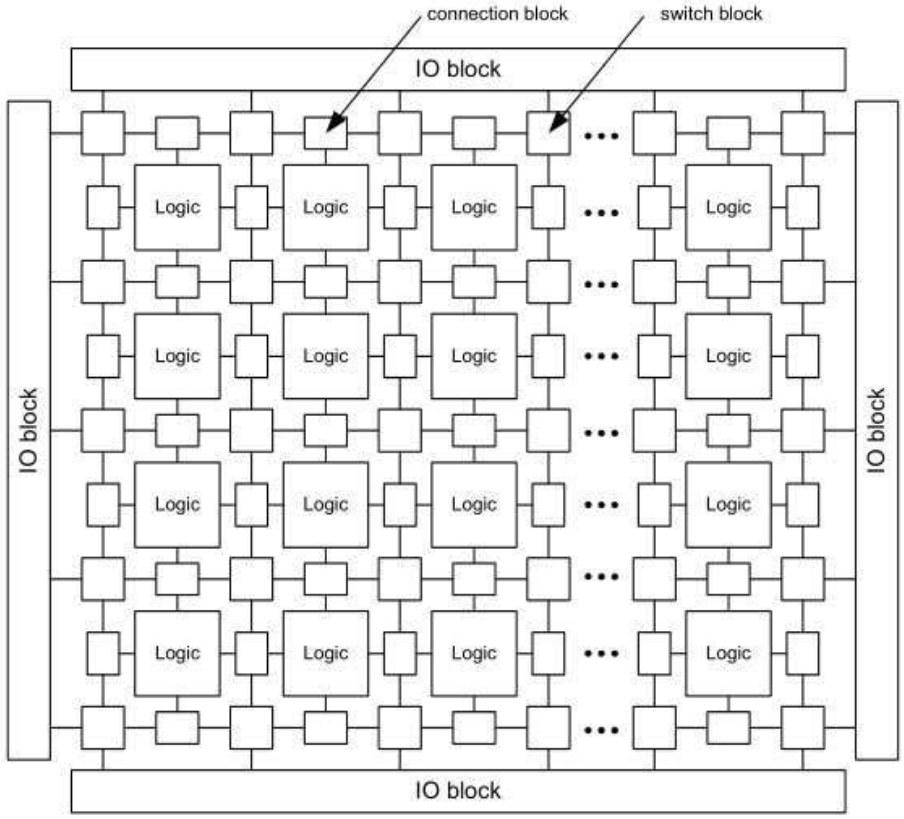

# Thank You

# Field Programmable Gate Array (FPGA)

- Typical FPGA consists of many small logic blocks interconnected by programmable routing resources.



# Tabular Form Specification of interconnection

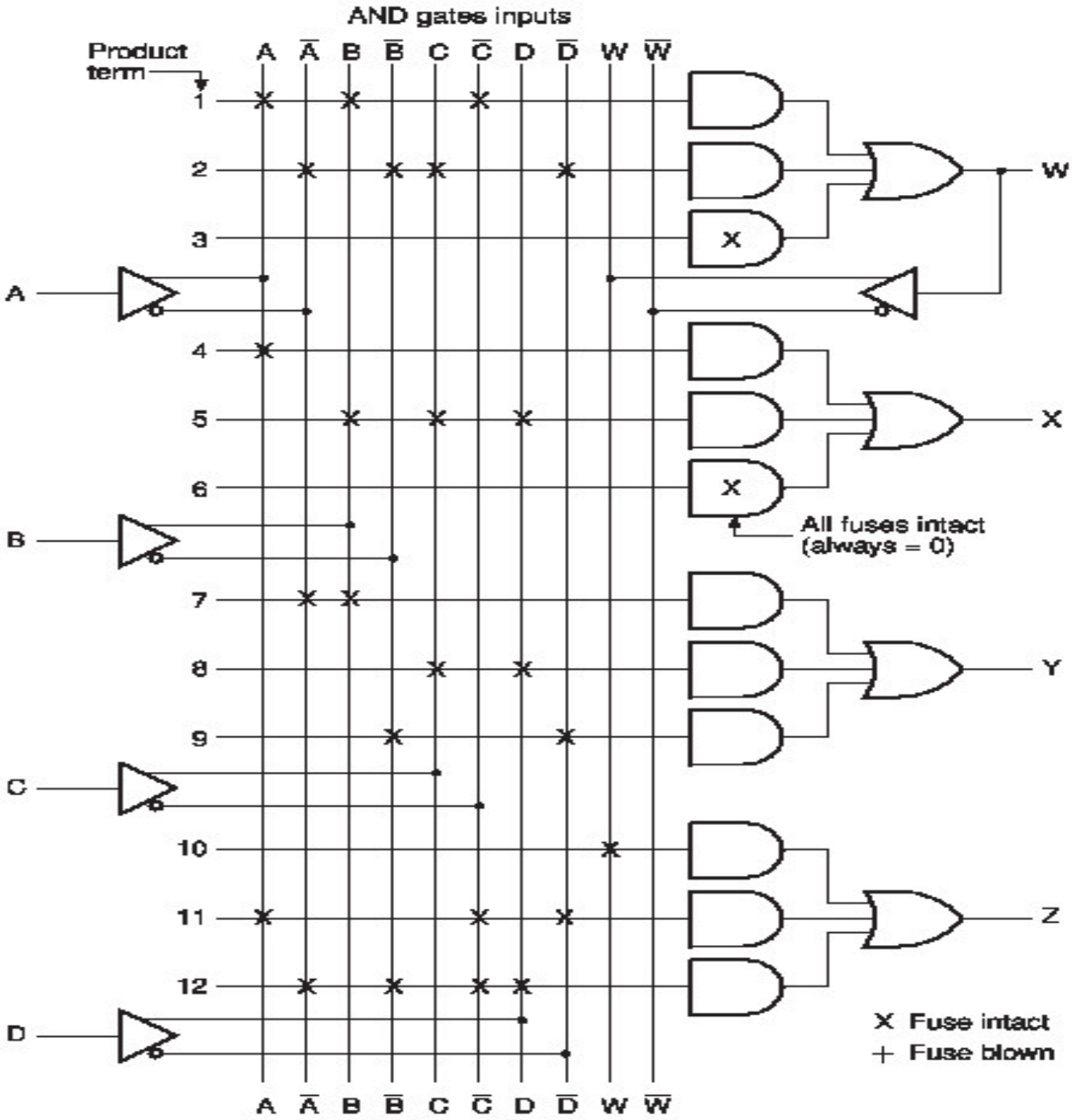
**Solution**

**Contd.:**

| Product term | AND Inputs A | B | C | D | W | Outputs |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | — | — | $W = A B \overline{C}$ |
| 2 | 0 | 0 | 1 | 0 | — | $+\overline{A}\,\overline{B} C \overline{D}$ |
| 3 | — | — | — | — | — | |
| 4 | 1 | — | — | — | — | $X = A$ |
| 5 | — | 1 | 1 | 1 | — | $+ BCD$ |
| 6 | — | — | — | — | — | |
| 7 | 0 | 1 | — | — | — | $Y = \overline{A} B$ |
| 8 | — | — | 1 | 1 | — | $+ CD$ |
| 9 | — | 0 | — | 0 | — | $+\overline{B}\,\overline{D}$ |
| 10 | — | — | — | — | 1 | $Z = W$ |
| 11 | 1 | — | 0 | 0 | — | $+ A \overline{C}\,\overline{D}$ |
| 12 | 0 | 0 | 0 | 1 | — | $+\overline{A}\,\overline{B}\,\overline{C} D$ |

**Solution**

**Contd..:**



## Solution:

**Use function simplification techniques to derive:**

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

$X = A + BCD$

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

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

$= W + AC'D' + A'B'C'D$

**Example:**

Implement the combinational circuit described by the following equations, using a PAL with 4 inputs, 4 outputs, and 3-wide AND-OR structure.

