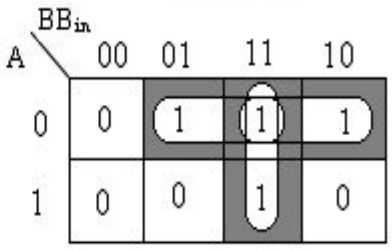
Logical expression

For Difference



Difference, $D = A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$

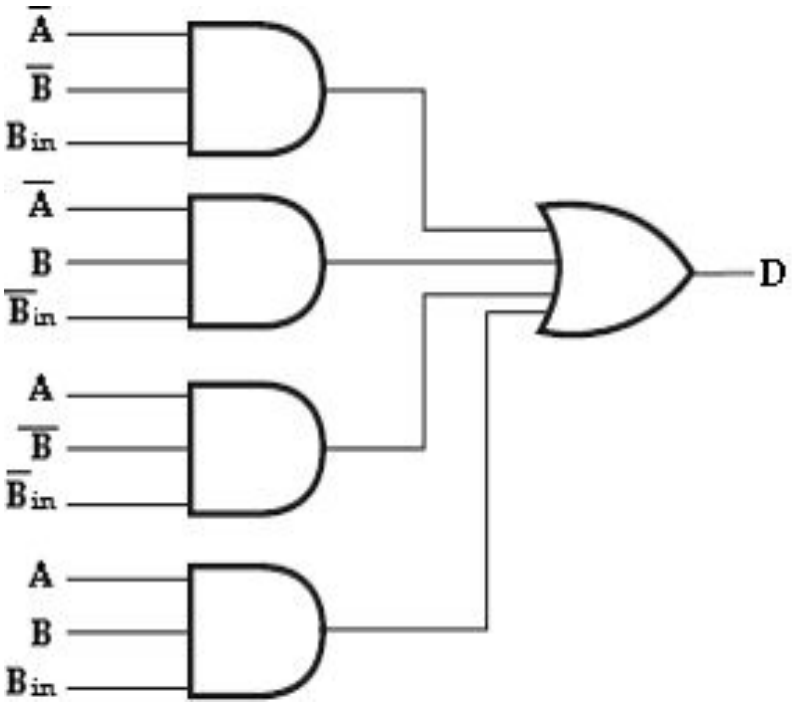
For Borrow

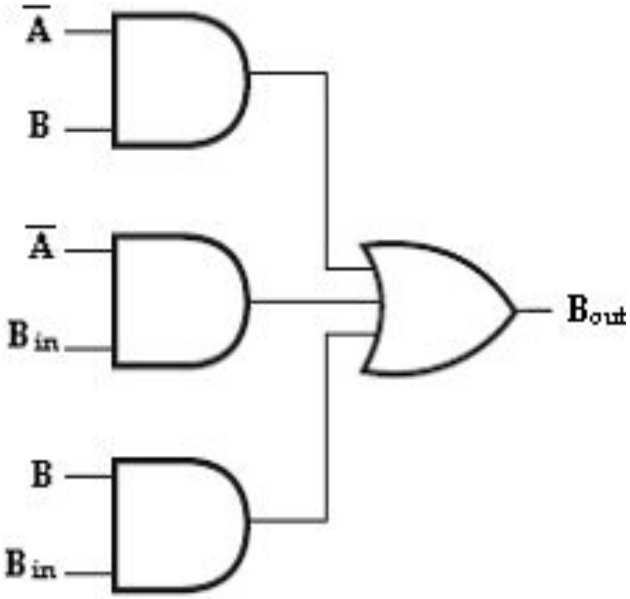


Borrow, $B_{out} = A'B + A'B_{in} + BB_{in}$

Binary Subtractor

Logical design





Binary Subtractor

- **1's Complement of a Binary Number**

There is a simple algorithm to convert a binary number into 1's complement. To get 1's complement of a binary number, simply invert the given number.

- **2's Complement of a Binary Number**

There is a simple algorithm to convert a binary number into 2's complement. To get 2's complement of a binary number, simply invert the given number and add 1 to the least significant bit (LSB) of given result.

Example

Binary representation of 3 is: 0 0 1 1

1's Complement of 3 is: 1 1 0 0

2's Complement of 3 is: (1's Complement + 1) i.e.

1 1 0 0 (1's Compliment)

+ 1

1 1 0 1 (2's Complement i.e. -3)

Now 2 + (-3) = 0 0 1 0 (2 in binary)

+ 1 1 0 1 (-3)

1 1 1 1 (-1)

Now, to check whether it is -1 or not simply. takes 2's Complement of -1 and kept -ve sign as it is.

-1 = 1 1 1 1

2's Complement = -(0 0 0 0)

+ 1

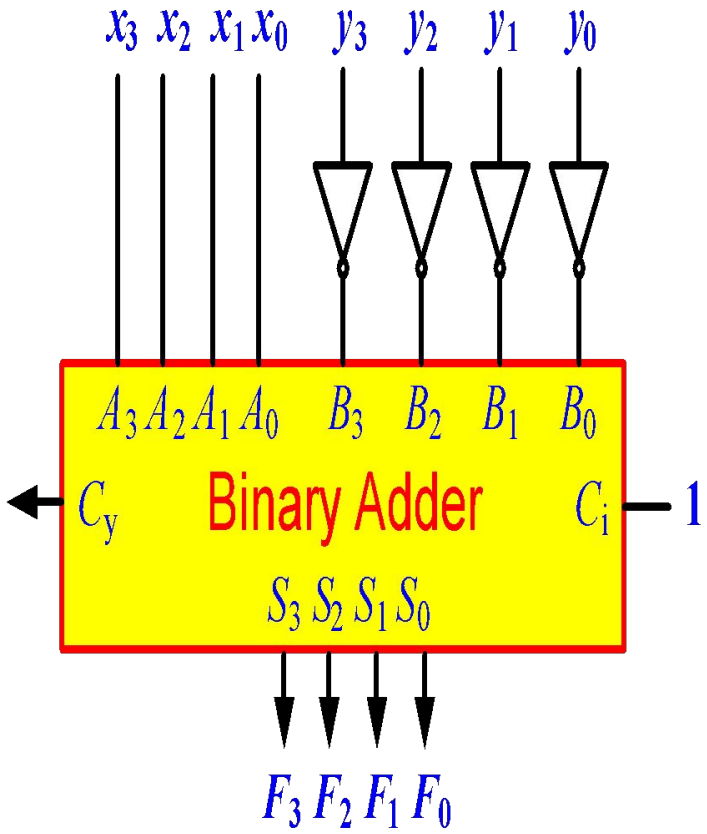
-(0 0 0 1) i.e. -1

Binary Subtractor

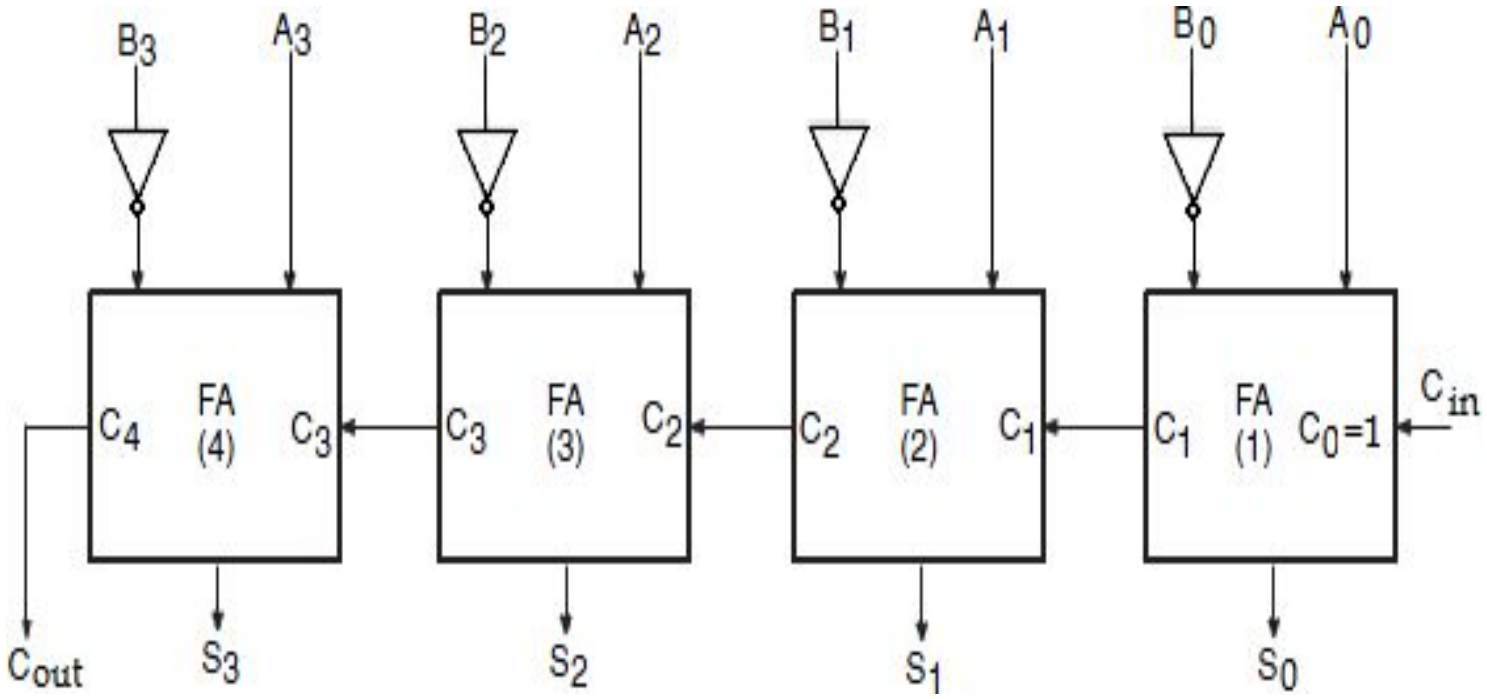
- Use 2's complement with binary adder
  - $x - y = x + (-y) = x + y' + 1$

Binary Subtractor

- To perform the subtraction $-B$, we can use the ***2's complements***, so the subtraction can be converted to addition.
- ***2's complement*** can be obtained by talking the *1's complement* and adding *1* to the *LSD* bit.
  1) *1's complement* can be implemented with inventors.
  2) *1* can be added to the sum through the input carry.
- The circuit for subtracting $A - B$ consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to *1*.

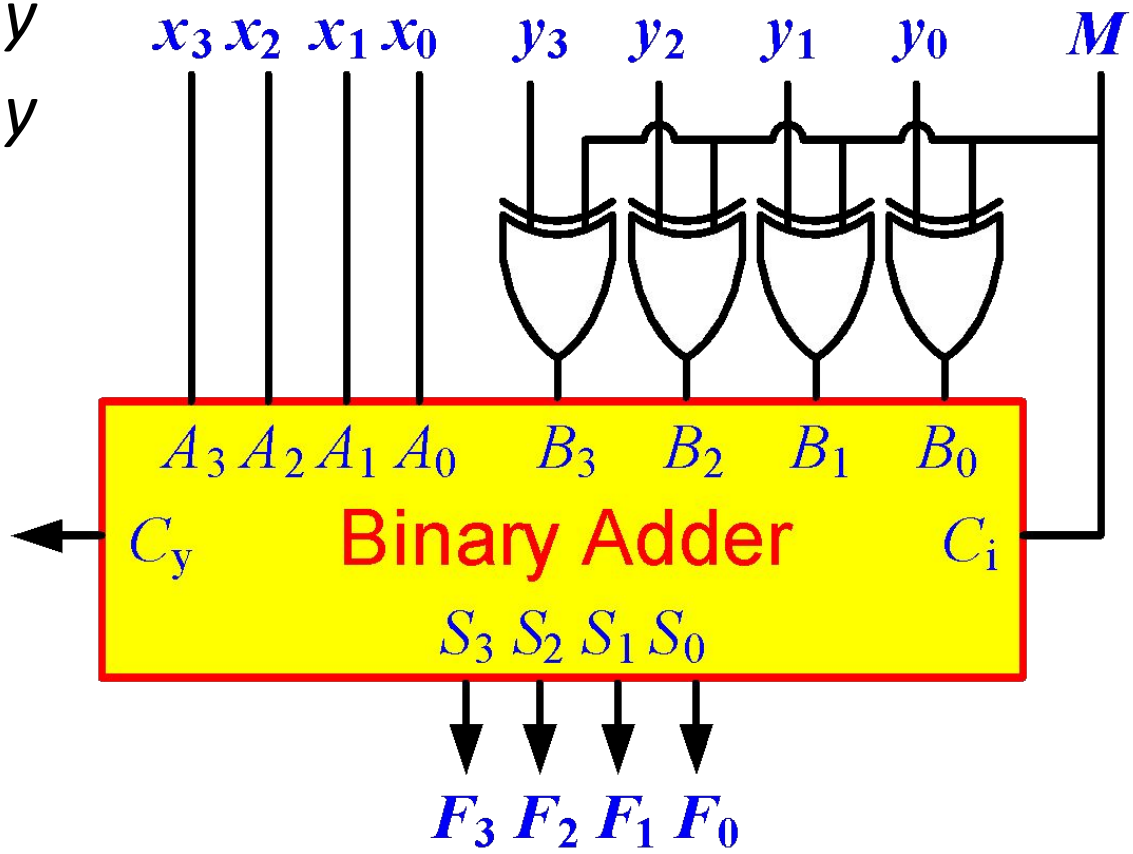


Binary Subtractor



Binary Adder/Subtractor

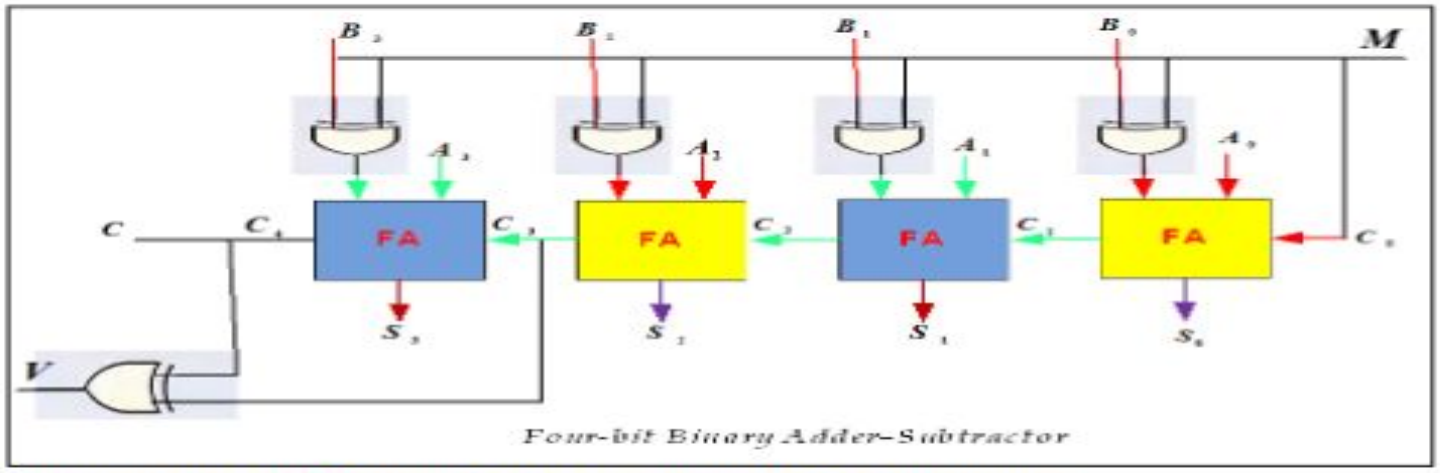
- *M*: Control Signal (Mode)
  - *M*=0 □ *F* = *x* + *y*
  - *M*=1 □ *F* = *x* – *y*



Binary Subtractor

Binary Adder–Subtractor

- The addition and subtraction operations can be combined into one circuit with one common binary adder by including an *exclusive-OR* gate with each full-adder.



Four-bit Binary Adder–Subtractor

The mode input M controls the operation as the following:

- ($M = 0$ →adder.
- $M = 1$ → subtractor.

- Each *XOR* gate receives M signal and B
  - When $M = 0$ then $B \oplus 0 = B$ and the carry = 0, then the circuit performs the operation $A + B$.
  - When $M = 1$ then $B \oplus 1 = \overline{B}$ and the carry = 1, then the circuit performs the operation $A - B$.
- The *exclusive-OR* with output V is for detecting an overflow.

Carry look ahead Adder

- The ***carry propagation time*** is an important attribute of the adder because it limits the speed with which two numbers are added.
- To reduce the carry propagation delay time:
  1) Employ faster gates with reduced delays.
  2) Employ the principle of Carry Lookahead Logic.

Proof: *(using carry lookahead logic)*

$$P_i = A_i \oplus B_i$$

$$G_i = A_i B_i$$

The output sum and carry are:

$$S_i = P_i \oplus C_i$$

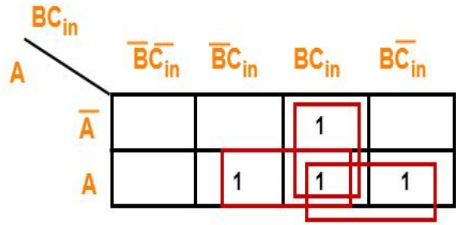
$$C_{i+1} = G_i + P_i C_i$$

Full adder equation

For S:

$S = A \oplus B \oplus C_{in}$

For C_{in}:



AB+ (A XOR B)C

$C_{out} = AB + BC_{in} + C_{in}A$

Carry look ahead Adder

- ✓ G_i-called a ***carry generate***, and it produces a carry of 1 when both A_i and B_i are 1.
- ✓ P_i-called a ***carry propagate***, it determines whether a carry into stage i will propagate into stage $i+1$.
- ✓ The ***Boolean function*** for the carry outputs of each stage and substitute the value of each C_i form the previous equations:

$$\begin{cases} C_0 = input\ carry \\ C_1 = G_0 + P_0C_0 \\ C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) \\ \quad = G_1 + P_1G_0 + P_1P_0C_0 \\ C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0) \\ \quad = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \end{cases}$$

$P_i = A_i \oplus B_i$

$G_i = A_iB_i$

The output sum and carry are:

$S_i = P_i \oplus C_i$

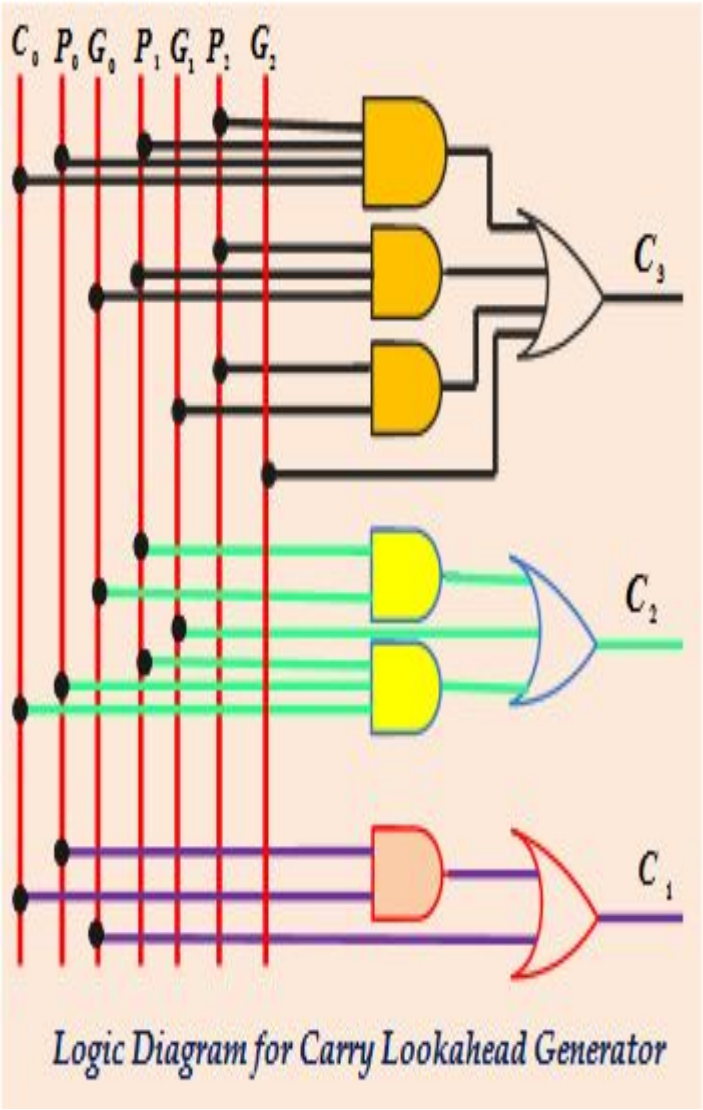
$C_{i+1} = G_i + P_iC_i$

- ➢ The three Boolean functions C_1, C_2 and C_3 are implemented in the ***carry lookahead generator.***

> ***The two level-circuit for the output carry C_4 is not shown, it can be easily derived by the equation.***

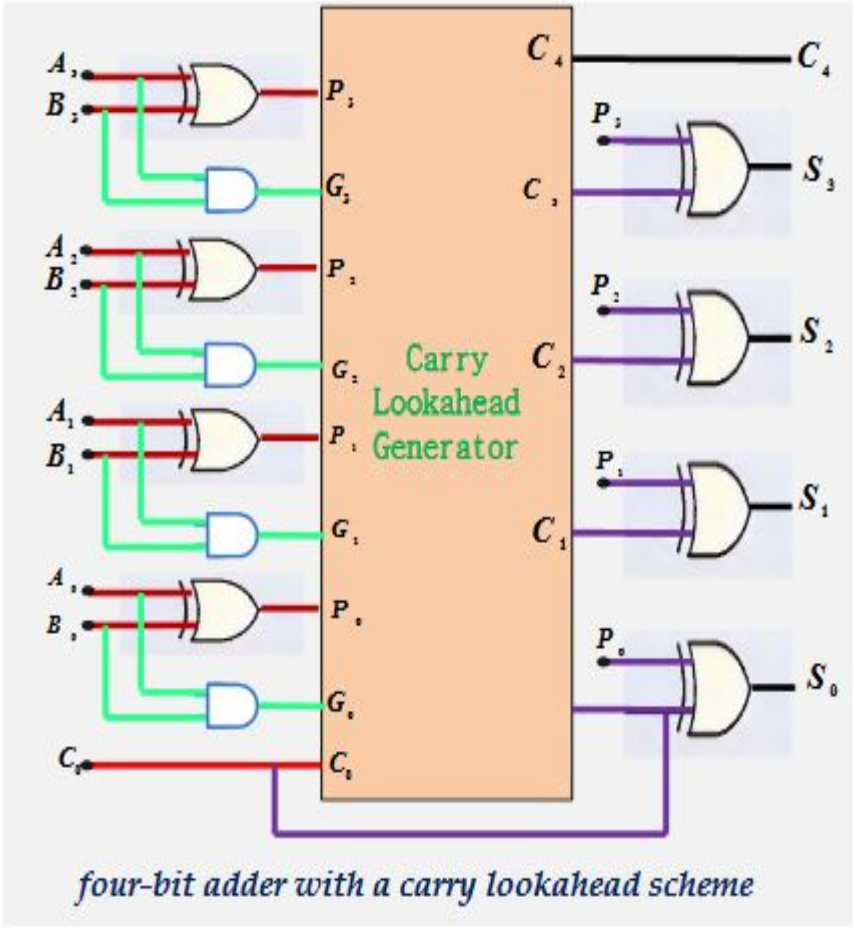
- ➢ C_3 does not have to wait for C_2 and C_1 to propagate, in fact C_3 is propagated at the same time as C_1 and C_2.

Carry look ahead Adder



Logic Diagram for Carry Lookahead Generator

- The construction of a ***four-bit adder with a carry lookahead scheme*** is the following:



four-bit adder with a carry lookahead scheme

Decimal Adder

A decimal adder requires a minimum of 9 inputs and 5 outputs

- 1 digit requires 4-bit
- Input: 2 digits + 1-bit carry
- Output: 1 digit + 1-bit carry

BCD adder

- Perform the addition of two decimal digits in BCD, together with an input carry from a previous stage.
- The output sum cannot be greater than 19 (9+9+1)

BCD Adder

- 4-bits plus 4-bits
- Operands and Result: 0 to 9

$$\begin{array}{rcccc} + & x_3 & x_2 & x_1 & x_0 \\ + & y_3 & y_2 & y_1 & y_0 \\ \hline Cy & S_3 & S_2 & S_1 & S_0 \end{array}$$

$X+Y$	$x_3\,x_2\,x_1\,x_0$	$y_3\,y_2\,y_1\,y_0$	Sum	Cy	$S_3\,S_2\,S_1\,S_0$
0 + 0	0 0 0 0	0 0 0 0	= 0	0	0 0 0 0
0 + 1	0 0 0 0	0 0 0 1	= 1	0	0 0 0 1
0 + 2	0 0 0 0	0 0 1 0	= 2	0	0 0 1 0
0 + 9	0 0 0 0	1 0 0 1	= 9	0	1 0 0 1
1 + 0	0 0 0 1	0 0 0 0	= 1	0	0 0 0 1
1 + 1	0 0 0 1	0 0 0 1	= 2	0	0 0 1 0
1 + 8	0 0 0 1	1 0 0 0	= 9	0	1 0 0 1
1 + 9	0 0 0 1	1 0 0 1	= A	0	1 0 1 0
2 + 0	0 0 1 0	0 0 0 0	= 2	0	0 0 1 0
9 + 9	1 0 0 1	1 0 0 1	= 12	1	0 0 1 0

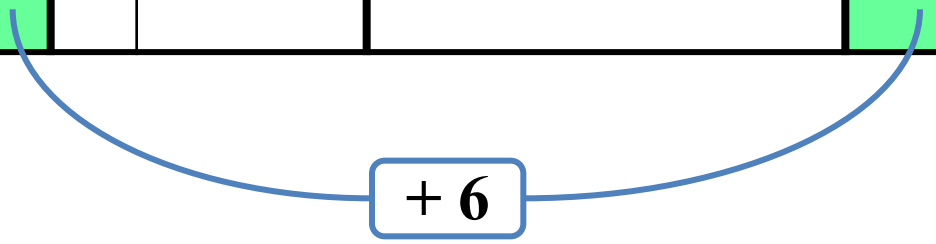
Invalid Code

Wrong BCD Value

0001 1000

BCD Adder

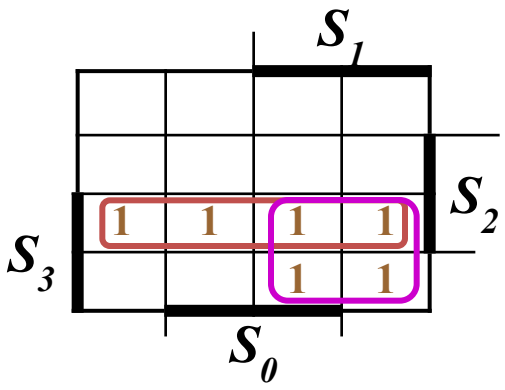
$X+Y$	$x_3\,x_2\,x_1\,x_0$	$y_3\,y_2\,y_1\,y_0$	Sum	Cy	$S_3\,S_2\,S_1\,S_0$	Required BCD Output	Value
9 + 0	1 0 0 1	0 0 0 0	= 9	0	1 0 0 1	0 0 0 0 1 0 0 1	= 9
9 + 1	1 0 0 1	0 0 0 1	= 10	0	1 0 1 0	0 0 0 1 0 0 0 0	= 16
9 + 2	1 0 0 1	0 0 1 0	= 11	0	1 0 1 1	0 0 0 1 0 0 0 1	= 17
9 + 3	1 0 0 1	0 0 1 1	= 12	0	1 1 0 0	0 0 0 1 0 0 1 0	= 18
9 + 4	1 0 0 1	0 1 0 0	= 13	0	1 1 0 1	0 0 0 1 0 0 1 1	= 19
9 + 5	1 0 0 1	0 1 0 1	= 14	0	1 1 1 0	0 0 0 1 0 1 0 0	= 20
9 + 6	1 0 0 1	0 1 1 0	= 15	0	1 1 1 1	0 0 0 1 0 1 0 1	= 21
9 + 7	1 0 0 1	0 1 1 1	= 16	1	0 0 0 0	0 0 0 1 0 1 1 0	= 22
9 + 8	1 0 0 1	1 0 0 0	= 17	1	0 0 0 1	0 0 0 1 0 1 1 1	= 23
9 + 9	1 0 0 1	1 0 0 1	= 18	1	0 0 1 0	0 0 0 1 1 0 0 0	= 24



BCD Adder

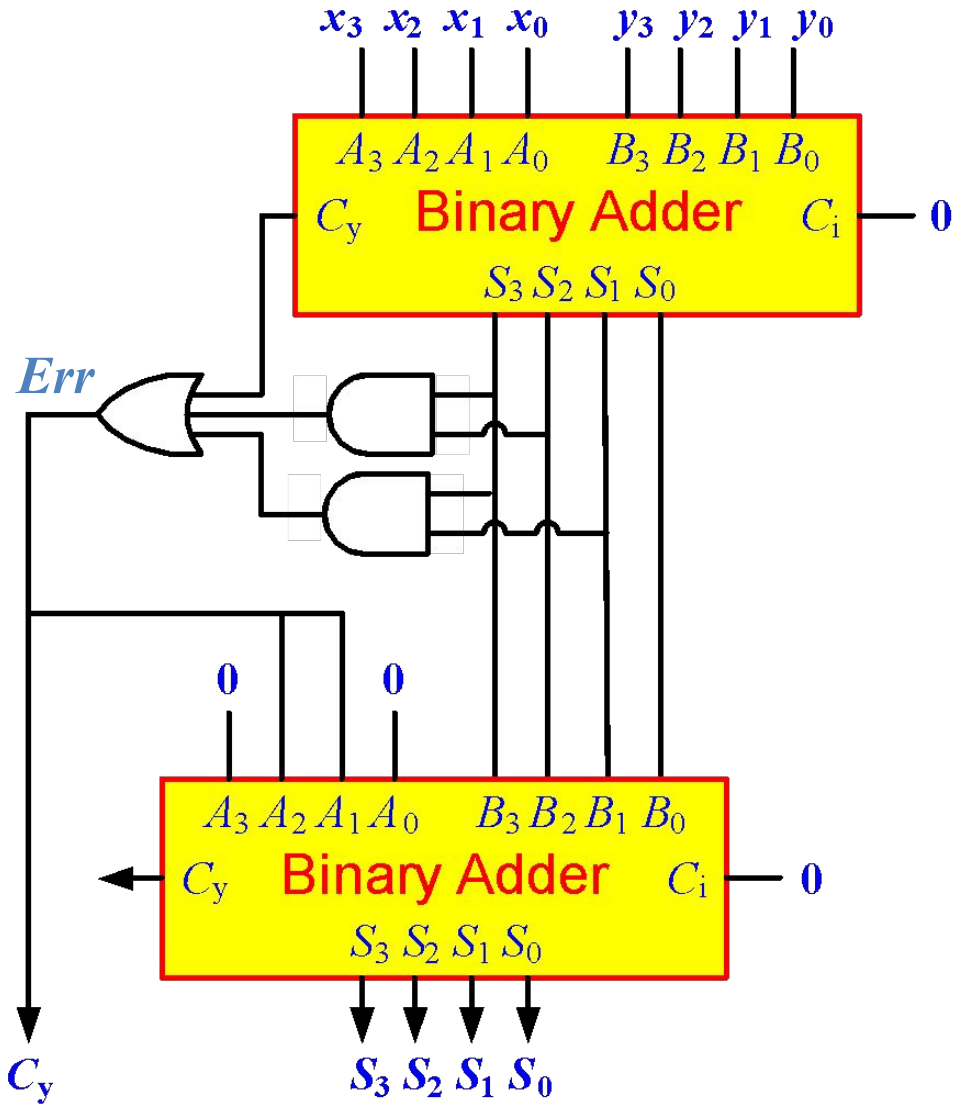
- Correct Binary Adder's Output (+6)
  - If the result is between 'A' and 'F'
  - If Cy = 1