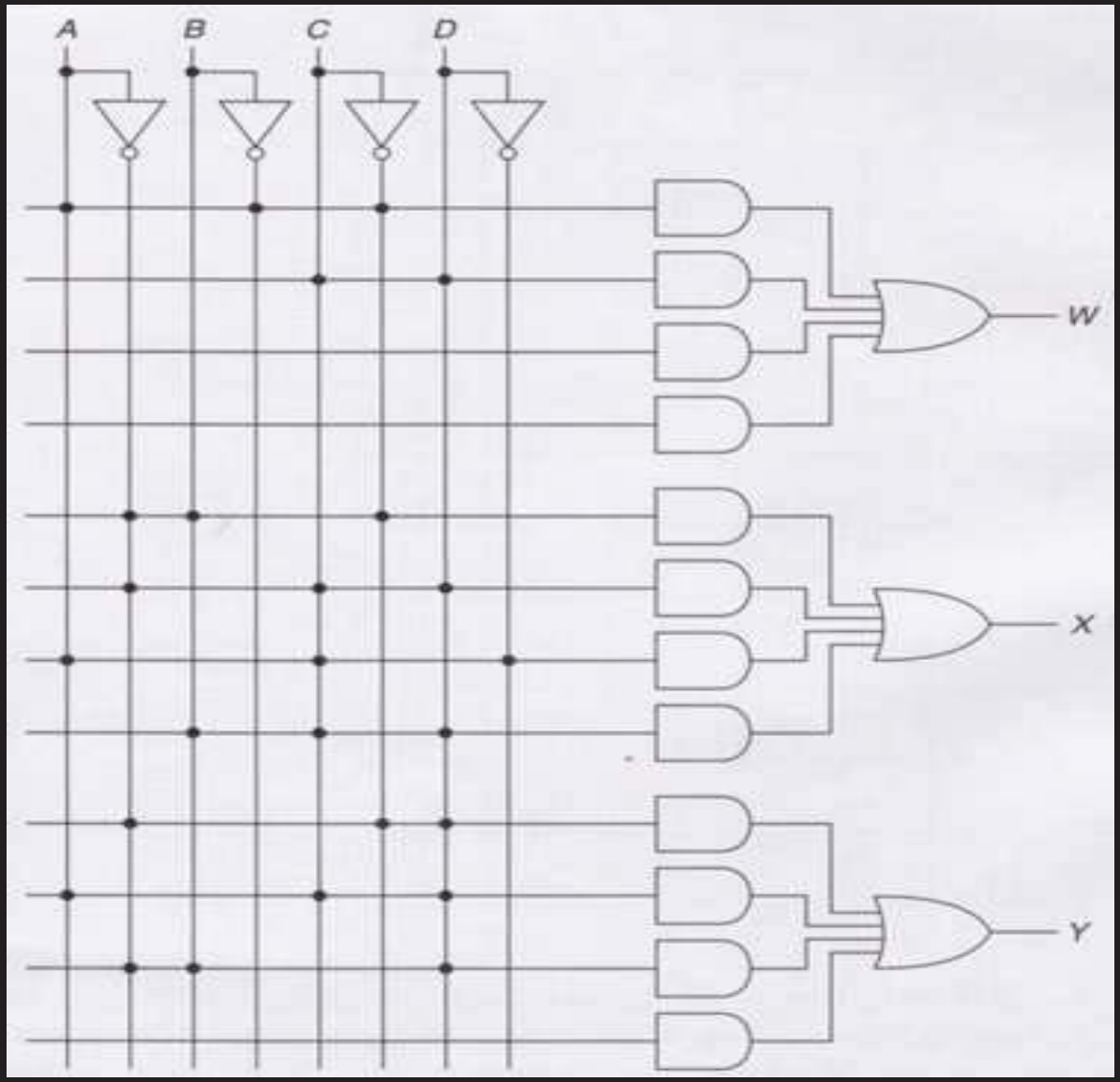
$$W(A,B,C,D) = \sum m(2,12,13)$$

$$X(A,B,C,D) = \sum m(7,8,9,10,11,12,13,14,15)$$

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

$$Z(A,B,C,D) = \sum m(1,2,8,12,13)$$

# PAL-based circuit implementation




$W = AB'C' + CD$

$X = A'BC' + A'CD + ACD' + BCD$

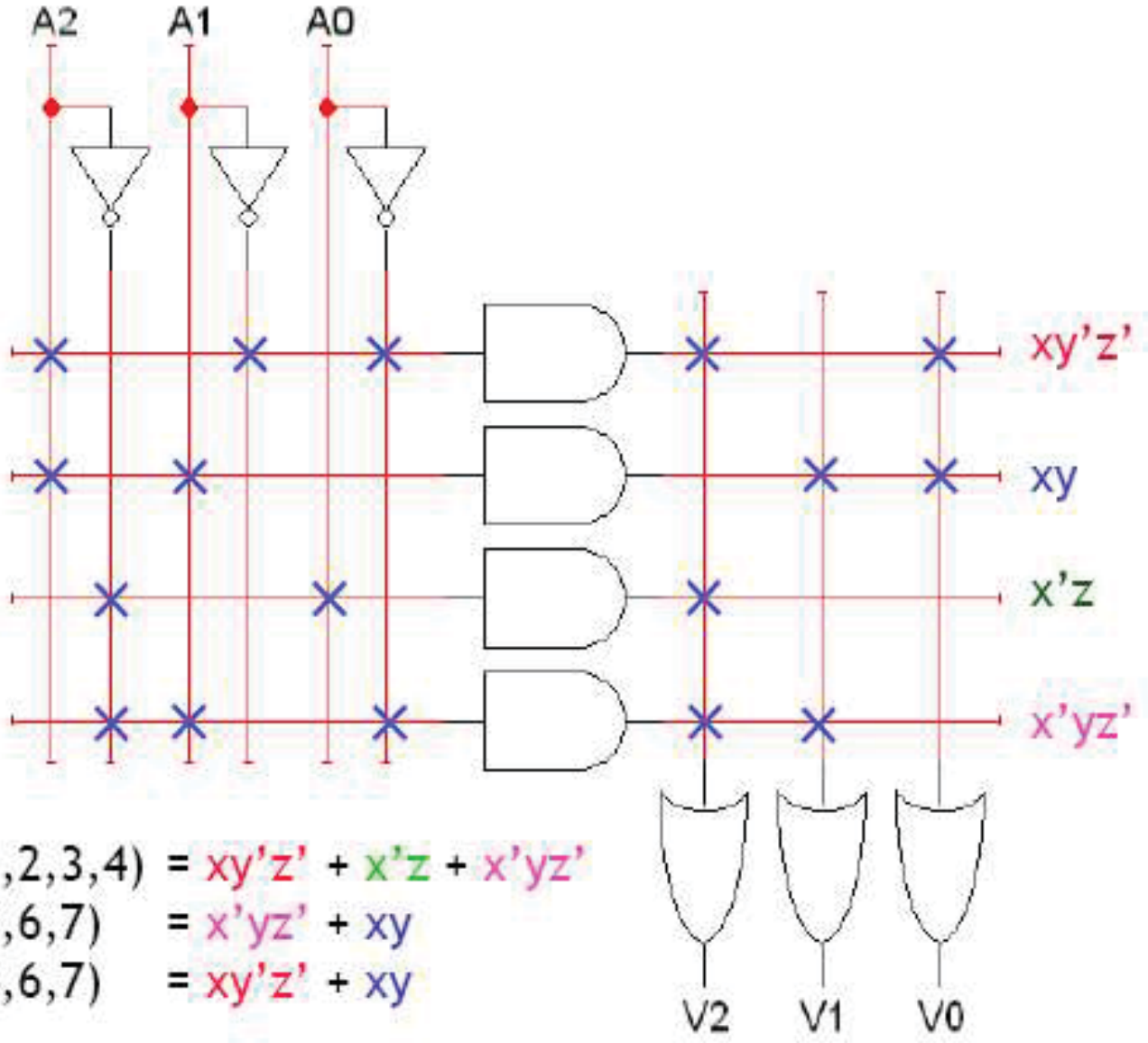
$Y = A'C'D + ACD + A'BD$

# PAL

- Similar to a PLA, but only has a programmable AND plane.
  - The OR plane is fixed.
- Not as flexible as a PLA since only certain AND gates feed each OR gate, but has fewer things that need programming.

# PLA example

- So we can implement these three functions using a 3 × 4 × 3 PLA.




$V2 = \Sigma m(1,2,3,4) = xy'z' + x'z + x'yz'$

$V1 = \Sigma m(2,6,7) = x'yz' + xy$

$V0 = \Sigma m(4,6,7) = xy'z' + xy$

# PLA minimization

- But for a PLA, what we really want is to minimize the number of product terms in *all* of the functions.
- We could express V2, V1 and V0 with just *four* total products instead.

$$V2 = \Sigma m(1,2,3,4)$$
$$V1 = \Sigma m(2,6,7)$$
$$V0 = \Sigma m(4,6,7)$$

| X \ YZ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |

$V2 = xy'z' + x'z + x'yz'$

| X \ YZ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |

$V1 = x'yz' + xy$

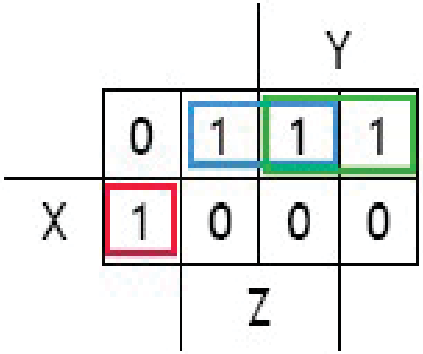
| X \ YZ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |

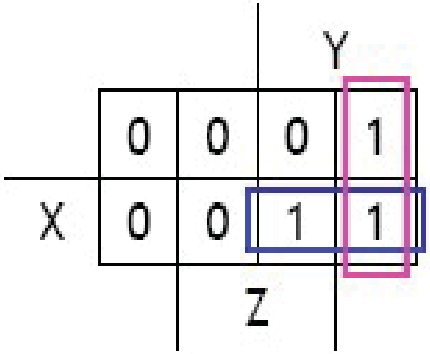
$V0 = xy'z' + xy$

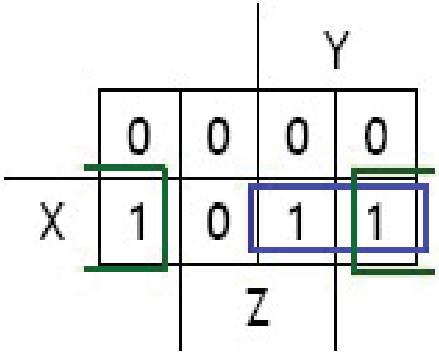
# Regular K-map minimization

- The normal K-map approach is to minimize the number of product terms for each *individual* function.
- For our three sample functions, this would result in a total of six different product terms.