$S_3\ S_2\ S_1\ S_0$	y1
0 0 0 0	0
1 0 0 0	0
1 0 0 1	0
1 0 1 0	1
1 0 1 1	1
1 1 0 0	1
1 1 0 1	1
1 1 1 0	1
1 1 1 1	1



$$Err = S_3 S_2 + S_3 S_1$$

BCD Adder


x_3 x_2 x_1 x_0
y_3 y_2 y_1 y_0
A_3 A_2 A_1 A_0
B_3 B_2 B_1 B_0
C_y
Binary Adder
C_i
0
S_3 S_2 S_1 S_0
Err
0
0
A_3 A_2 A_1 A_0
B_3 B_2 B_1 B_0
C_y
Binary Adder
C_i
0
S_3 S_2 S_1 S_0
C_y
S_3 S_2 S_1 S_0

BCD Adder

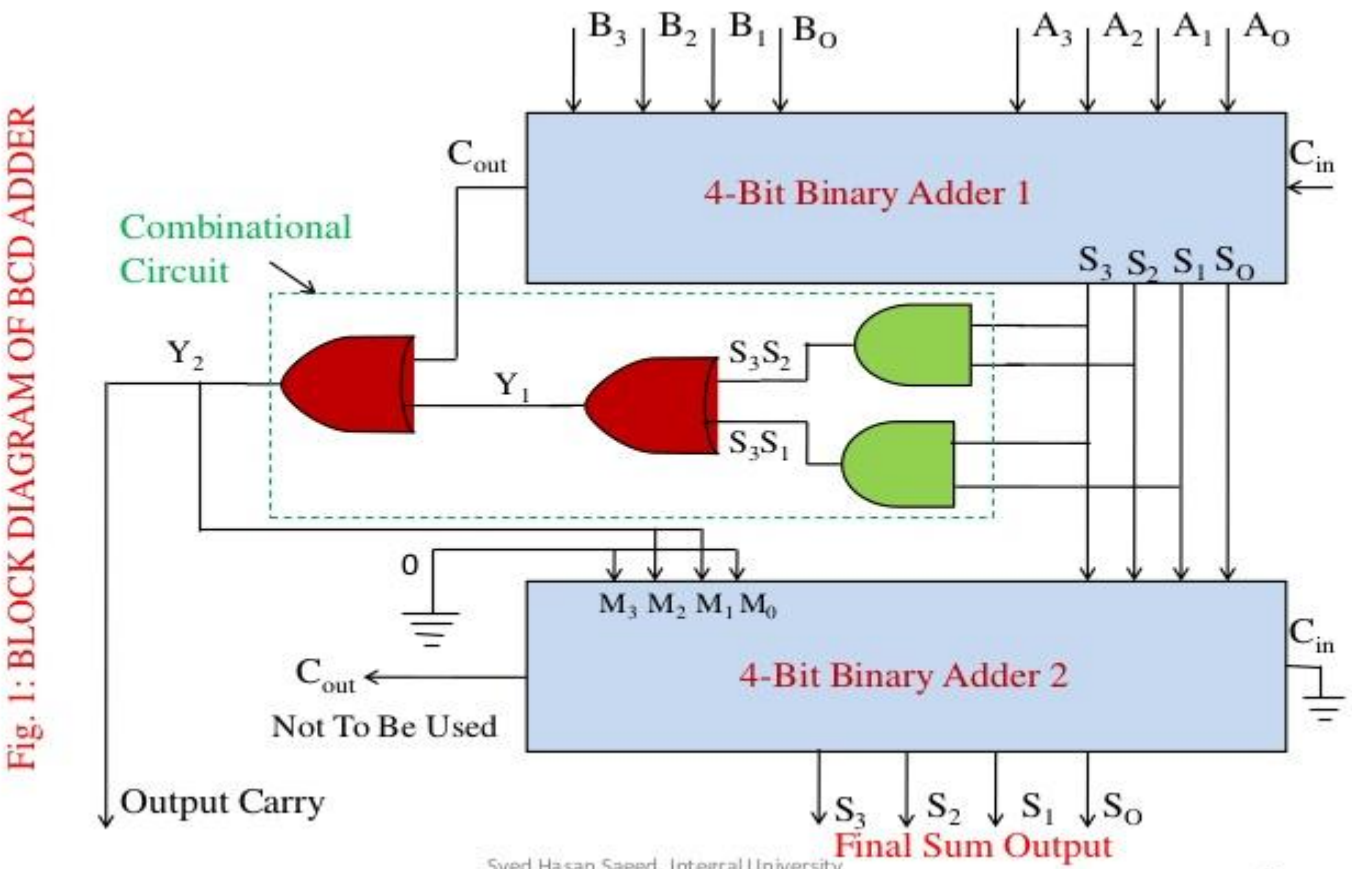
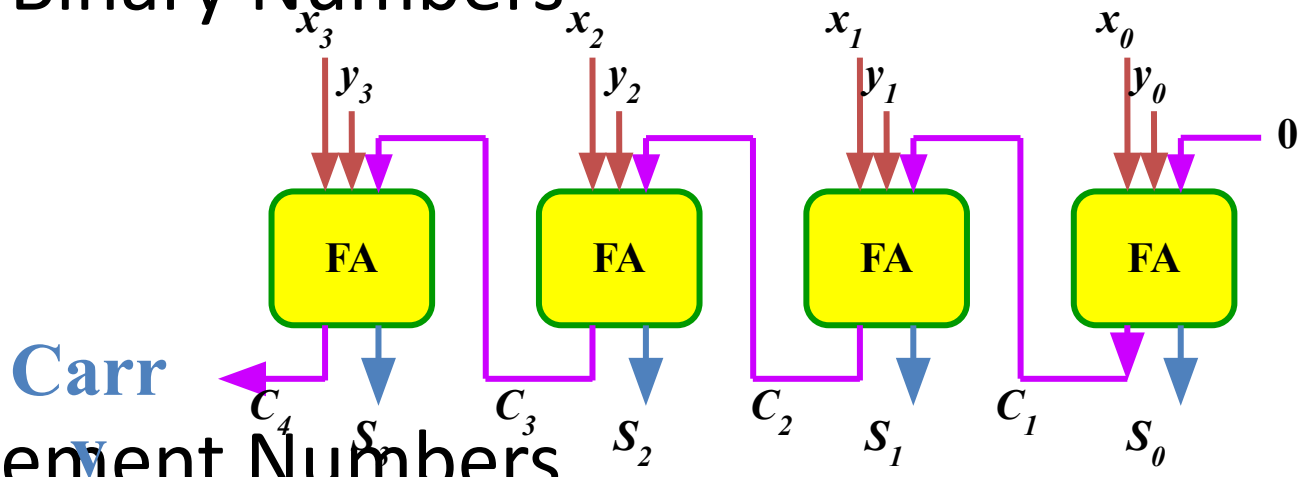
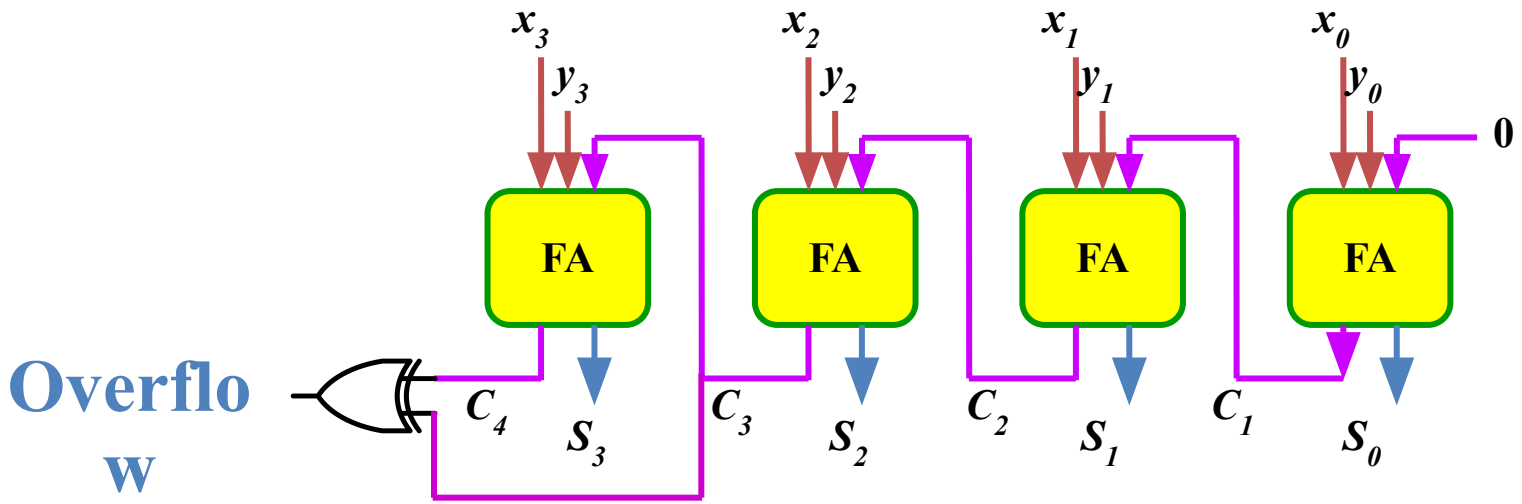


Fig. 1: BLOCK DIAGRAM OF BCD ADDER

Overflow

- Unsigned Binary Numbers
- 2's Complement Numbers





Magnitude Comparator

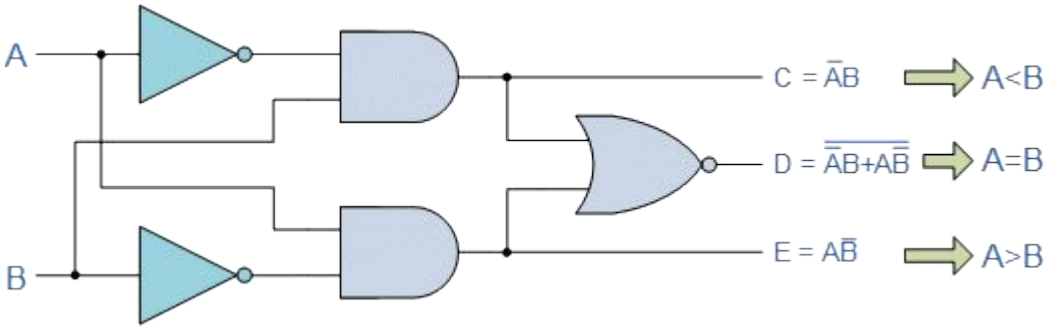
1- Bit magnitude comparator

- A comparator used to compare two bits is called a single bit comparator.
- It consists of two inputs each for two single bit numbers and three outputs to generate less than, equal to and greater than between two binary numbers.

Truth table

A	B	A<B	A=B	A>B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Logic design



Equation from truth table

A<B – A'B

A>B – AB'

A=B – A'B'+AB

Magnitude Comparator

2- Bit magnitude comparator

A comparator used to compare two binary numbers each of two bits is called a 2-bit Magnitude comparator. It consists of four inputs and three outputs to generate less than, equal to and greater than between two binary numbers.

Truth table

INPUT				OUTPUT		
A1	A0	B1	B0	A<B	A=B	A>B
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

K map for 2- Bit magnitude comparator

A>B

A1A0 \ B1B0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
00	1	1	0	0

A =B

A1A0 \ B1B0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
00	0	0	0	1

A <B

A1A0 \ B1B0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
00	0	0	1	0

Design equations

A>B:A1B1' + A0B1'B0' + A1A0B0'

A=B: A1'A0'B1'B0' + A1'A0B1'B0 + A1A0B1B0 + A1A0'B1B0'

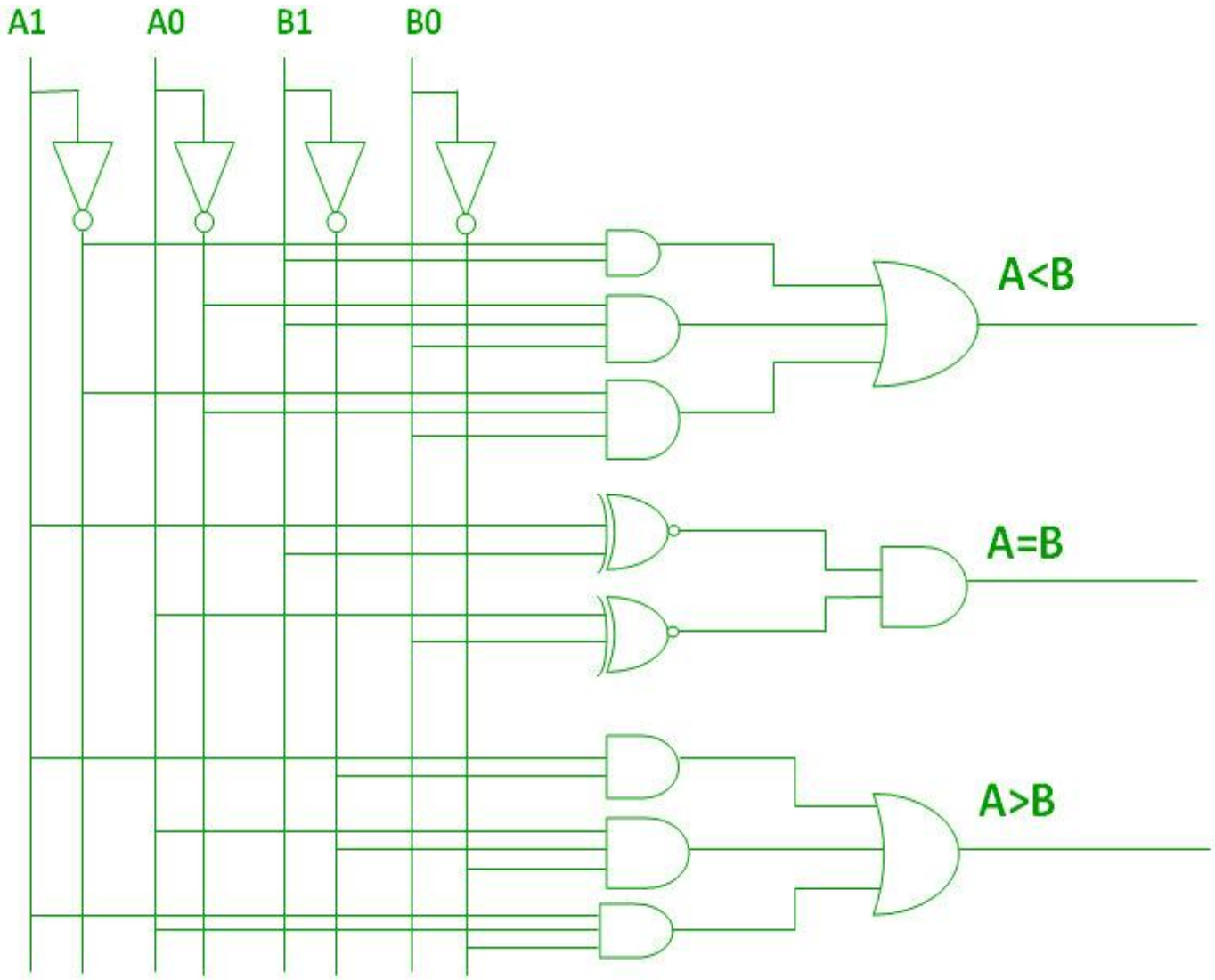
: A1'B1' (A0'B0' + A0B0) + A1B1 (A0B0 + A0'B0')

: (A0B0 + A0'B0') (A1B1 + A1'B1')

: (A0 Ex-Nor B0) (A1 Ex-Nor B1)

A<B:A1'B1 + A0'B1B0 + A1'A0'B0

Logic design of 2 –bit magnitude comparator



ANALOG AND DIGITAL ELECTRONICS

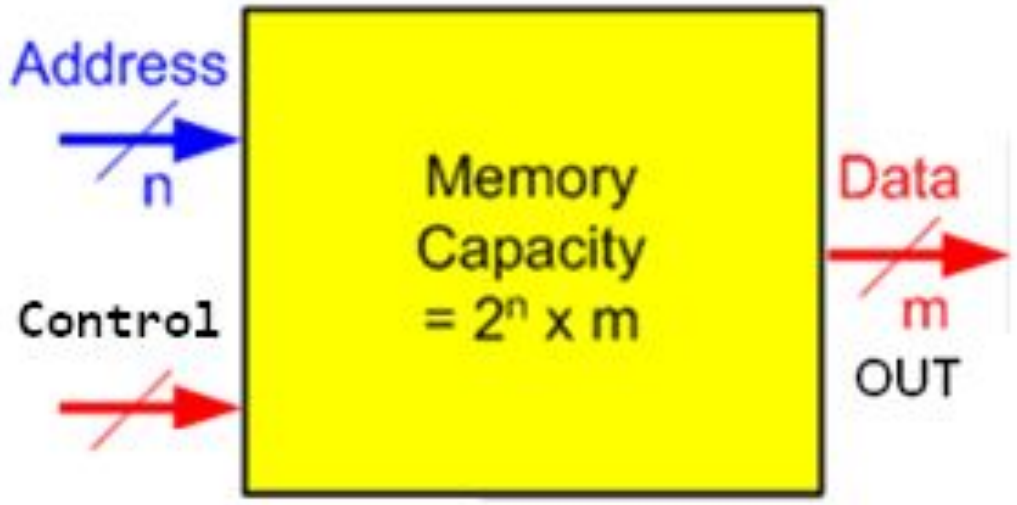
UNIT 3 – COMBINATIONAL LOGIC CIRCUITS

READ ONLY MEMORY

-S.Vaishali

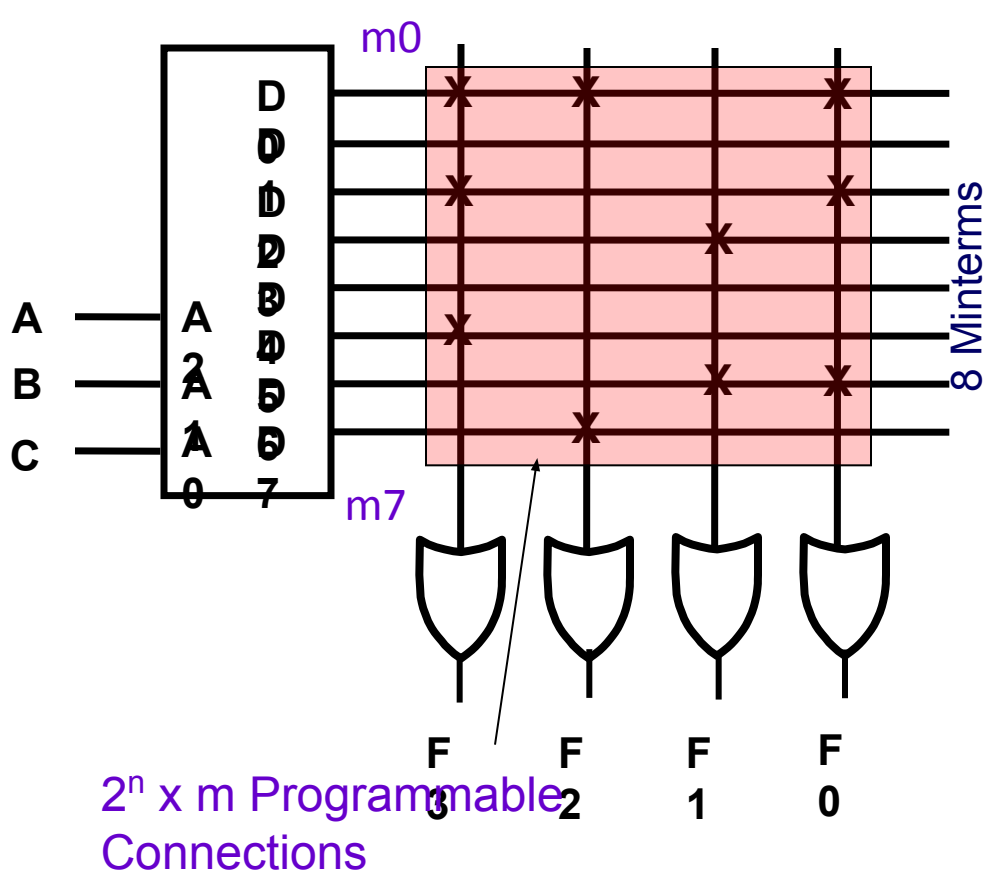
Read Only Memory (ROM)

- Data stored in a ROM is non-volatile, i.e. this data is permanently stored until erased or changed through re-programming (if applicable)
- The ROM has n input lines for the address and m output data lines
- Total memory capacity of a ROM is 2^n x m bits
- ROMs do not have input lines as a write operation does not exist in them
- Programmable ROMs receive data to be programmed on the output lines
- Generally, system-level programs that need to be accessed frequently and at power up access are stored in the computer's ROM, e.g. the BIOS firmware



Read Only Memory (ROM)
Programmable sum of (fixed) minterms

- Example: 8 X 4 PROM (n = 3 input lines, m = 4 output lines)
- The fixed "AND" array is a "decoder" with 3 inputs and 8 outputs implementing minterms
- The programmable "OR" array uses a single line to represent all inputs to an OR gate. An "X" in the array corresponds to attaching the minterm to the OR
- Read Example: For input (A_2,A_1,A_0) = 010, output is (F_3,F_2,F_1,F_0) = 1001.
- What are functions F_3, F_2 , F_1 and F_0 in terms of (A_2, A_1, A_0)?



Read Only Memory (ROM): n i/ps to m o/ps
2^n locations x m bits each

- Read Only Memories (ROM) have:
  - n input (address) lines □ 2^n locations □ 2^n decoded minterms
  - m output lines (word width)
- Fixed array of 2^n AND gates implementing all the N-literal minterms.
- Programmable OR Array with m outputs lines to form up to m expressions, each being a sum of selected minterm.
- The program for a PROM is simply the multiple-output truth table to be implemented
  - If a 1 entry, a connection is made to the corresponding minterm for the corresponding output
  - If a 0, no connection is made
- Can be viewed as a *memory* with the inputs as *addresses* of *data* (output values), hence ROM or PROM names!
  Device on previous slide is an 8 x 4 memory (8 locations, each 4 bits)
- Truth table is a listing of the memory contents

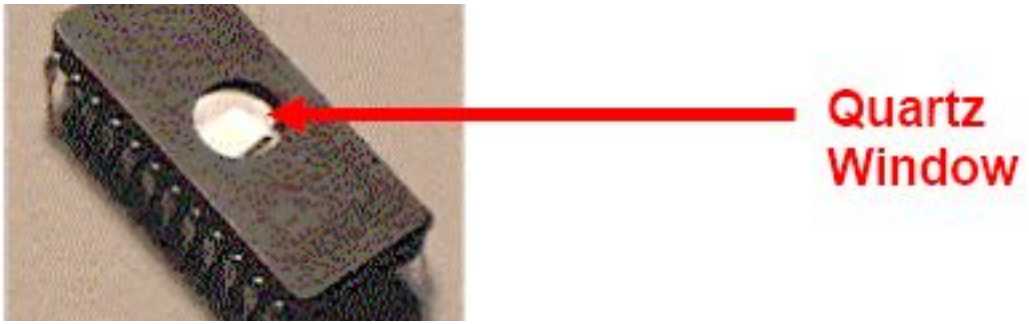
Read Only Memory (ROM) Advantages/Limitations

Advantages:	•Can implement <u>any</u> function (all the minterms are available) •Program is derived directly from the truth table (uses the canonical form)
Disadvantages:	•Becomes complex for a large number of inputs n (# of ANDs = 2^n, each n-input wide) •Does not support multi-level circuits (no outputs brought back as inputs)

Types of ROM Devices

- Simply ROM: Programmed only once and by the manufacturer (in factory), based on the client's truth table
- PROM: A ROM programmable only once by the user (in the field). The user blows fuses to remove unwanted connections. This process is irreversible and hence device is programmed only once
- EPROM: Erasable, Programmable ROMs. Can have their data erased using Ultraviolet light and reprogrammed. The user can then reprogram the ROM many times using special programmers Off-situ.

Off-situ: Remove from computer to erase/program



Types of ROM, Contd.

- EEPROMs: Electrically Erasable Programmable ROMs. Have memory cells that can be erased and reprogrammed by exposure to electrical signals.

  Erasure/Programming is now much easier and in-situ. The processor can now "write" into the EEPROM.

- Flash memory devices:
  - Memory cells are erased in blocks not one-by-one as in EEPROMs Shorter life but faster operation

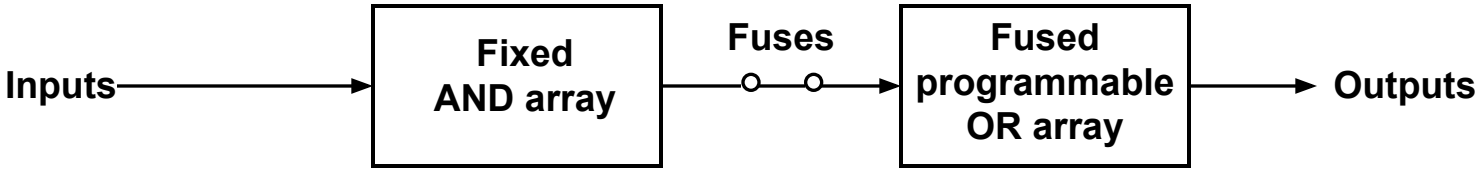
UNIT – IV

Programmable Logic devices, Memory and Logic Families

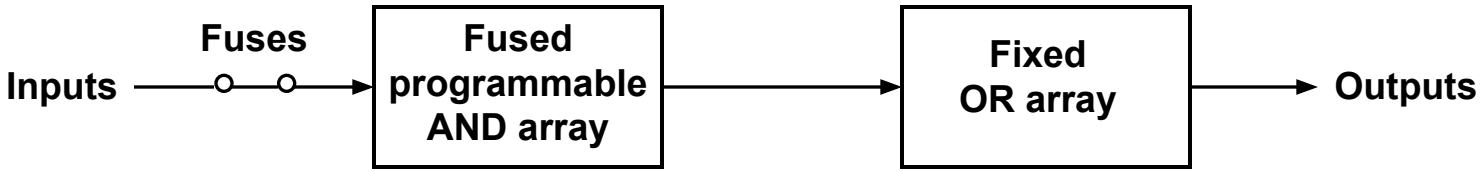
Programmable Logic Devices (PLD)

- Programmable Logic Devices (PLDs) are IC chips with internal logic gates connected by electronic fuses.
- These fuses can be ‘blown’ (by programming) to obtain different circuit configurations.
- Semi-customized chips that give high packing density at reasonable cost.
- Three classes of PLDs are :
  - Programmable Logic Array (PLA)
  - Programmable Read Only Memory (PROM)
  - Programmable Array Logic (PAL)

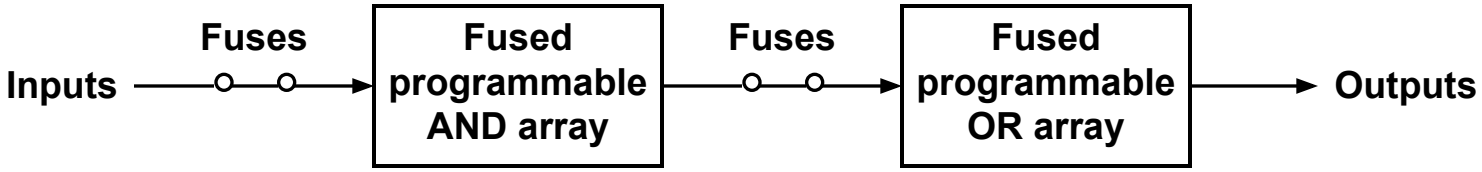
Programmable Logic Devices



Programmable Read Only Memory (PROM)



Programmable Array Logic (PAL)



Programmable Logic Array (PLA)

Programmable Logic Array (PLA)

- Combination of a programmable AND array followed by a programmable OR array.
- Example: Design a PLA to realise the following three logic functions and show the internal connections.

$f_1(A,B,C,D,E) = A'.B'.D' + B'.C.D' + A'.B.C.D.E'$

$f_2(A,B,C,D,E) = A'.B.E + B'.C.D'.E$

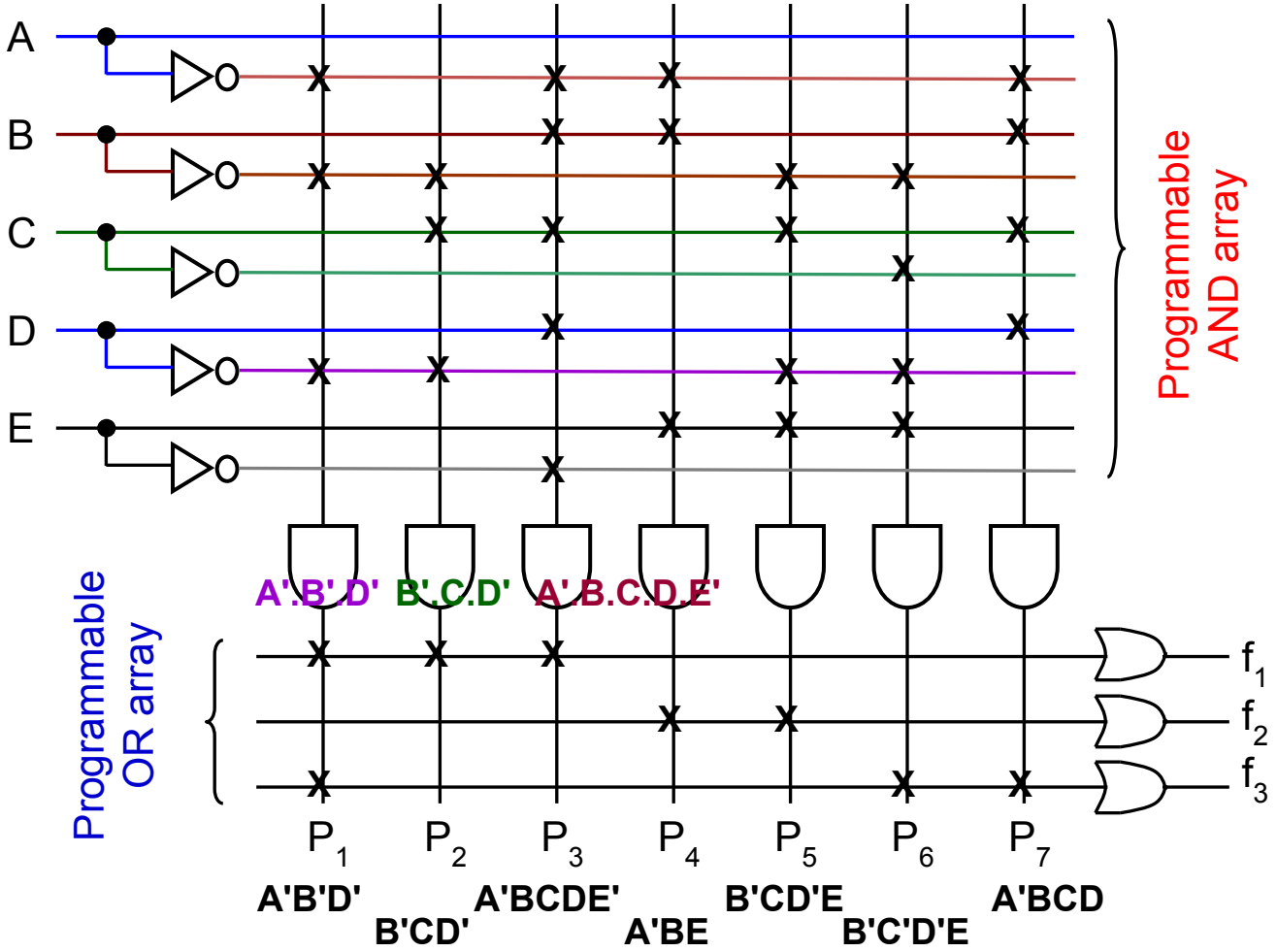
$f_3(A,B,C,D,E) = A'.B'.D' + B'.C'.D'.E + A'.B.C.D$

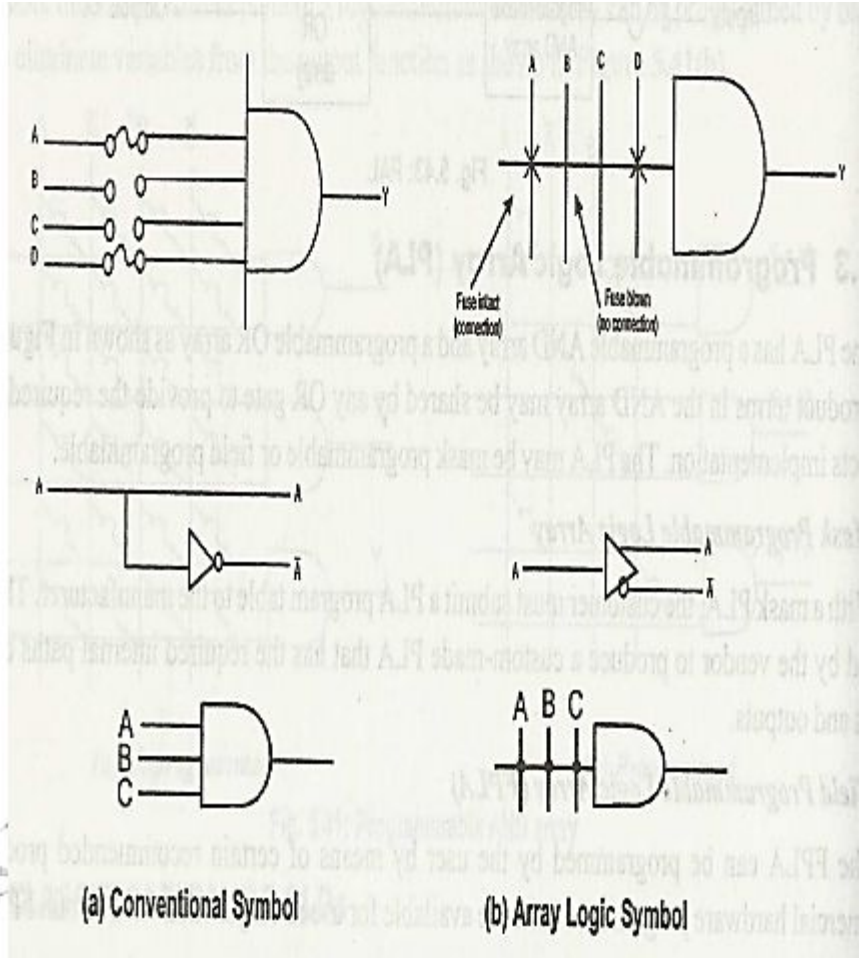
Realising Logic Functions with PLAs

$f_1(A,B,C,D,E) = A'.B'.D' + B'.C.D' + A'.B.C.D.E'$

$f_2(A,B,C,D,E) = A'.B.E + B'.C.D'.E$

$f_3(A,B,C,D,E) = A'.B'.D' + B'.C'.D'.E + A'.B.C.D$





(a) Conventional Symbol

(b) Array Logic Symbol

Implement the following functions using PLA.