$$V2 = \Sigma m(1,2,3,4)$$
$$V1 = \Sigma m(2,6,7)$$
$$V0 = \Sigma m(4,6,7)$$



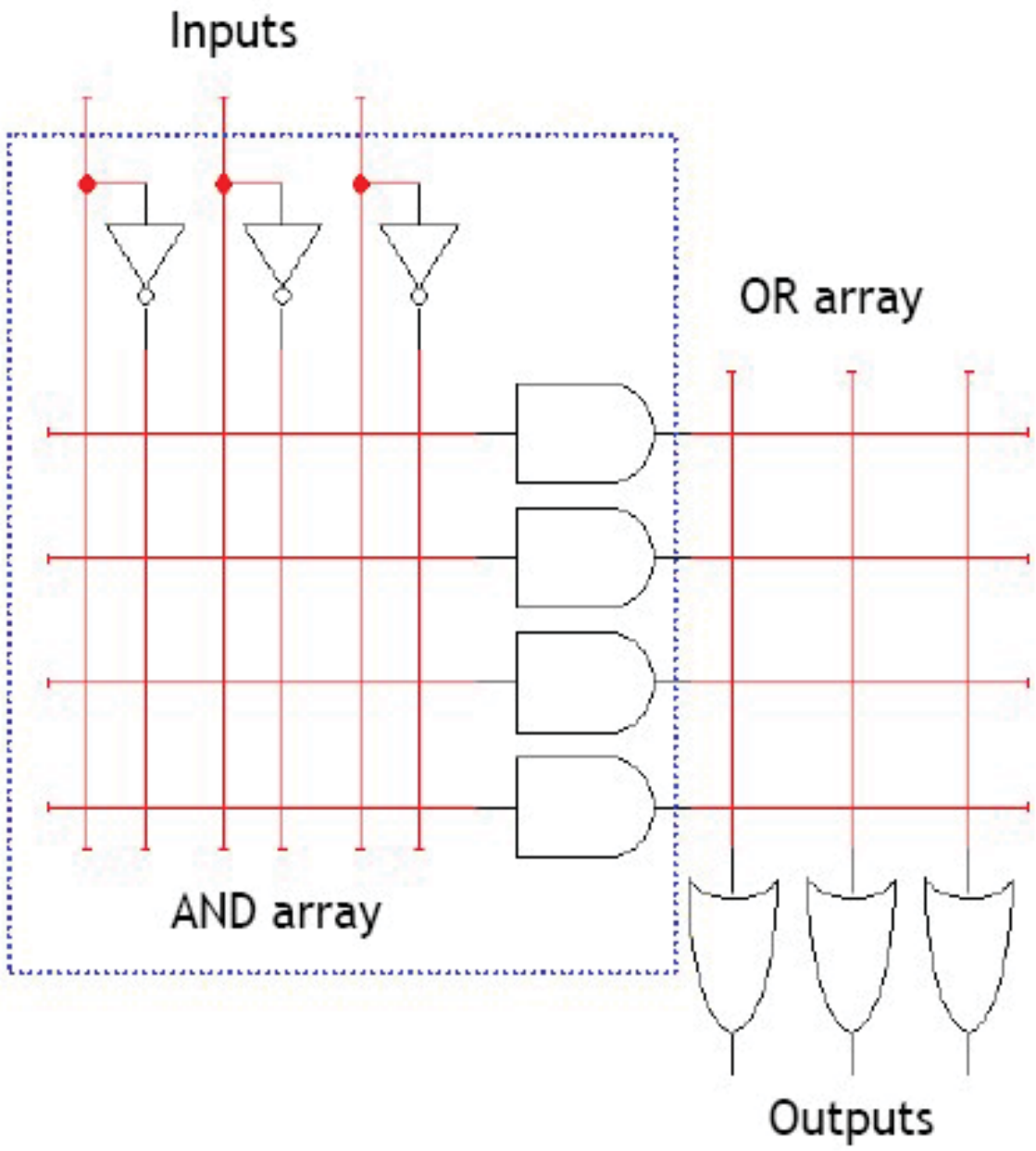




$V2 = xy'z' + x'z + x'y$ $V1 = yz' + xy$ $V0 = xz' + xy$

# A blank 3 × 4 × 3 PLA

- This is a 3 × 4 × 3 PLA (3 inputs, up to 4 product terms, and 3 outputs), ready to be programmed.
- The left part of this diagram replaces the decoder used in a ROM.
- Connections are made within the AND array to produce four arbitrary products.
- Those products are then summed in the OR array.



# PLA

It's a programmable device capable of implementing functions expressed in SOP.

- Consists of input buffers and inverters followed by:
- Programmable AND plane, followed by
- Programmable OR plane.
- Can implement '*m*' logic functions of *'n*' variables.

# Types of PLD

Programmable Read Only Memory (PROM)

Programmable Logic Array (PLA)

Programmable Array Logic (PAL)

Field Programmable Gate Arrays (**FPGAs**)

# Digital Design

## Lecture 18: Programmable Logic Devices

Contd..

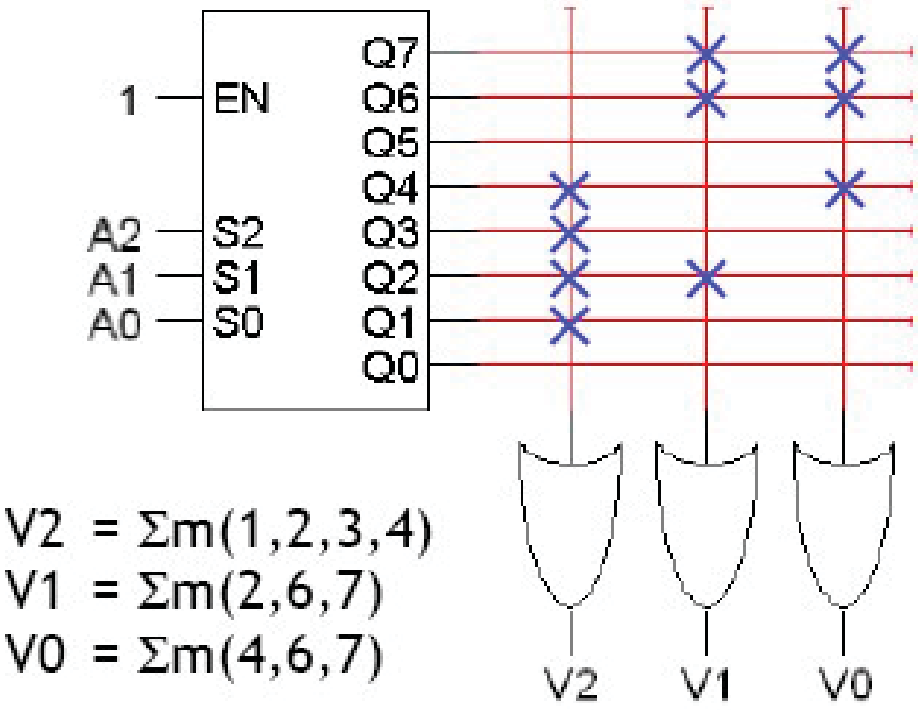
Birla Institute of Technology & Science, Pilani
Hyderabad Campus


# Thank You

- ROM implementation is not efficient for inputs with many don’t care conditions
- It uses Decoders and generates all possible ‘minterms’, thus no circuit minimization is done

# The same example again

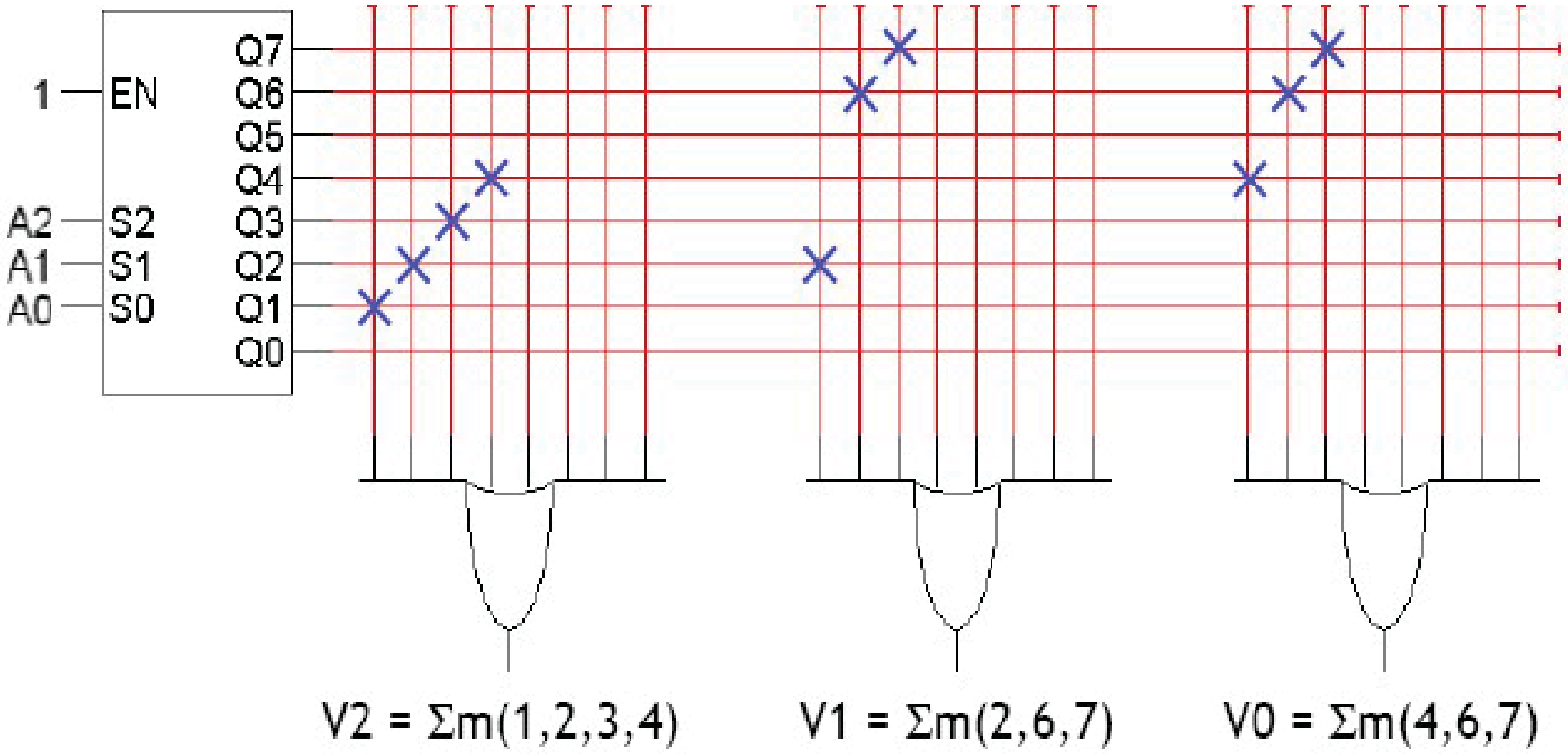
- Here is an alternative presentation of the same 8 × 3 ROM, but with some simplified OR gates just to make the diagram neater.




$V2 = \Sigma m(1,2,3,4)$
$V1 = \Sigma m(2,6,7)$
$V0 = \Sigma m(4,6,7)$

# ROM example

- Here are three functions V2, V1 and V0, implemented with our ROM.
- Blue crosses (X) indicate connections between decoder outputs and OR gates. Empty intersections are not connected.
- This is an 8 × 3 ROM since there are 8 decoder outputs and 3 OR gates.