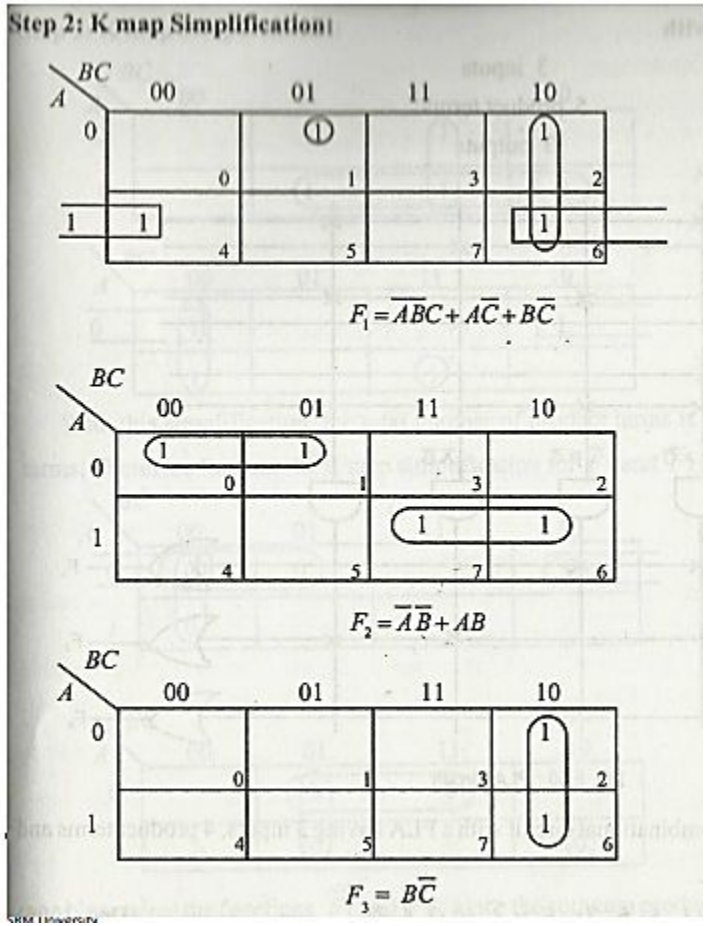
$$F_1 = \Sigma m\,(1, 2, 4, 6)\;;\; F_2 = \Sigma m\,(0, 1, 6, 7)$$

$$F_3 = \Sigma m\,(2, 6)$$

Solution: Step 1: Truth Table for given functions:

A	B	C	F_1	F_2	F_3
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	1	0

Step 2: K map Simplification:

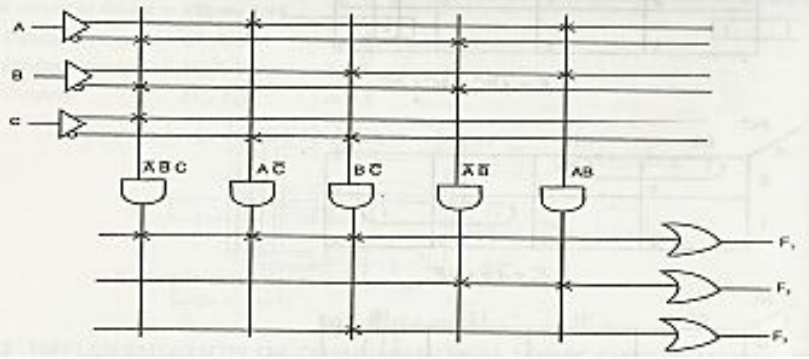


Step 3: PLA Program Table:

Product Term		Inputs A	B	C	Outputs F_1	F_2	F_3	
$\overline{A}\overline{B}C$	1	0	0	1	1	–	–	
$A\overline{C}$	2	1	–	0	1	–	–	
$B\overline{C}$	3	–	1	0	1		1	
$\overline{A}\overline{B}$	4	0	0	–	–	1	–	
AB	5	1	1	–	–	1	–	
					T	T	T	T/C

Step 4: Draw the PLA circuit with

3 inputs
5 product terms
3 outputs



Implement the combinational circuit with PLA having 3 inputs,4 product terms and 2 output for the functions:

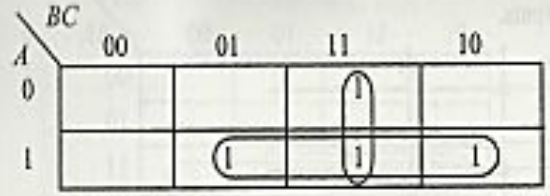
$$F_1 = \Sigma\ (3, 5, 6, 7),\quad F_2 = \Sigma\ (0, 2, 4, 7)$$

Solution :

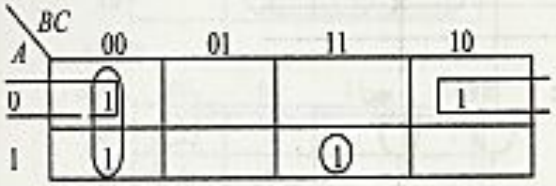
Step 1: Truth table for Boolean Functions:

A	B	C	F_1	F_2
0	0	0	0	1
0	0	1	0	0
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

Step 2: K map Simplification:

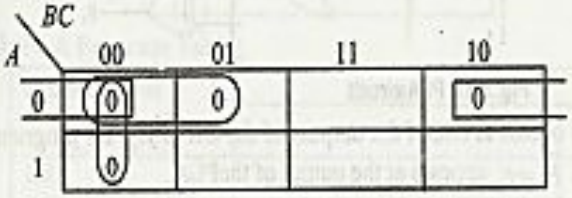


$F_1 = AC + AB + BC$

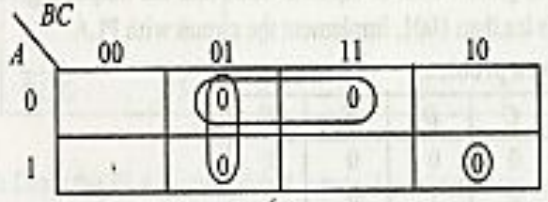


$\overline{F}_2 = \overline{B}\,\overline{C} + \overline{A}\,\overline{C} + ABC$

With this simplification, the total number of product terms is 6. But we require only 4 product terms. Therefore find out the K map simplification for $\overline{F}_1$ and $\overline{F}_2$.



$\overline{F}_1 = \overline{B}\,\overline{C} + \overline{A}\,\overline{C} + \overline{A}\,\overline{B}$



$\overline{F}_2 = \overline{A}C + \overline{B}C + AB\overline{C}$

Now select the functions $\overline{F}_1$ and $\overline{F}_2$, since the common product terms are $(\overline{B}C, \overline{A}C, \overline{A}\overline{B}, ABC)$

Step 3: PLA Program Table:

Product Term		Inputs A	Inputs B	Inputs C	Outputs F_1	Outputs F_2	
$\overline{B}\,\overline{C}$	1	–	0	0	1	1	
$\overline{A}\,\overline{C}$	2	0	–	0	1	1	
$\overline{A}\,\overline{B}$	3	0	0	–	1	–	
ABC	4	1	1	1	–	1	
					C	T	T/C

$$F_1=\left(\overline{\overline{B}\,\overline{C}+\overline{A}\,\overline{C}+\overline{A}\,\overline{B}}\right) \;;\; F_2=\overline{B}\,\overline{C}+\overline{A}\,\overline{C}+ABC$$

Step 4: Draw the PLA circuit with

3 inputs, 4 product terms and 2 outputs.

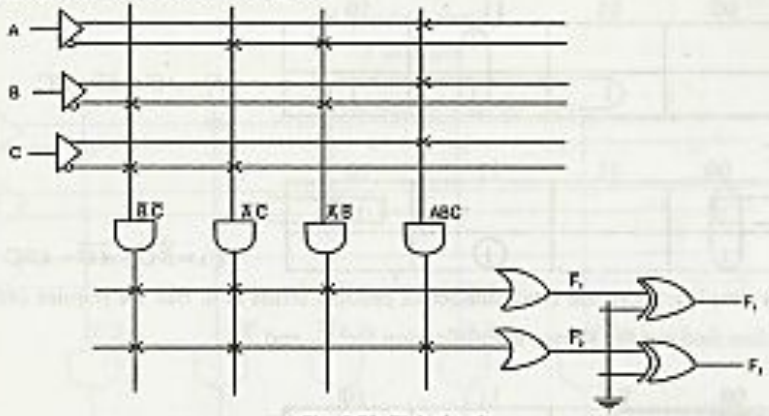


Fig. 5.51: PLA circuit

It should be noted that $\overline{F}_1$ really occurs at one of the outputs of the OR-array. By programming the corresponding EX-OR gate fuse, $\overline{\overline{F}_1}=F_1$ appears at the output of thePLA.

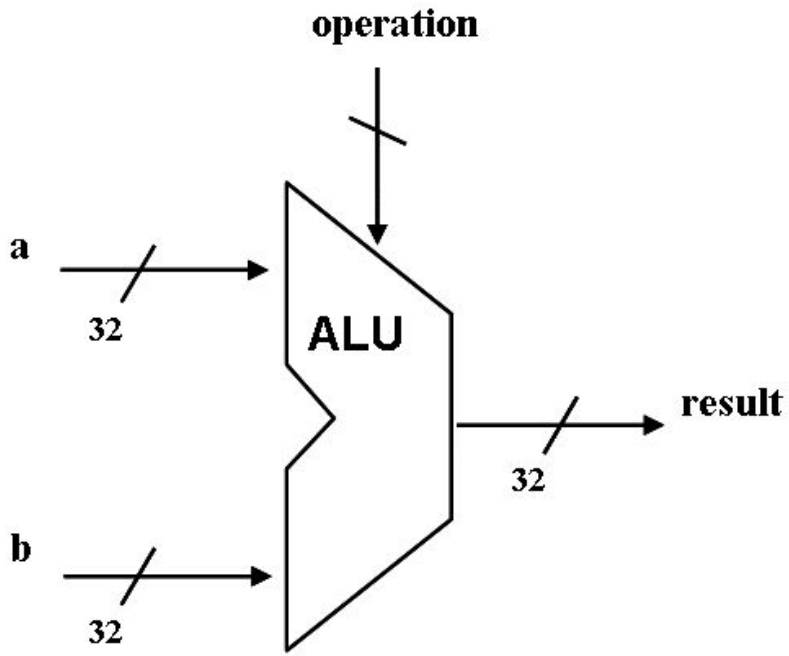
ANALOG AND DIGITAL ELECTRONICS

UNIT 3 – COMBINATIONAL LOGIC CIRCUITS

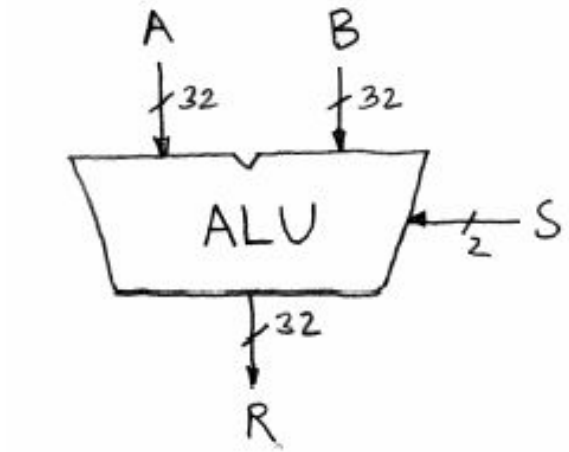
ARITHMETIC LOGIC UNIT

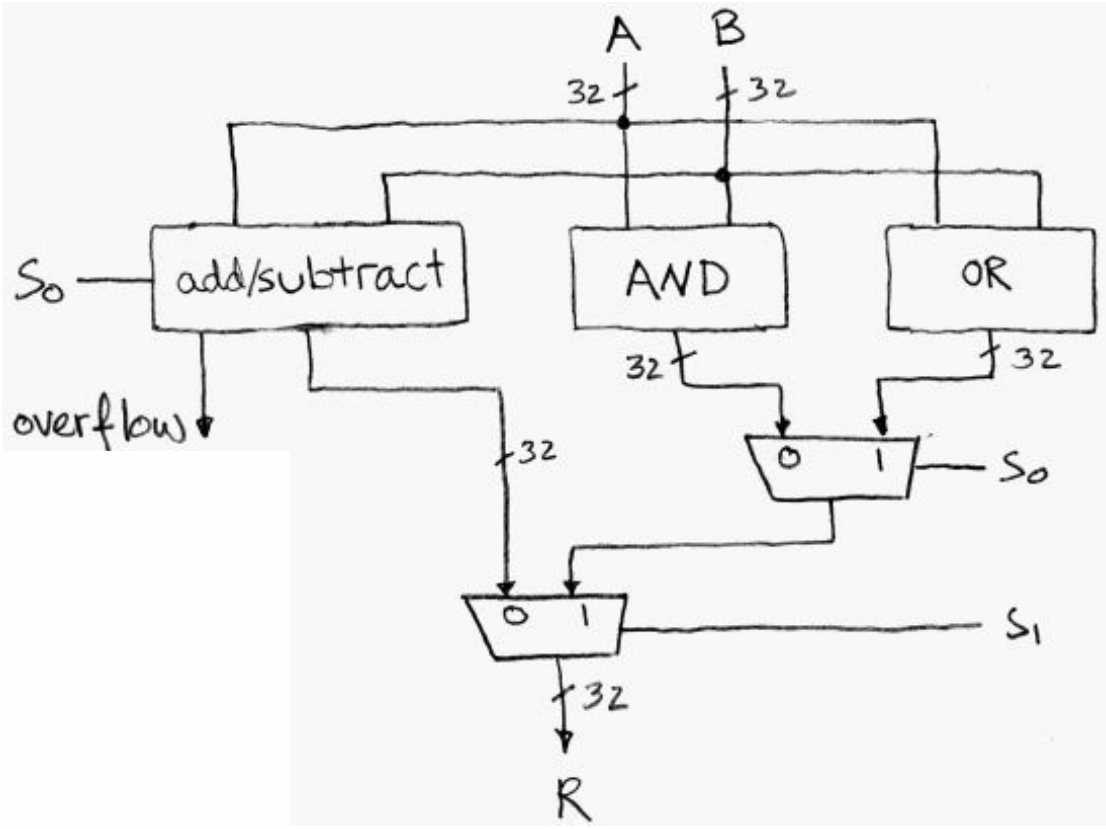
-S.Vaishali

- An **arithmetic logic unit** (*ALU*)
  - Performs arithmetic and logic operations
  - A fundamental building block of the Central Processing Unit (CPU) of a computer
  - Even the simplest microprocessors contain one for purposes such as maintaining timers
  - A combinational logic circuit



Simple ALU



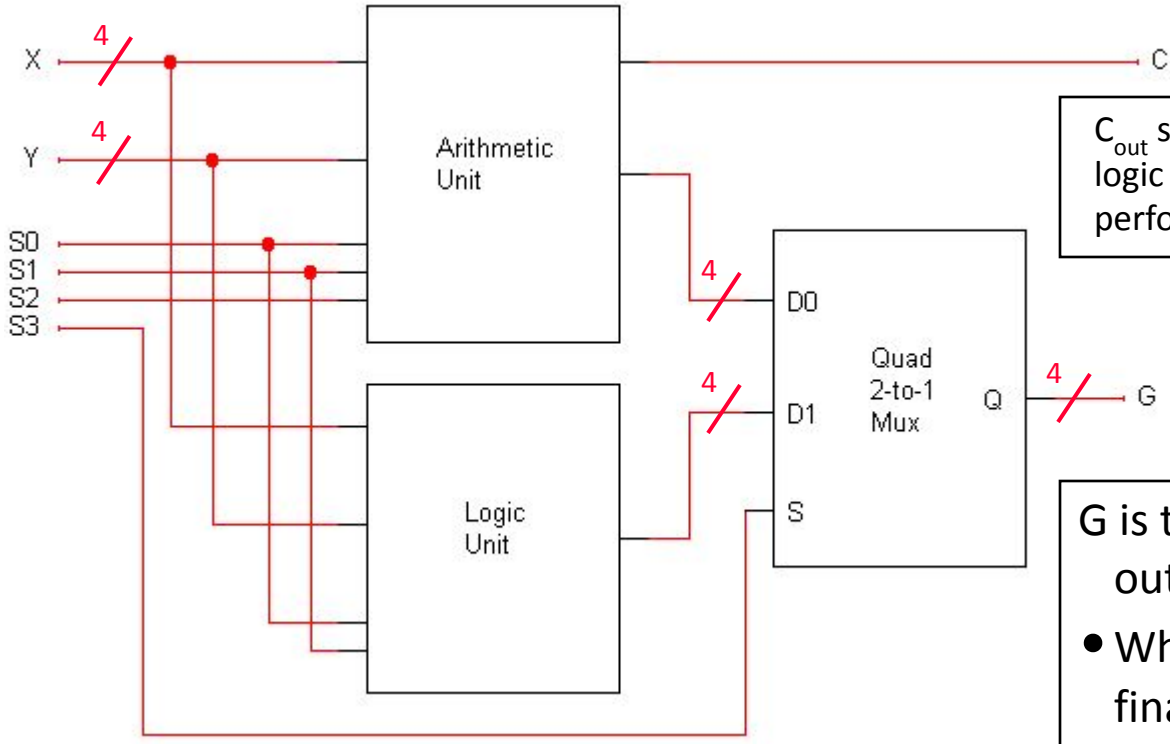


when S=00, R=A+B
when S=01, R=A-B
when S=10, R=A AND B
when S=11, R=A OR B

The S input is controlled by the processor based on the op code

A complete ALU circuit

The / and 4 on a line indicate that it's actually *four* lines.



C_{out} should be ignored when logic operations are performed (when S3=1).

The arithmetic and logic units share the select inputs S1 and S0, but only the arithmetic unit uses S2.

G is the final ALU output.

- When S3 = 0, the final output comes from the arithmetic unit.
- When S3 = 1, the output comes from the logic unit.

Unit - 3

Hardware Description Language

Hardware Description Language - Introduction

- HDL is a language that describes the hardware of digital systems in a textual form.
- It resembles a programming language, but is specifically oriented to describing hardware structures and behaviors.
- The main difference with the traditional programming languages is HDL's representation of extensive parallel operations whereas traditional ones represents mostly serial operations.
- The most common use of a HDL is to provide an alternative to schematics.

HDL – Introduction

- When a language is used for the above purpose (i.e. to provide an alternative to schematics), it is referred to as a *structural description* in which the language describes an interconnection of components.
- Such a structural description can be used as input to logic simulation just as a schematic is used.
- Models for each of the primitive components are required.
- If an HDL is used, then these models can also be written in the HDL providing a more uniform, portable representation for simulation input.

HDL – Introduction

- HDL can be used to represent logic diagrams, Boolean expressions, and other more complex digital circuits.
- Thus, in top down design, a very high-level description of a entire system can be precisely specified using an HDL.
- This high-level description can then be refined and partitioned into lower-level descriptions as a part of the design process.

HDL – Introduction

- As a documentation language, HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers.
- The language content can be stored and retrieved easily and processed by computer software in an efficient manner.
- There are two applications of HDL processing: ***Simulation*** and ***Synthesis***

Logic Simulation

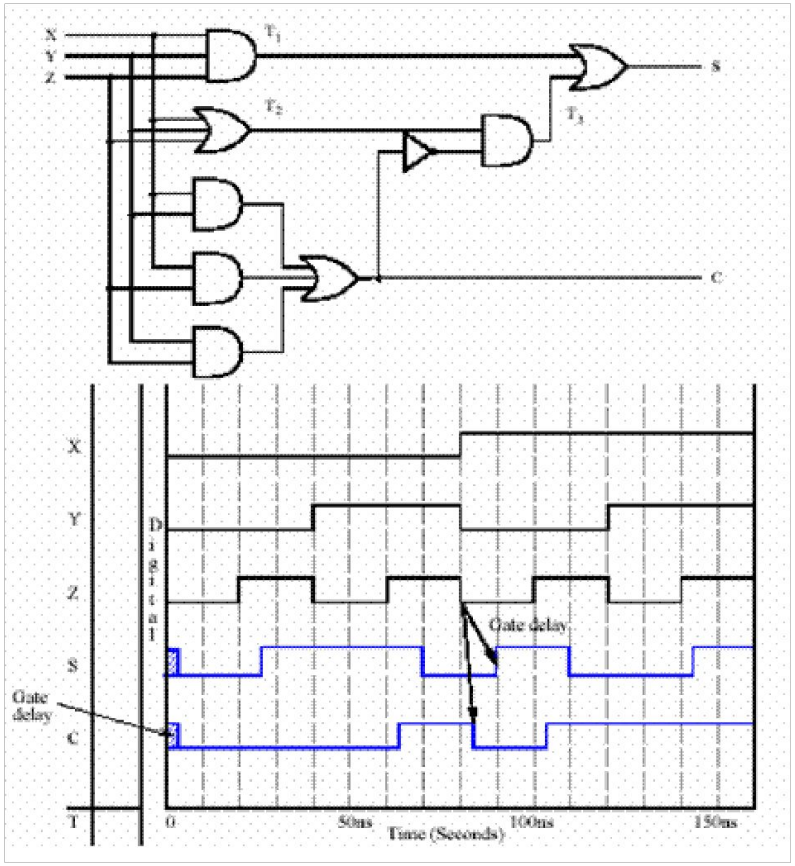
- A simulator interprets the HDL description and produces a readable output, such as a timing diagram, that predicts how the hardware will behave before its is actually fabricated.
- Simulation allows the detection of functional errors in a design without having to physically create the circuit.

Logic Simulation

- The stimulus that tests the functionality of the design is called a test bench.
- To simulate a digital system
  - Design is first described in HDL
  - Verified by simulating the design and checking it with a test bench which is also written in HDL.

Logic Simulation

- Logic simulation is a fast, accurate method of analyzing a circuit to see its waveforms



Types of HDL

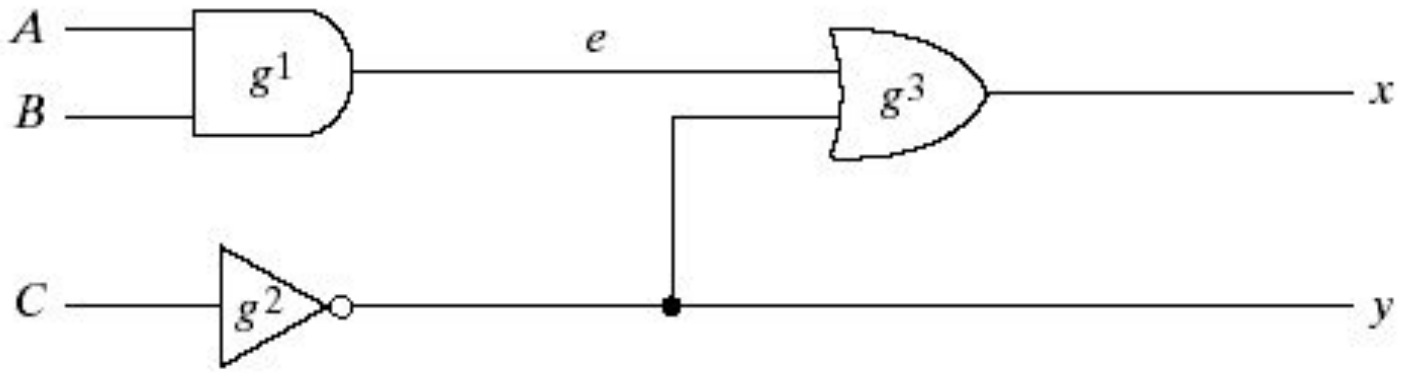
- There are two standard HDL's that are supported by IEEE.
  - **VHDL** (*Very-High-Speed Integrated Circuits Hardware Description Language*) - Sometimes referred to as VHSIC HDL, this was developed from an initiative by US. Dept. of Defense.
  - **Verilog HDL** – developed by Cadence Data systems and later transferred to a consortium called *Open Verilog International* (OVI).

Verilog

- Verilog HDL has a syntax that describes precisely the legal constructs that can be used in the language.
- It uses about 100 keywords pre-defined, lowercase, identifiers that define the language constructs.
- Example of keywords: *module, endmodule, input, output wire, and, or, not* , etc.,
- Any text between two slashes (//) and the end of line is interpreted as a comment.
- Blank spaces are ignored and names are case sensitive.

Verilog Module

- A *module* is the building block in Verilog.
- It is declared by the keyword *module* and is always terminated by the keyword *endmodule*.
- Each statement is terminated with a semicolon, but there is no semi-colon after *endmodule*.



Verilog Module for simple circuit

HDL Example

```
module smpl_circuit(A,B,C,x,y);
  input  A,B,C;
  output x,y;
  wire e;
  and g1(e,A,B);
  not g2(y,C);
  or  g3(x,e,y);
endmodule
```

Verilog – Gate Delays

- Sometimes it is necessary to specify the amount of delay from the input to the output of gates.
- In Verilog, the delay is specified in terms of time units and the symbol **#**.
- The association of a time unit with physical time is made using ***timescale*** compiler directive.
- Compiler directive starts with the “backquote (`)” symbol.

```
`timescale 1ns/100ps
```

- The first number specifies the *unit of measurement* for time delays.
- The second number specifies the *precision* for which the delays are rounded off, in this case to 0.1ns.

Verilog Module with Gate Delays

```
//Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
    input  A,B,C;
    output x,y;
    wire   e;
    and #(30) g1(e,A,B);
    or  #(20) g3(x,e,y);
    not #(10) g2(y,C);
endmodule
```

Test Bench

- In order to simulate a circuit with HDL, it is necessary to apply inputs to the circuit for the simulator to generate an output response.
- An HDL description that provides the stimulus to a design is called a **test bench**.
- The ***initial*** statement specifies inputs between the keyword ***begin*** and ***end***.
- Initially ABC=000 (A,B and C are each set to 1'b0 (one binary digit with a value 0).
- **$finish** is a *system task*.

Test Bench for Simple Circuit

```
module circuit_with_delay
(A,B,C,x,y);
  input A,B,C;
  output x,y;
  wire e;
and #(30) g1(e,A,B);
or  #(20) g3(x,e,y);
not #(10) g2(y,C);
endmodule
```

```
//Stimulus for simple circuit
module stimcrct;
reg A,B,C;
wire x,y;
circuit_with_delay cwd(A,B,C,x,y);
initial
  begin
     A = 1'b0; B = 1'b0; C = 1'b0;
   #100
     A = 1'b1; B = 1'b1; C = 1'b1;
   #100  $finish;
  end
endmodule
```

Verilog Module for Boolean Expressions

Bitwise operators

- Bitwise NOT : ~
- Bitwise AND: &
- Bitwise OR: |
- Bitwise XOR: ^
- Bitwise XNOR: ~^ or ^~

Verilog Module for Boolean Expressions

Boolean Expressions:

- These are specified in Verilog HDL with a continuous assignment statement consisting of the keyword *assign* followed by a Boolean Expression.
- The earlier circuit can be specified using the statement:

```
assign x = (A&B)|~C)
```

E.g. x = A + BC + B'D

y = B'C + BC'D'

Verilog Module for Boolean Expressions

```
//Circuit specified with Boolean equations
module circuit_bln (x,y,A,B,C,D);
    input A,B,C,D;
    output x,y;
    assign x = A | (B & C) | (~B & C);
    assign y = (~B & C) | (B & ~C & ~D);
endmodule
```

User Defined Primitives

User Defined Primitives (UDP):

- The logic gates used in HDL descriptions with keywords ***and, or***,etc., are defined by the system and are referred to as ***system primitives***.
- The user can create additional primitives by defining them in tabular form.
- These type of circuits are referred to as ***user-defined primitives***.

User Defined Primitives

UDP features ….

- UDP's do not use the keyword module. Instead they are declared with the keyword ***primitive***.
- There can be ***only one output*** and it must be listed first in the port list and declared with an *output* keyword.
- There can be any number of inputs. The order in which they are listed in the ***input*** declaration must conform to the order in which they are given values in the table that follows.
- The truth table is enclosed within the keywords ***table*** and ***endtable***.
- The values of the inputs are listed with a colon (:). The output is always the last entry in a row followed by a semicolon (;).
- It ends with the keyword ***endprimitive***.

Verilog Module for User Defined Primitives

```
//User defined primitive(UDP)
primitive crctp (x,A,B,C);
    output x;
    input  A,B,C;
//Truth table for x(A,B,C) = Minterms (0,2,4,6,7)
    table
//      A   B   C  :  x  (Note that this is only a
   comment)
        0   0   0  :  1;
        0   0   1  :  0;
        0   1   0  :  1;
        0   1   1  :  0;
        1   0   0  :  1;
        1   0   1  :  0;
        1   1   0  :  1;
        1   1   1  :  1;
    endtable
endprimitive
```

```
// Instantiate primitive
module declare_crctp;
    reg    x,y,z;
    wire   w;
    crctp (w,x,y,z);
endmodule
```

Types of Modeling

- A module can be described in any one (or a combination) of the following modeling techniques.
  - **Gate-level modeling** using instantiation of primitive gates and user defined modules.
    - This describes the circuit by specifying the gates and how they are connected with each other.
  - **Dataflow modeling** using continuous assignment statements with the keyword **`assign`.**
    - This is mostly used for describing combinational circuits.
  - **Behavioral modeling** using procedural assignment statements with keyword **`always`**.
    - This is used to describe digital systems at a higher level of abstraction.

Gate-Level Modeling

- Here a circuit is specified by its **logic gates** and their **interconnections**.
- It provides a textual description of a schematic diagram.
- Verilog recognizes 12 basic gates as predefined primitives.
  - 4 primitive gates of 3-state type.
  - Other 8 are: **`and`, `nand`, `or`, `nor`, `xor`, `xnor`, `not`, `buf`**
- When the gates are simulated, the system assigns a four-valued logic set to each gate – *0,1,unknown (x) and high impedance (z)*

Gate-level modeling

- When a primitive gate is incorporated into a module, we say it is *instantiated* in the module.
- In general, component instantiations are statements that reference lower-level components in the design, essentially creating unique copies (or *instances*) of those components in the higher-level module.
- Thus, a module that uses a gate in its description is said to *instantiate* the gate.