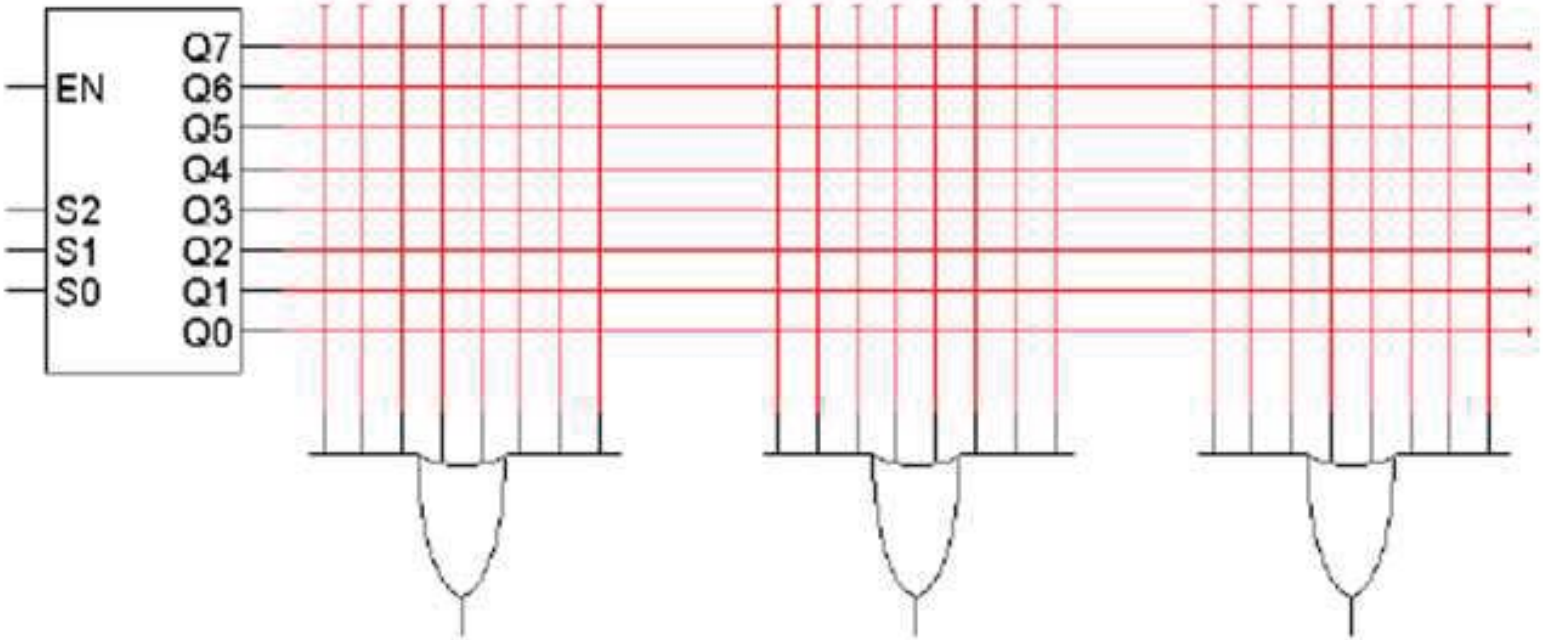
# Boolean Function Implementation using ROM

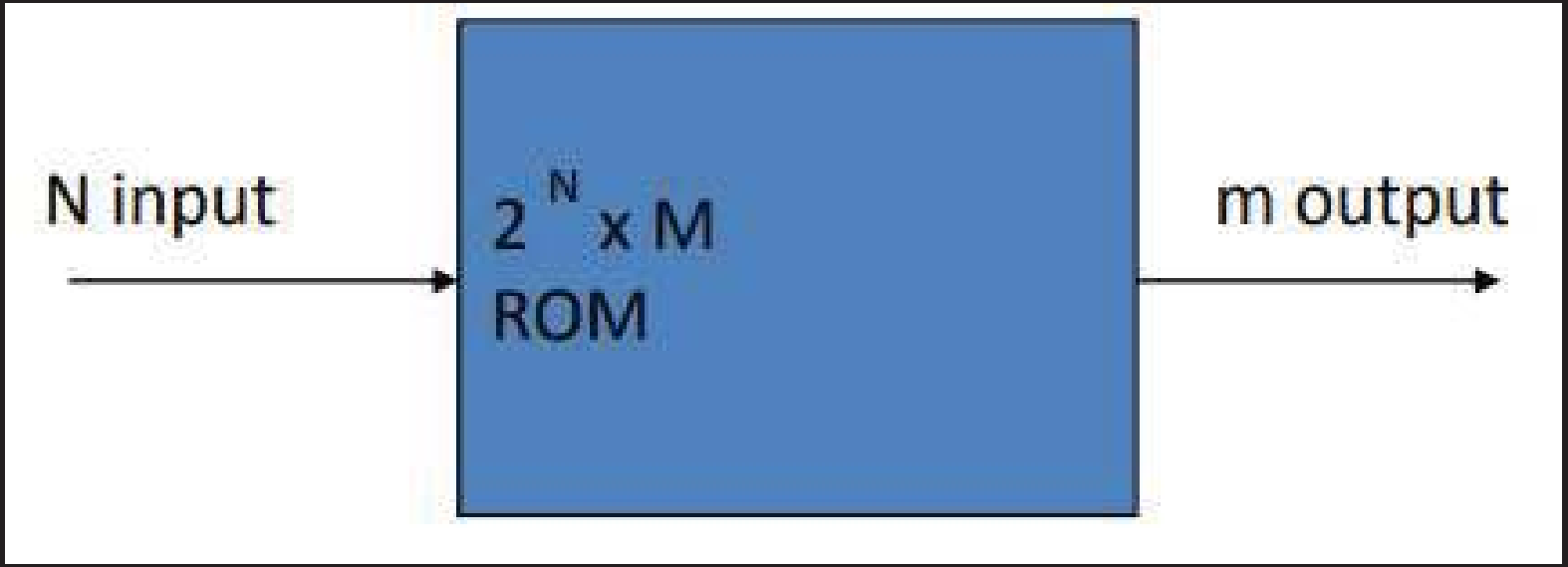
$V0 = \sum (4, 6, 7)$

$V1 = \sum (2, 6, 7)$

$V2 = \sum (1,2,3,4)$

- Building circuits with decoders is so easy that programmable logic devices are often based around decoders.
- The diagram below shows a blank read-only memory, or ROM, device.
  - It's just a decoder whose outputs may be sent to several OR gates.
  - The connections between the decoder outputs and the OR gate inputs are "programmable," so different functions can be implemented.
- To program a ROM for some specific functions, you just have to make the right connections.

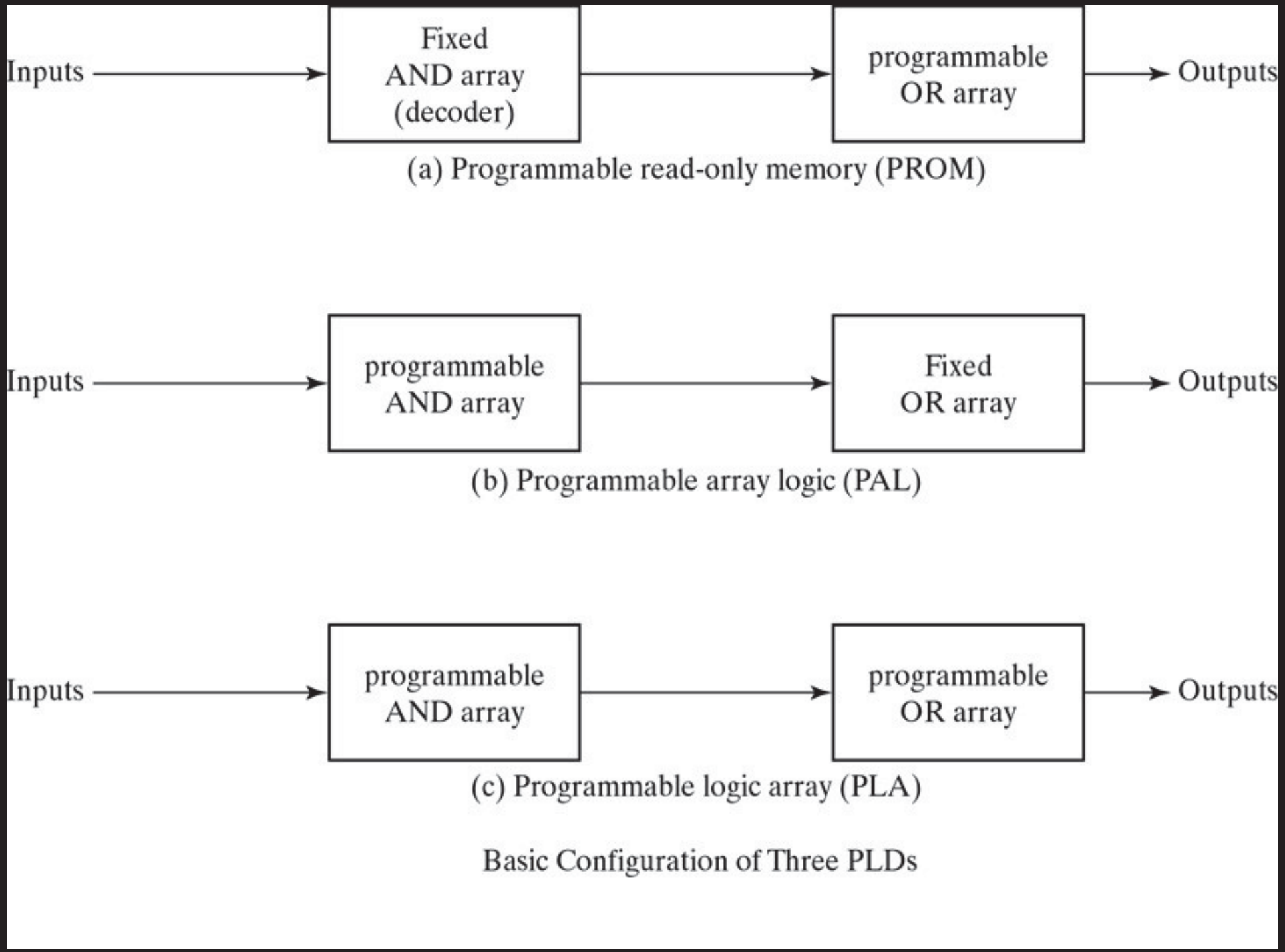





- Address: m bits; data: n bits
- ROM contains $2^N$ word of M bit each
- The input bits decide the particular word that becomes available on output lines

# ROM basics

- A Memory Device in which permanent binary Information is stored
- Used to implement combinational circuits or store binary information
- ROM includes both decoders and OR gates within a single IC




Basic Configuration of Three PLDs

# Types of PLD

Programmable Read Only Memory (PROM)

Programmable Logic Array (PLA)

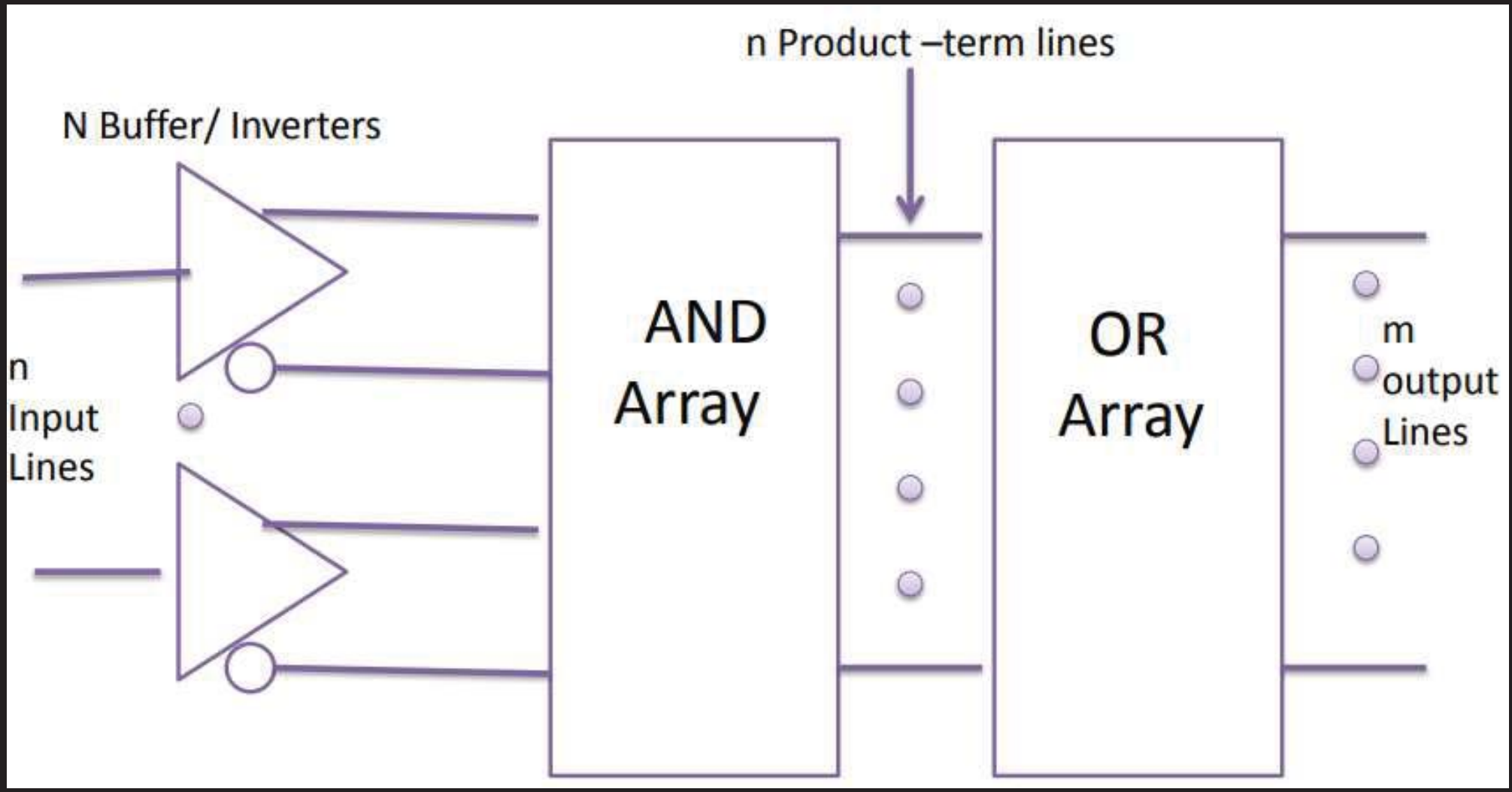
Programmable Array Logic (PAL)

Simple Programmable Logic Device (SPLD)

Complex Programmable Logic Device (CPLD)

Field Programmable Gate Arrays (**FPGAs**)

# PLD basic Architecture



# About PLDs

- Large circuit is designed on a single chip
- Can be programmed by the users as per their requirement
- Possible to implement a combinational or sequential circuit using the PLD
- Consist of array of AND, OR, NOT gates
- Any Boolean function can be represented by sum of the product (SOP) form.

# Digital Design

## Lecture 17: Programable Logic Devices

Birla Institute of Technology & Science, Pilani
Hyderabad Campus

***Thank you***

| A | Q | $Q_{-1}$ | M = 0111 |
|---|---|---|---|
| 0000 | 1101 | 0 | initial |
| 1001 | 1101 | 0 | A ← A-M |
| 1100 | 1110 | 1 | shift |
| 0011 | 1110 | 1 | A ← A+M |
| 0001 | 1111 | 0 | shift |
| 1010 | 1111 | 0 | A ← A-M |
| 1101 | 0111 | 1 | shift |
| 1110 | 1011 | 1 | shift |

Verify the operation of

(+7) x (-3)

Multiplicand M = 0111

Multiplier Q = 1101

| A | Q | $Q_{-1}$ | M | | |
|---|---|---|---|---|---|
| 0000 | 0011 | 0 | 0111 | Initial Values | |
| 1001 | 0011 | 0 | 0111 | A ← A - M | First Cycle |
| 1100 | 1001 | 1 | 0111 | Shift | |
| 1110 | 0100 | 1 | 0111 | Shift | Second Cycle |
| 0101 | 0100 | 1 | 0111 | A ← A + M | Third Cycle |
| 0010 | 1010 | 0 | 0111 | Shift | |
| 0001 | 0101 | 0 | 0111 | Shift | Fourth Cycle |

| A | Q | $Q_{-1}$ | M | | |
|---|---|---|---|---|---|
| 0000 | 0011 | 0 | 0111 | Initial Values | |
| 1001 | 0011 | 0 | 0111 | A ← A - M | First Cycle |
| 1100 | 1001 | 1 | 0111 | Shift | |
| 1110 | 0100 | 1 | 0111 | Shift | Second Cycle |
| 0101 | 0100 | 1 | 0111 | A ← A + M | Third Cycle |
| 0010 | 1010 | 0 | 0111 | Shift | |

| A | Q | $Q_{-1}$ | M | | |
|---|---|---|---|---|---|
| 0000 | 0011 | 0 | 0111 | Initial Values | |
| 1001 | 0011 | 0 | 0111 | A ← A - M | First Cycle |
| 1100 | 1001 | 1 | 0111 | Shift | |
| 1110 | 0100 | 1 | 0111 | Shift | Second Cycle |
| 0101 | 0100 | 1 | 0111 | A ← A + M | |

| A | Q | $Q_{-1}$ | M | | |
|---|---|---|---|---|---|
| 0000 | 0011 | 0 | 0111 | Initial Values | |
| 1001 | 0011 | 0 | 0111 | $A \leftarrow A - M$ | First Cycle |
| 1100 | 1001 | 1 | 0111 | Shift | |
| 1110 | 0100 | 1 | 0111 | Shift | Second Cycle |

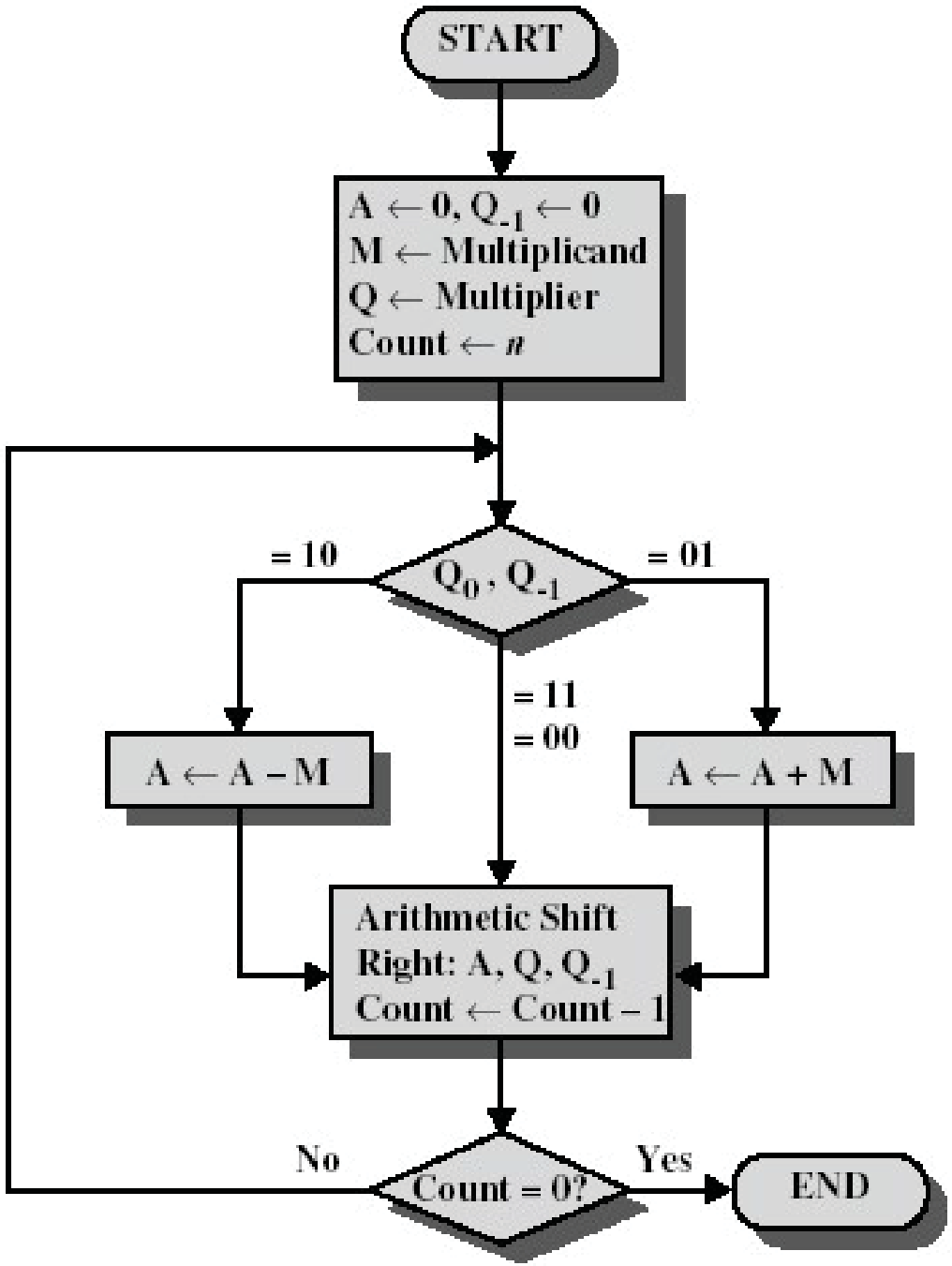
| A | Q | $Q_{-1}$ | M | | |
|---|---|---|---|---|---|
| 0000 | 0011 | 0 | 0111 | Initial Values | |
| 1001 | 0011 | 0 | 0111 | A ← A - M | First Cycle |
| 1100 | 1001 | 1 | 0111 | Shift | |

| A | Q | $Q_{-1}$ | M | |
|---|---|---|---|---|
| 0000 | 0011 | 0 | 0111 | Initial Values |
| 1001 | 0011 | 0 | 0111 | A ← A - M |

| A | Q | $Q_{-1}$ | M | |
|---|---|---|---|---|
| 0000 | 0011 | 0 | 0111 | Initial Values |