Gate-level Modeling

- Modeling with vector data (multiple bit widths):
  - A vector is specified within square brackets and two numbers separated with a colon.

  e.g. **`output`**`[0:3] D;` - This declares an output vector D with 4 bits, 0 through 3.

  `wire``[7:0] SUM;` – This declares a wire vector SUM with 8 bits numbered 7 through 0.

  The first number listed is the most significant bit of the vector.

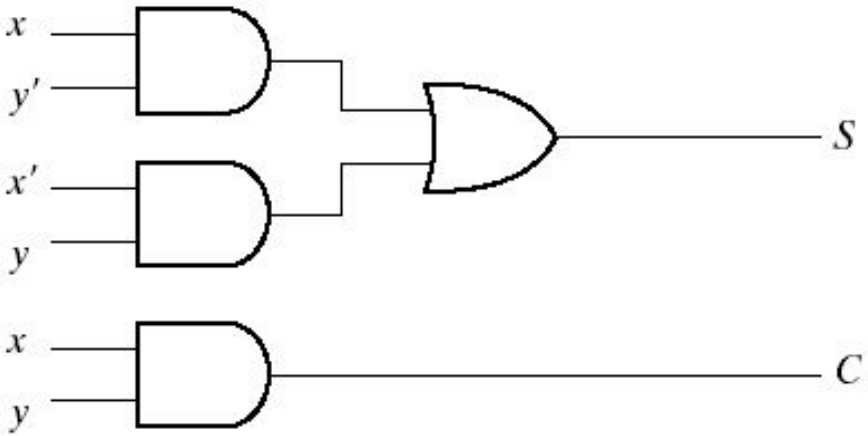
Gate-level Modeling

- Two or more modules can be combined to build a hierarchical description of a design.
- There are two basic types of design methodologies.
  - **Top down**: In top-down design, the top level block is defined and then sub-blocks necessary to build the top level block are identified.
  - **Bottom up**: Here the building blocks are first identified and then combine to build the top level block.
- In a top-down design, a 4-bit binary adder is defined as top-level block with 4 full adder blocks. Then we describe two half-adders that are required to create the full adder.
- In a bottom-up design, the half-adder is defined, then the full adder is constructed and the 4-bit adder is built from the full adders.

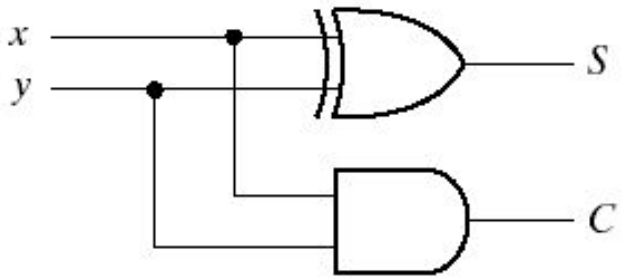
Gate-level Modeling

- A bottom-up hierarchical description of a 4-bit adder is described in Verilog as
  - Half adder: defined by instantiating primitive gates.
  - Then define the full adder by instantiating two half-adders.
  - Finally the third module describes 4-bit adder by instantiating 4 full adders.
- ***Note:*** In Verilog, one module definition cannot be placed within another module description.

4-bit Half Adder



(a) $S = xy' + x'y$
$C = xy$



(b) $S = x \oplus y$
$C = xy$

4-bit Full Adder

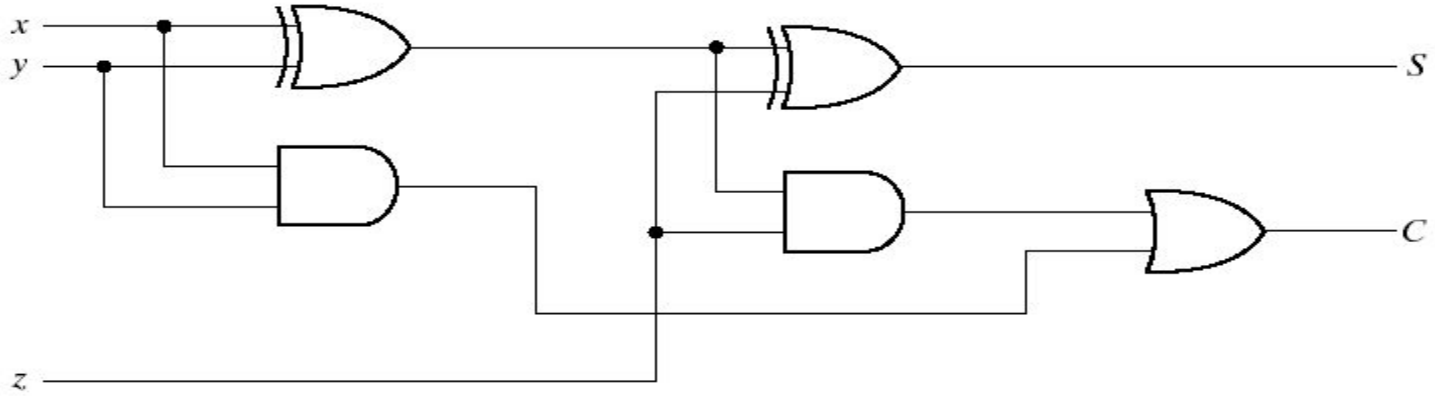


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

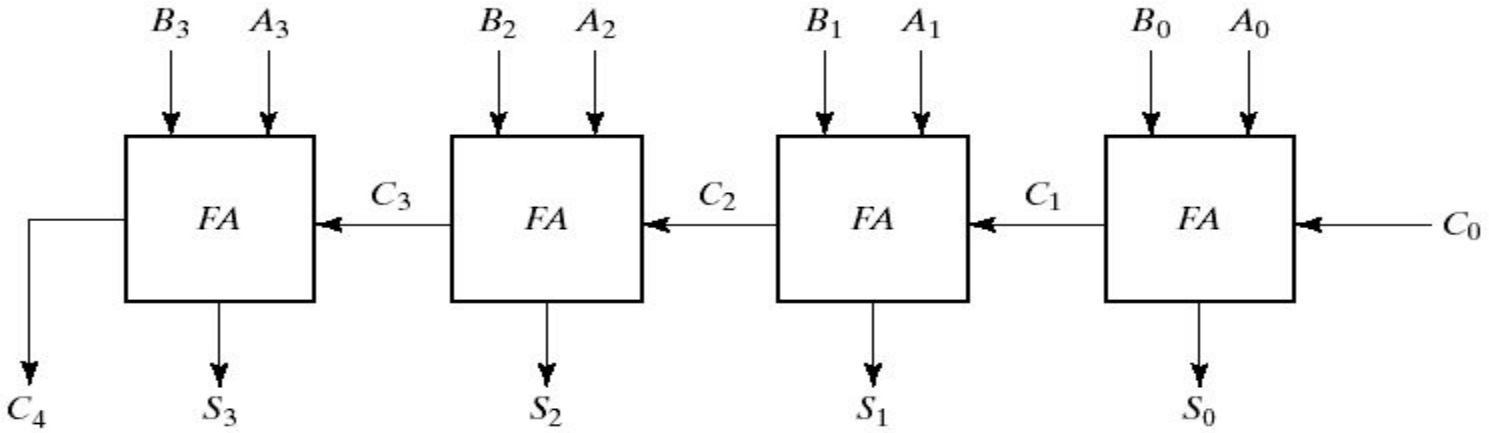


Fig. 4-9 4-Bit Adder

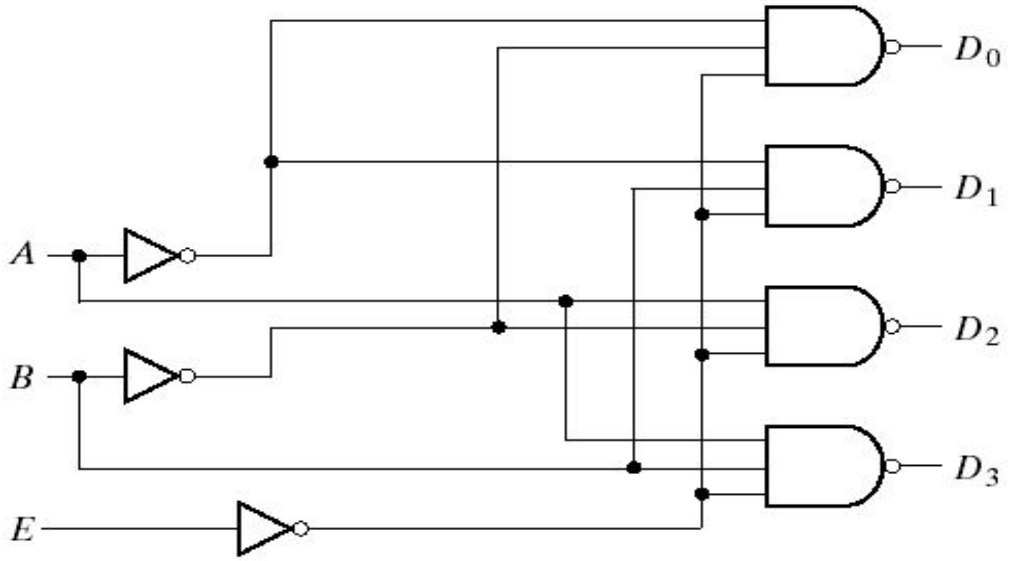
4-bit Full Adder

```
//Gate-level hierarchical description of 4-bit adder
module halfadder (S,C,x,y);
    input x,y;
    output S,C;
    //Instantiate primitive gates
    xor (S,x,y);
    and (C,x,y);
endmodule

module fulladder (S,C,x,y,z);
    input  x,y,z;
    output S,C;
wire   S1,D1,D2; //Outputs of first XOR and two AND gates
    //Instantiate the half adders
halfadder HA1(S1,D1,x,y), HA2(S,D2,S1,z);
    or g1(C,D2,D1);
endmodule
```

4-bit Full Adder

```
module _4bit_adder (S,C4,A,B,C0);
          input [3:0] A,B;
             input C0;
          output [3:0] S;
             output C4;
wire C1,C2,C3;  //Intermediate carries

     //Instantiate the full adder
fulladder  FA0 (S[0],C1,A[0],B[0],C0),
           FA1 (S[1],C2,A[1],B[1],C1),
           FA2 (S[2],C3,A[2],B[2],C2),
           FA3 (S[3],C4,A[3],B[3],C3);
             endmodule
```

2 to 4 Decoder



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

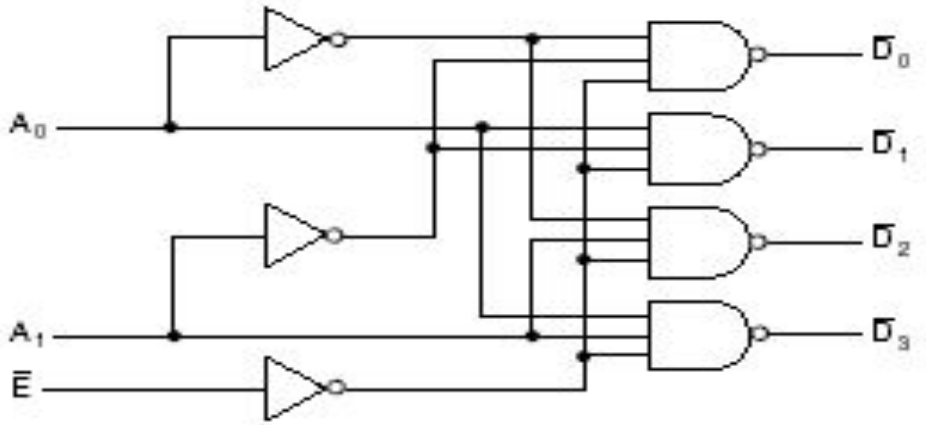
(b) Truth table

Fig. 4-19 2-to-4-Line Decoder with Enable Input

2 to 4 Decoder

```
//Gate-level description of a 2-to-4-line decoder
module decoder_gl (A,B,E,D);
        input   A,B,E;
        output[0:3]D;
    wire    Anot,Bnot,Enot;
          not
        n1 (Anot,A),
        n2 (Bnot,B),
        n3 (Enot,E);
          nand
    n4 (D[0],Anot,Bnot,Enot),
      n5 (D[1],Anot,B,Enot),
      n6 (D[2],A,Bnot,Enot),
       n7 (D[3],A,B,Enot);
          endmodule
```

2-to-4 Line Decoder



(a) Logic diagram

$\overline{E}$	A_1	A_0	$\overline{D}_0$	$\overline{D}_1$	$\overline{D}_2$	$\overline{D}_3$
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	X	X	1	1	1	1

(b) Truth table

$$\overline{D}_0 = \overline{E\ \overline{A}_1\ \overline{A}_0}$$
$$\overline{D}_1 = \overline{E\ \overline{A}_1\ A_0}$$
$$\overline{D}_2 = \overline{E\ A_1\ \overline{A}_0}$$
$$\overline{D}_3 = \overline{E\ A_1\ A_0}$$

(c) Logic Equations

Fig.3-14 A 2–to–4-Line Decoder

2-to-4 Line Decoder

```
//2 to 4 line decoder
module decoder_2_to_4_st_v(E_n, A0, A1, D0_n, D1_n,
   D2_n, D3_n);
   input E_n, A0, A1;
   output D0_n, D1_n, D2_n, D3_n;

   wire   A0_n, A1_n, E;
   not    g0(A0_n, A0), g1(A1_n, A1), g2(E,E_n);
   nand   g3(D0_n,A0_n,A1_n,E), g4(D1_n,A0,A1_n,E),
          g5(D2_n,A0_n,A1,E), g6(D3_n,A0,A1,E);
endmodule
```

Dataflow Modeling

- Dataflow modeling uses a number of operators that act on operands to produce desired results.
- Verilog HDL provides about 30 operator types.
- Dataflow modeling uses continuous assignments and the keyword **`assign`**.
- A continuous assignment is a statement that assigns a value to a net.
- The value assigned to the net is specified by an expression that uses operands and operators.

Dataflow Modeling - 2-to-4-line decoder

```
//Dataflow description of a 2-to-4-line decoder
module decoder_df (A,B,E,D);
    input A,B,E;
    output [0:3] D;
  assign D[0] = ~(~A & ~B & ~E),
         D[1] = ~(~A & B & ~E),
         D[2] = ~(A & ~B & ~E),
         D[3] = ~(A & B & ~E);
endmodule
```

A 2-to-1 line multiplexer with data inputs A and B, select input S, and output Y is described with the continuous assignment

```
assign Y = (A & S) | (B & ~S)
```

Dataflow Modeling - 4-bit adder and 4-bit comparator

```
//Dataflow description of 4-bit adder
module binary_adder (A,B,Cin,SUM,Cout);
    input [3:0] A,B;
    input Cin;
    output [3:0] SUM;
    output Cout;
    assign {Cout,SUM} = A + B + Cin;
endmodule
```

```
//Dataflow description of a 4-bit comparator.
module magcomp (A,B,ALTB,AGTB,AEQB);
    input [3:0] A,B;
    output ALTB,AGTB,AEQB;
    assign ALTB = (A < B),
           AGTB = (A > B),
           AEQB = (A == B);
endmodule
```

Dataflow Modeling - 2-to-1-line mux

- Dataflow Modeling provides the means of describing combinational circuits by their function rather than by their gate structure.
- **Conditional operator (?:)**

  condition ? true-expression : false-expression;
- A 2-to-1 line multiplexer

```
assign OUT = select ? A : B;
```

```
//Dataflow description of 2-to-1-line mux
    module mux2x1_df (A,B,select,OUT);
        input A,B,select;
        output OUT;
    assign OUT = select ? A : B;
    endmodule
```

HDL Behavioral Modeling

Behavioral Modeling

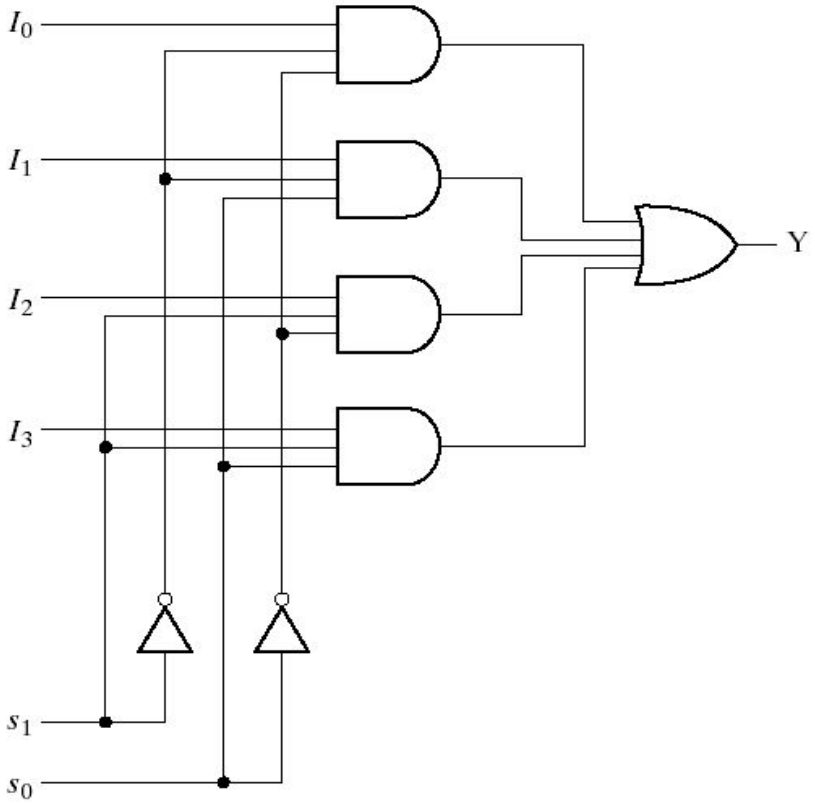
- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.
- Behavioral descriptions use the keyword **`always`** followed by a list of procedural assignment statements.
- The target output of procedural assignment statements must be of the **`reg`** data type.
- A **`reg`** data type retains its value until a new value is assigned.