# Booth's Algorithm

1. Multiplier and multiplicand are placed in Q and M registers.
2. A 1-bit register is placed to the right of the least significant bit ($Q_0$) and designated $Q_{-1}$.
3. Control logic scans the bits of the multiplier one at a time; but a bit and its bit to the right are examined. If the bits are the same (1-1 or 0-0) then all bits of the A,Q, and $Q_{-1}$ registers are shifted to the right 1 bit. If the two bits differ, then the multiplicand is added/subtracted depending on whether the bits are 0-1 or 1-0. Addition is followed by a **right arithmetic shift.**



- Booth re-coding Procedure working from LSB to MSB retain each 0 until a 1 is reached

- When a 1 is encountered insert $\bar{1}$ at that position and complement all the succeeding 1's until a 0 is encountered

- Replace that 0 with 1 and continue same

- while multiplying with $\bar{1}$; 2's compliment addition is used

- Booth's algorithm called skipping over one's
- String of 1's replaced by 0's
- **For example: 30 = 0011110**
- = 32 – 2 = 0100000 - 0000010
- **In the coded form = 0100$\bar{1}$0**

Multiplicand ‘M’ unchanged

The multiplier ‘Q’ to a re-coded value R

Each digit can assume a negative as well as positive and zero values

**Signed Digit ( SD) encoding**

**This Approach will not work if numbers are signed**

ALTERNATIVE

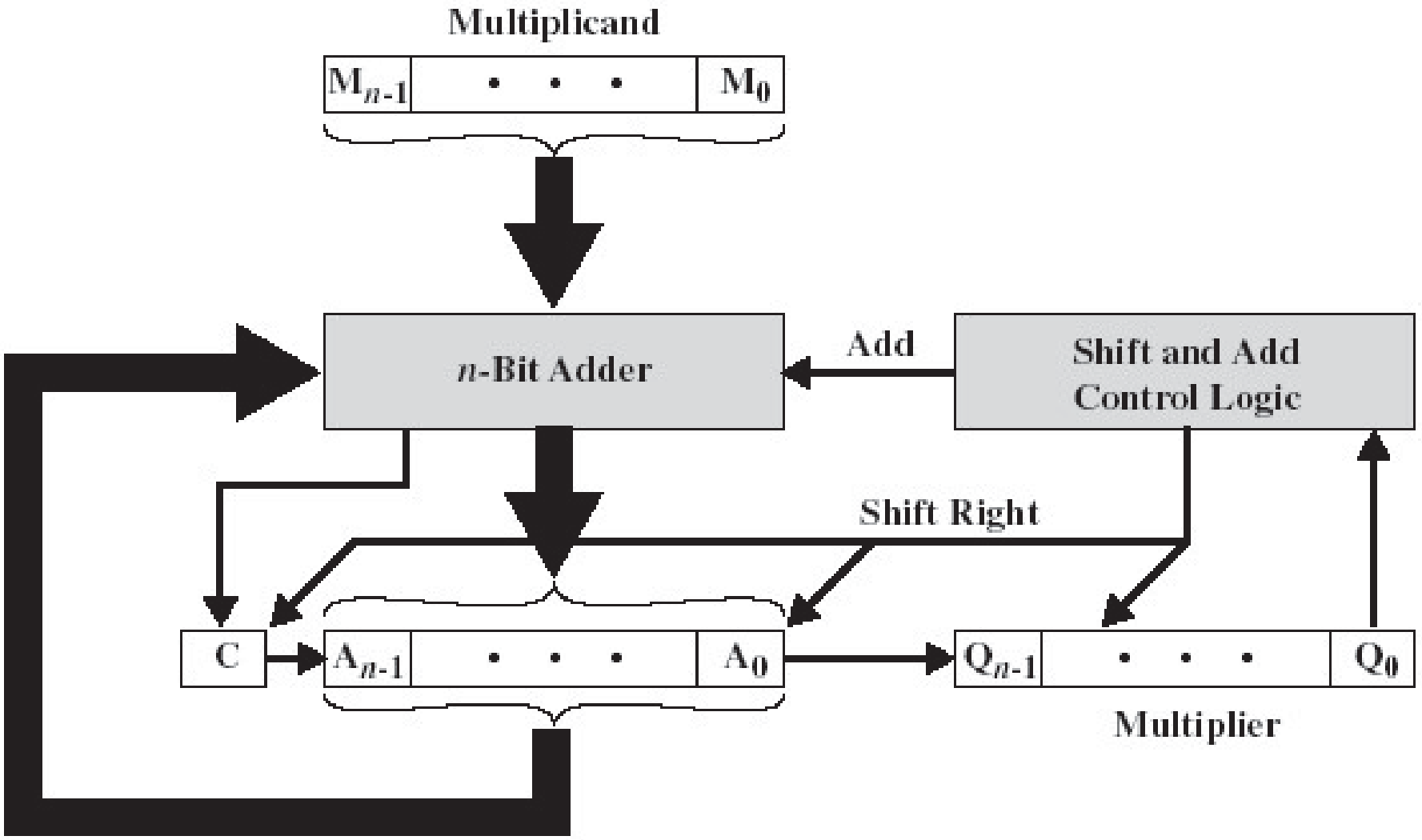
BOOTH'S MULTIPLIER

| C | A | Q | M | | |
|---|---|---|---|---|---|
| 0 | 0000 | 1101 | 1011 | Initial Values | |
| 0 | 1011 | 1101 | 1011 | Add | First Cycle |
| 0 | 0101 | 1110 | 1011 | Shift | |
| 0 | 0010 | 1111 | 1011 | Shift | Second Cycle |
| 0 | 1101 | 1111 | 1011 | Add | Third Cycle |
| 0 | 0110 | 1111 | 1011 | Shift | |
| 1 | 0001 | 1111 | 1011 | Add | Fourth Cycle |
| 0 | 1000 | 1111 | 1011 | Shift | |

| C | A | Q | M | | |
|---|---|---|---|---|---|
| 0 | 0000 | 1101 | 1011 | Initial Values | |
| 0 | 1011 | 1101 | 1011 | Add | First Cycle |
| 0 | 0101 | 1110 | 1011 | Shift | |
| 0 | 0010 | 1111 | 1011 | Shift | Second Cycle |

| C | A | Q | M | | |
|---|---|---|---|---|---|
| 0 | 0000 | 1101 | 1011 | Initial Values | |
| 0 | 1011 | 1101 | 1011 | Add | First Cycle |
| 0 | 0101 | 1110 | 1011 | Shift | |

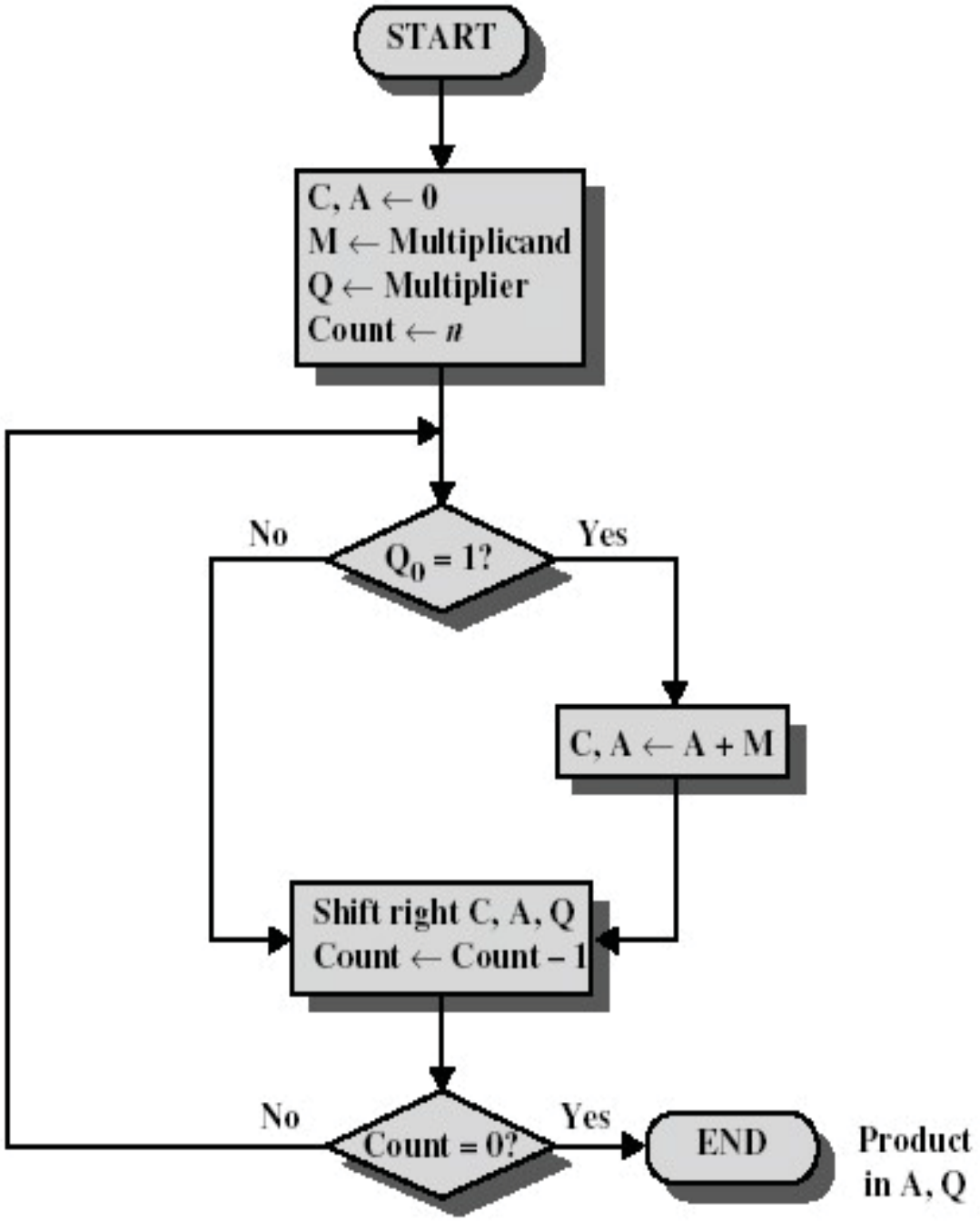
| C | A | Q | M | |
|---|---|---|---|---|
| 0 | 0000 | 1101 | 1011 | Initial Values |




1. Multiplier and multiplicand are loaded into registers Q and M.
2. A third register (A) is initially set to zero.
3. A one-bit C register (initialised to zero) holds carry bits.

# Multiplication

1. The Digital system can keep a running product rather than summing at the end.
2. For each ‘1’ in the multiplier we can apply an ‘add’ and a ‘shift’.
3. For each ‘0’ only a shift is needed.



# Multiplication

## Some general observations

- Multiplication involves the generation of partial products – one for each digit in the multiplier.
- Partial products are summed to produce the final product.
- Partial products are simple to define for binary multiplication. If the digit in multiplier is 'one' the partial product is the multiplicand, otherwise the partial product is zero.
- The total product is the sum of the partial products. Each successive partial product is shifted one position to the left.
- The multiplication of two n-bit binary numbers results in a product of up to 2n bits in length.

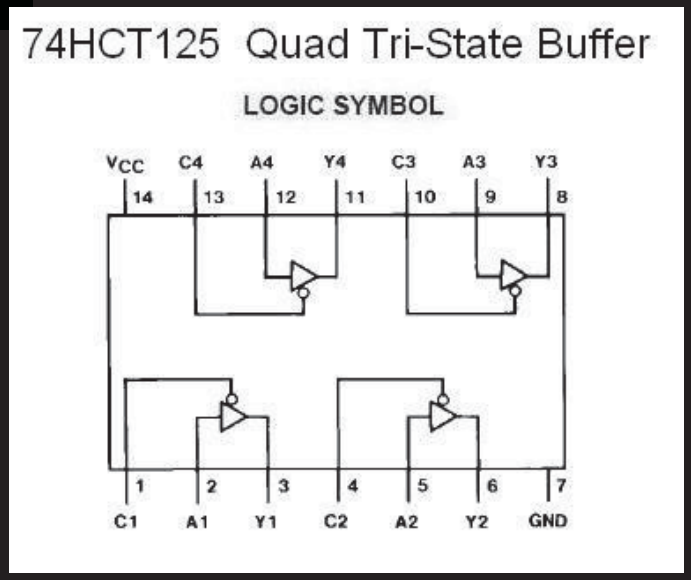
```
   1011
   1101
  ------
   1011
  0000
 1011
1011
--------
10001111
```

# Digital Design

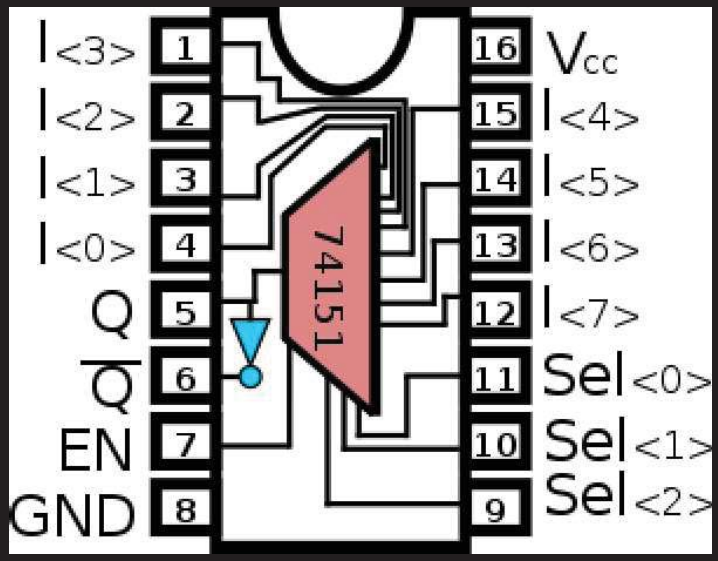
# Lecture No. 16
## MULTIPLICATION ALGORITHMS