Behavioral Modeling 2-to-1-line mux

- The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variable listed after the @ symbol. (Note that there is no ";" at the end of **always** statement)

```
//Behavioral description of 2-to-1-line multiplexer
module mux2x1_bh(A,B,select,OUT);
    input A,B,select;
    output OUT;
    reg OUT;
always @(select or A or B)
    if (select == 1) OUT = A;
        else OUT = B;
endmodule
```

Behavioral Modeling - 4-to-1 line multiplexer



(a) Logic diagram

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

(b) Function table

4-to-1 line multiplexer

Behavioral Modeling 4-to-1 line multiplexer

```
//Behavioral description of 4-to-1 line mux

module mux4x1_bh (i0,i1,i2,i3,select,y);
        input i0,i1,i2,i3;
        input [1:0] select;
        output y;
        reg y;
always @(i0 or i1 or i2 or i3 or select)
        case (select)
          2'b00: y = i0;
          2'b01: y = i1;
          2'b10: y = i2;
          2'b11: y = i3;
        endcase
endmodule
```

Behavioral Modeling 4-to-1 line multiplexer

- In 4-to-1 line multiplexer, the select input is defined as a 2-bit vector and output y is declared as a reg data.
- The always block has a sequential block enclosed between the keywords **`case`** and **`endcase`**.
- The block is executed whenever any of the inputs listed after the @ symbol changes in value.

Descriptions of Circuits

- ***Structural Description –*** This is directly equivalent to the schematic of a circuit and is specifically oriented to describing hardware structures using the components of a circuit.
- ***Dataflow Description –*** This describes a circuit in terms of function rather than structure and is made up of concurrent assignment statements or their equivalent. Concurrent assignments statements are executed concurrently, i.e. in parallel whenever one of the values on the right hand side of the statement changes.

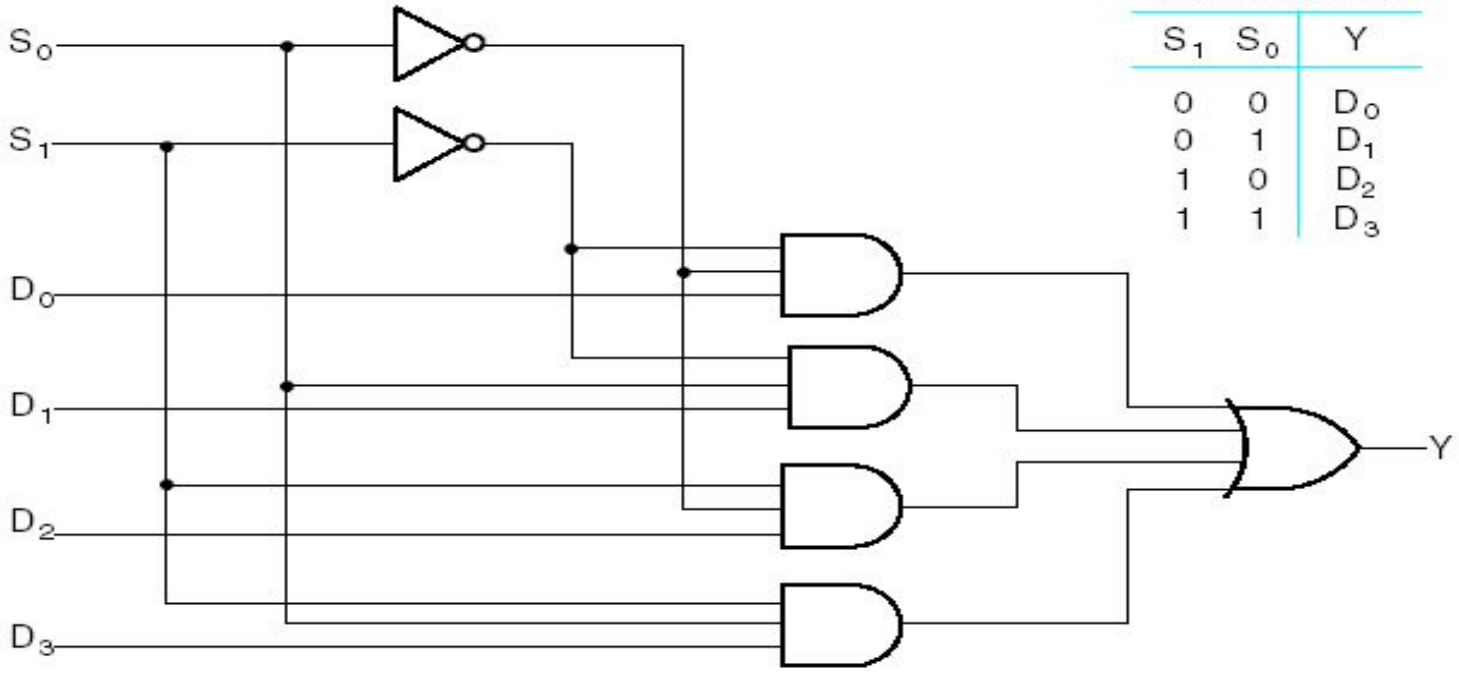
Descriptions of Circuits

- ***Hierarchical Description –*** Descriptions that represent circuits using hierarchy have multiple entities, one for each element of the Hierarchy.
- ***Behavioral Description –*** This refers to a description of a circuit at a level higher than the logic level. This type of description is also referred to as the *register transfers level*.

2-to-4 Line Decoder – Data flow description

```
//2-to-4 Line Decoder: Dataflow
module dec_2_to_4_df(E_n,A0,A1,D0_n,D1_n,D2_n,D3_n);
    input E_n, A0, A1;
    output D0_n,D1_n,D2_n,D3_n;
    assign D0_n=~(~E_n&~A1&~A0);
    assign D1_n=~(~E_n&~A1& A0);
    assign D2_n=~(~E_n& A1&~A0);
    assign D3_n=~(~E_n& A1& A0);
endmodule
```

4-to-1 Multiplexer



Function table

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Fig. 3-19 4-to-1-Line Multiplexer

4-to-1 Multiplexer

```
//4-to-1 Mux: Structural Verilog
module mux_4_to_1_st_v(S,D,Y);
   input [1:0]S;
   input [3:0]D;
   output Y;
   wire [1:0]not_s;
   wire [0:3]N;
   not g0(not_s[0],S[0]),g1(not_s[1],S[1]);
   and g2(N[0],not_s[0],not_s[1],D[0]),
   g3(N[1],S[0],not_s[1],D[0]),
   g4(N[2],not_s[0],S[1],D[0]),
   g5(N[3],S[0],S[1],D[0]);
   or g5(Y,N[0],N[1],N[2],N[3]);
endmodule
```

4-to-1 Multiplexer – Data Flow

```
//4-to-1 Mux: Dataflow description
module mux_4_to_1(S,D,Y);
   input [1:0]S;
   input [3:0]D;
   output Y;
   assign Y = (~S[1]&~S[0]&D[0])|(~S[1]&S[0]&D[1])
    |(S[1]&~S[0]&D[2])|(S[1]&S[0]&D[3]);
endmodule
```

```
//4-to-1 Mux: Conditional Dataflow description
module mux_4_to_1(S,D,Y);
   input [1:0]S;
   input [3:0]D;
   output Y;
   assign Y = (S==2'b00)?D[0] : (S==2'b01)?D[1] :
   (S==2'b10)?D[2] : (S==2'b11)?D[3]:1'bx;;
endmodule
```

4-to-1 Multiplexer

```
//4-to-1 Mux: Dataflow Verilog Description
module mux_4_to_1(S,D,Y);
   input [1:0]S;
   input [3:0]D;
   output Y;
   assign Y=S[1]?(S[0]?D[3]:D[2]):(S[0]?D[1]:D[0]);
endmodule
```

Adder

TABLE 3-7
Truth Table of Half Adder

Inputs		Outputs	
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 3-7 Truth Table of Half Adder

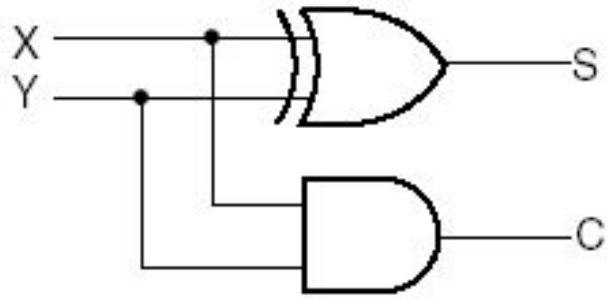
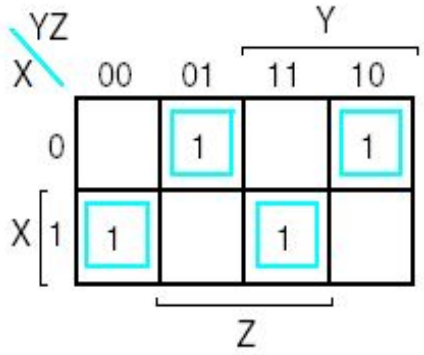


Fig. 3-25 Logic Diagram of Half Adder

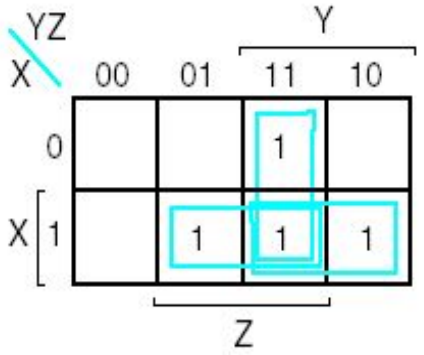
TABLE 3-8
Truth Table of Full Adder

Inputs			Outputs	
X	Y	Z	C	S
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 3-8 Truth Table of Full Adder



$S = \overline{X}\,\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XYZ$
$= X \oplus Y \oplus Z$



$C = XY + XZ + YZ$
$= XY + Z(X\overline{Y} + \overline{X}Y)$
$= XY + Z(X \oplus Y)$

Fig. 3-26 Maps for Full Adder

4-bit Adder

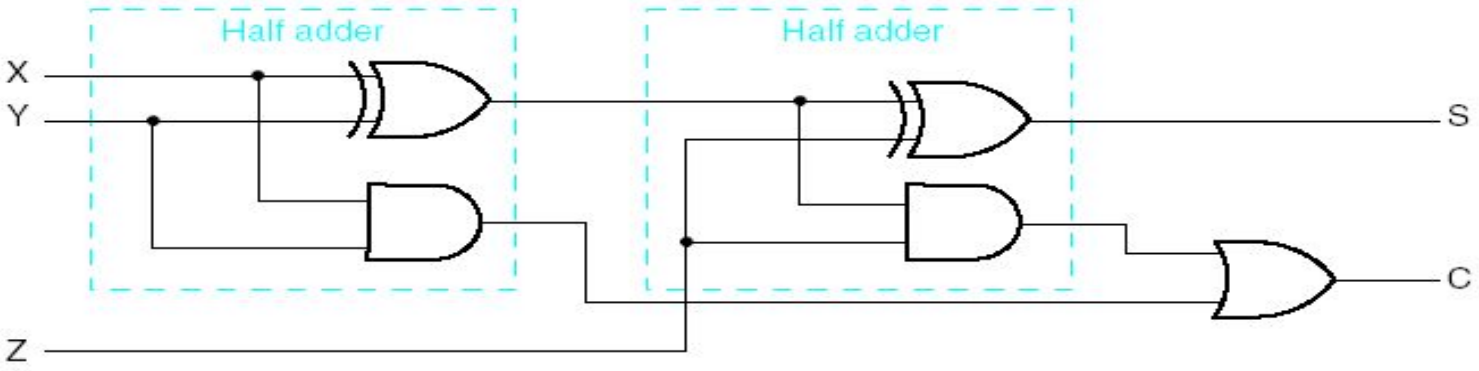


Fig. 3-27 Logic Diagram of Full Adder

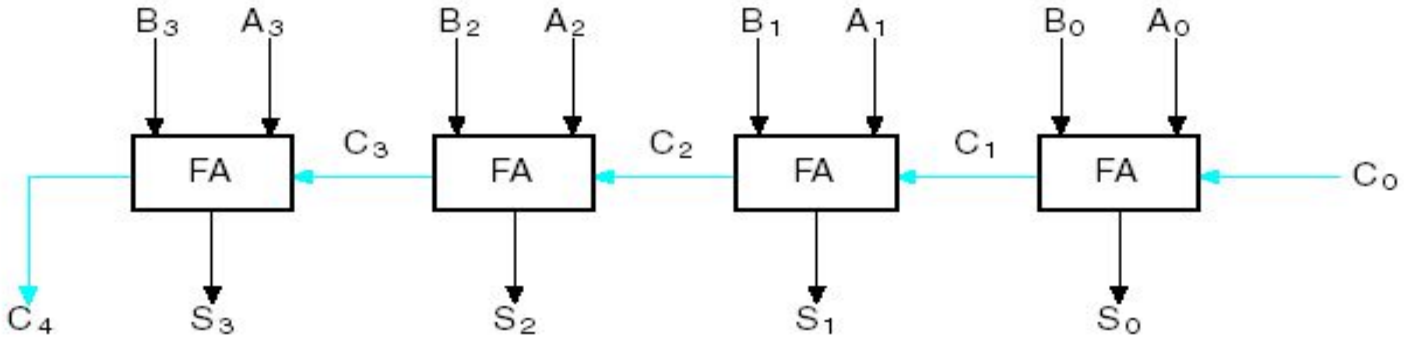


Fig. 3-28 4-Bit Ripple Carry Adder

4-bit-Adder

```
// 4-bit Adder: Hierarchical Dataflow/Structural
// (See Figures 3-27 and 3-28 for logic diagrams)

module half_adder_v(x, y, s, c);
   input x, y;
   output s, c;

   assign s = x ^ y;
   assign c = x & y;

endmodule

module full_adder_v(x, y, z, s, c);
   input x, y, z;
   output s, c;

   wire hs, hc, tc;

   half_adder_v   HA1(x, y, hs, hc),
                  HA2(hs, z, s, tc);
   assign c = tc | hc;

endmodule

module adder_4_v(B, A, C0, S, C4);
   input[3:0] B, A;
   input C0;
   output[3:0] S;
   output C4;

   wire[3:1] C;

   full_adder_v   Bit0(B[0], A[0], C0, S[0], C[1]),
                  Bit1(B[1], A[1], C[1], S[1], C[2]),
                  Bit2(B[2], A[2], C[2], S[2], C[3]),
                  Bit3(B[3], A[3], C[3], S[3], C4);
endmodule
```

4-bit Adder

```
//4-bit adder : dataflow description
module adder_4bit (A,B,C0,S,C4);
  input [3:0] A,B;
  input C0;
  output [3:0]S;
  output C4;
  assign {C4,S} = A + B + C0;
endmodule
```