*Thank You*

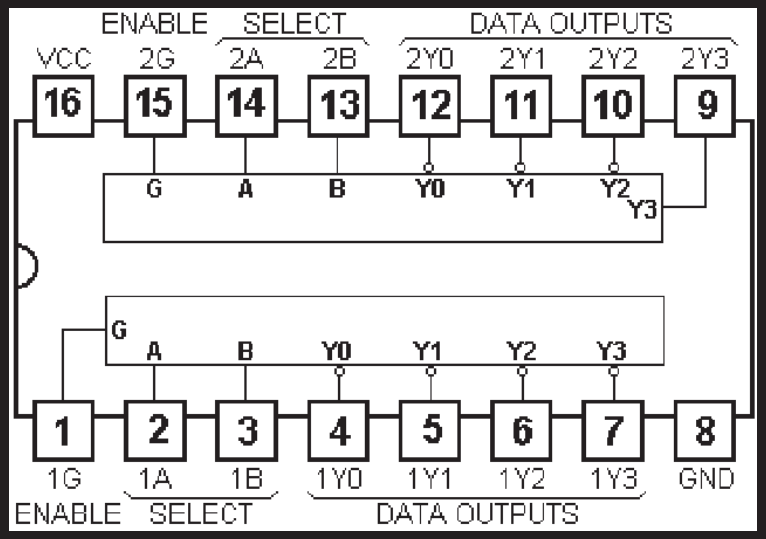
# 74XX series ICs



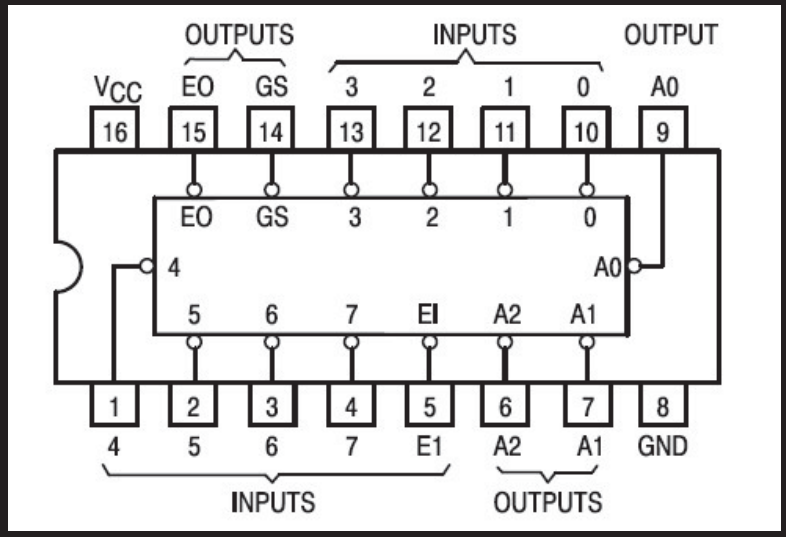

Tri-state Buffer




Mux

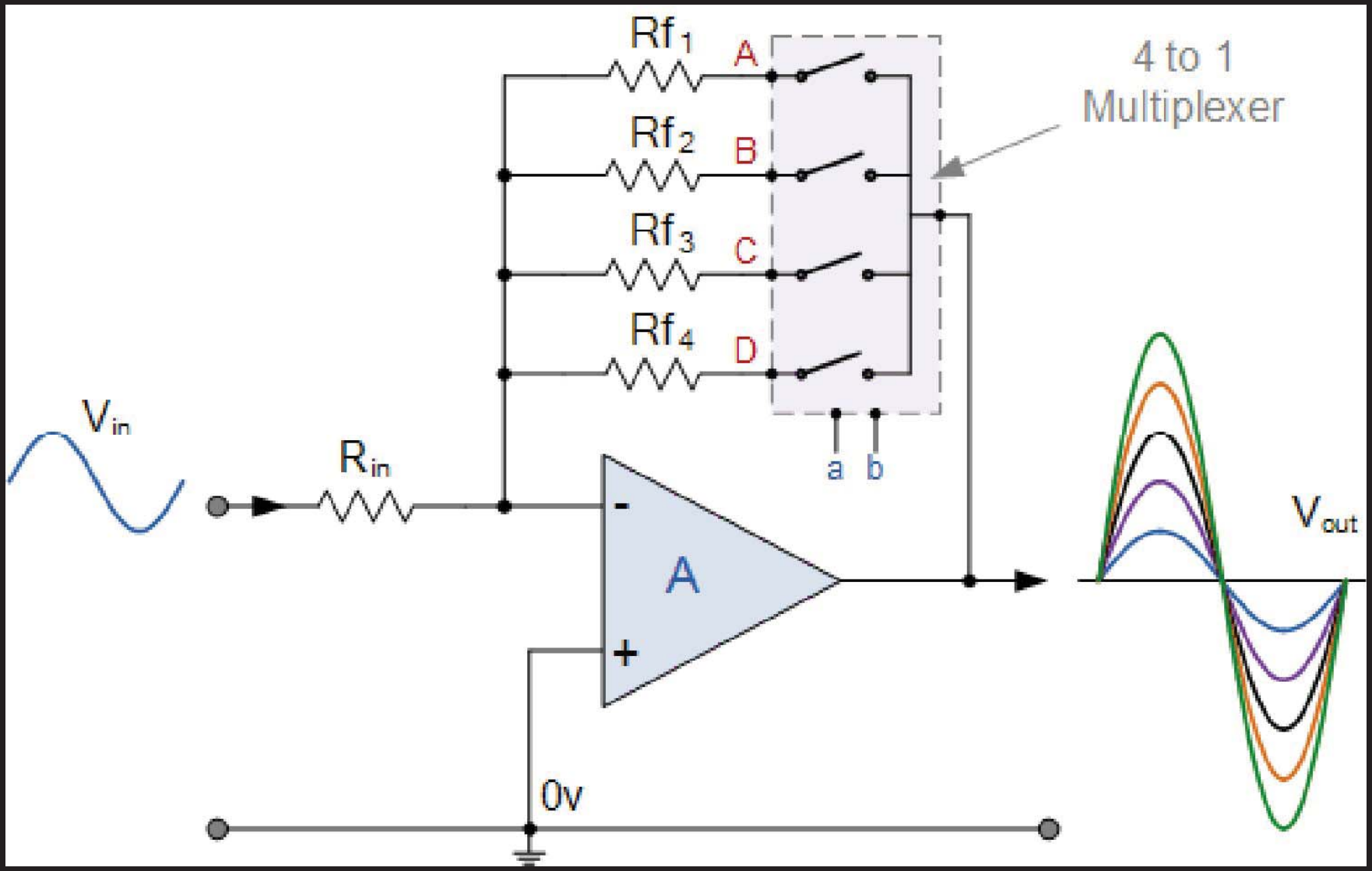



Decoder




Encoder

# Multiplexers: Realization using Op-Amp



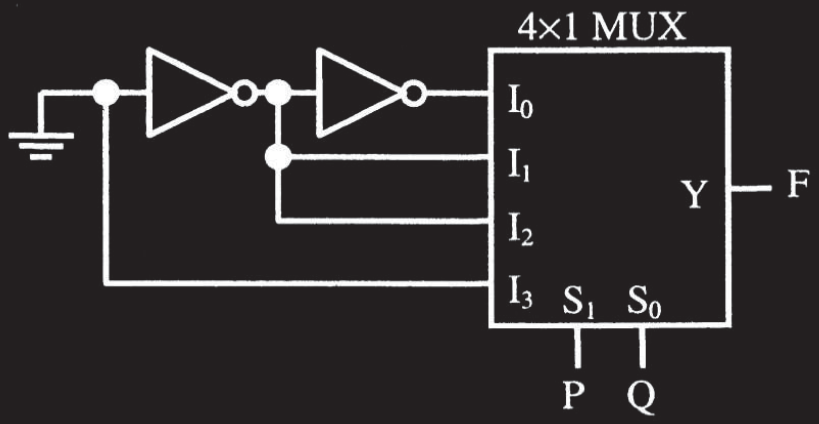
# Multiplexers

## Boolean Function implementation

1. F (X,Y,Z) = ∑ (1, 3, 5, 6, 7) using 4:1 MUX and 2:1 MUX

2. F (A, B, C, D) = ∑ (1, 3, 4, 11, 12, 13, 14, 15) Using 8:1 MUX and 4:1 MUX

3. The logic function implemented by the circuit below is (Ground implies logic '0')



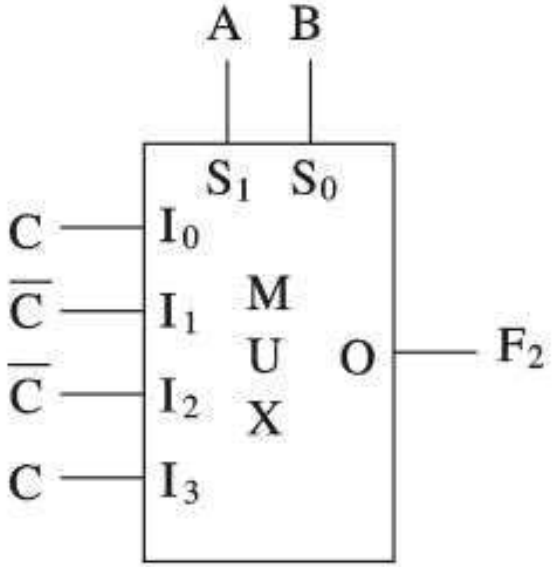
# Efficient implementation: Even-parity function

Original truth table

| A | B | C | $F_1$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

New truth table

| A | B | $F_1$ |
|---|---|---|
| 0 | 0 | C |
| 0 | 1 | $\overline{C}$ |
| 1 | 0 | $\overline{C}$ |
| 1 | 1 | C |



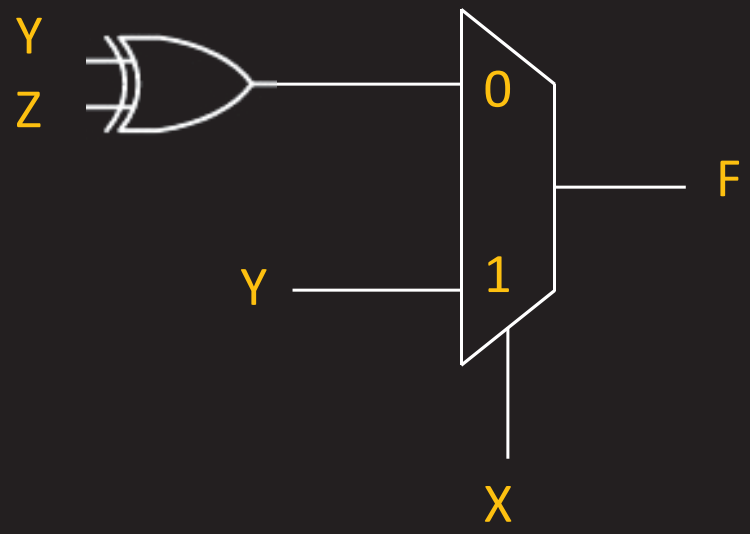
# Multiplexers

## Boolean Function implementation

$F(X,Y,Z) = \sum (1, 2, 6, 7)$

| X | Y | Z | F | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | |
| 0 | 0 | 1 | 1 | $F = Y \oplus Z$ |
| 0 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | $F = Y$ |
| 1 | 1 | 1 | 1 | |

2:1 Multiplexer



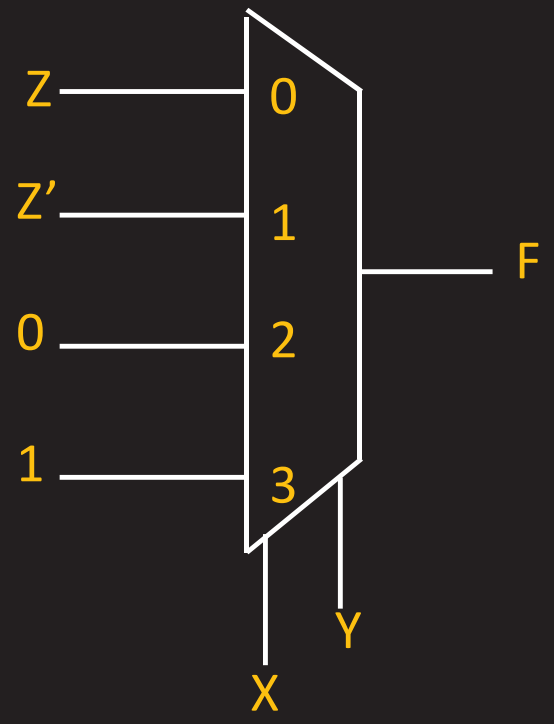
# Multiplexers

## Boolean Function implementation

$F(X,Y,Z) = \sum (1, 2, 6, 7)$

| X | Y | Z | F | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | F = Z |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | F = Z’ |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | F = 0 |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | F = 1 |
| 1 | 1 | 1 | 1 | |

4:1 Multiplexer

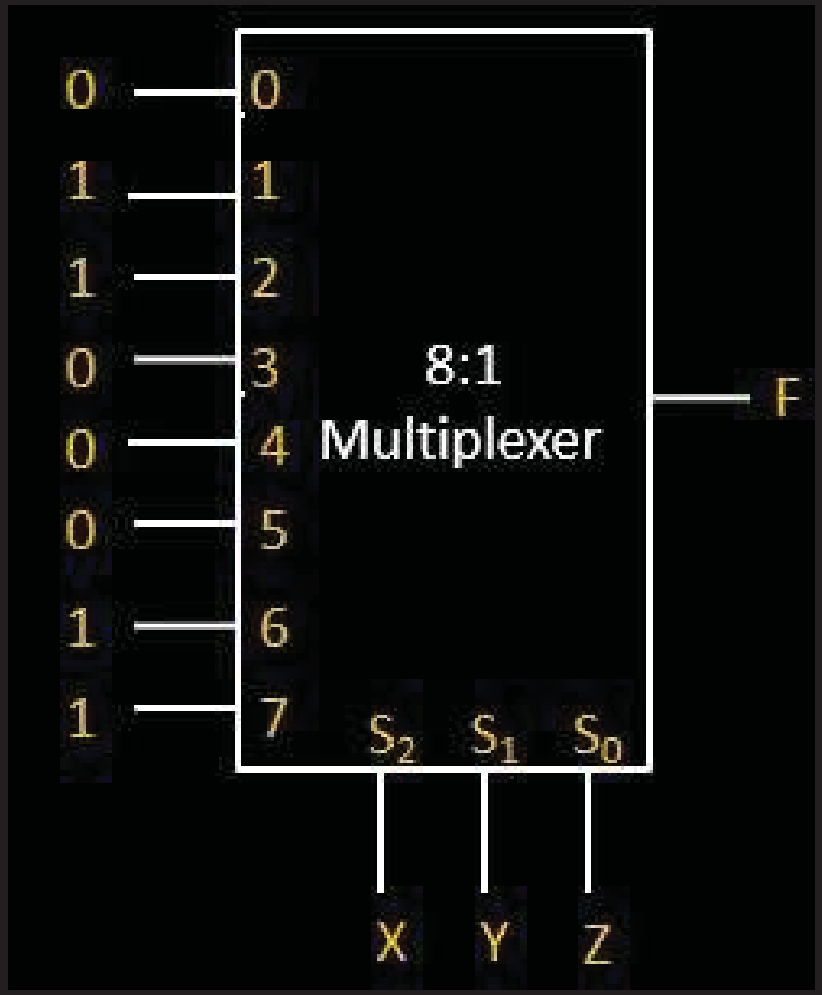


# Multiplexers

Boolean Function implementation using MUX,

$F(X,Y,Z) = \sum (1, 2, 6, 7)$

| X | Y | Z | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



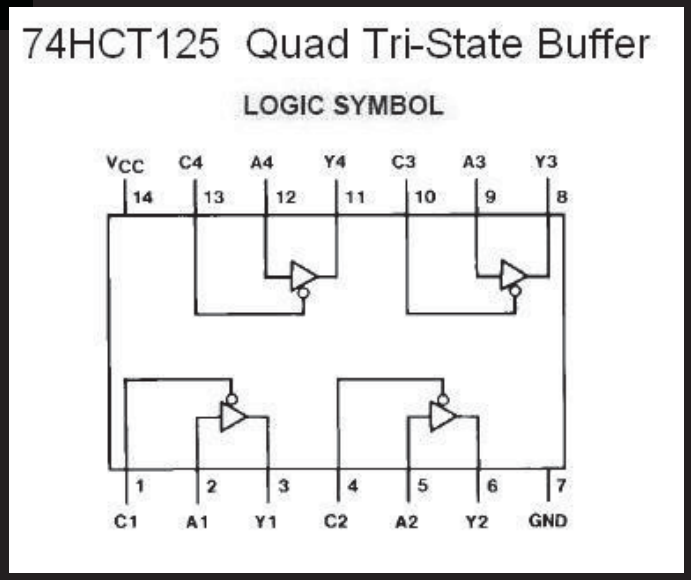
# Digital Design

Lecture 15: Multiplexer and Demultiplexer

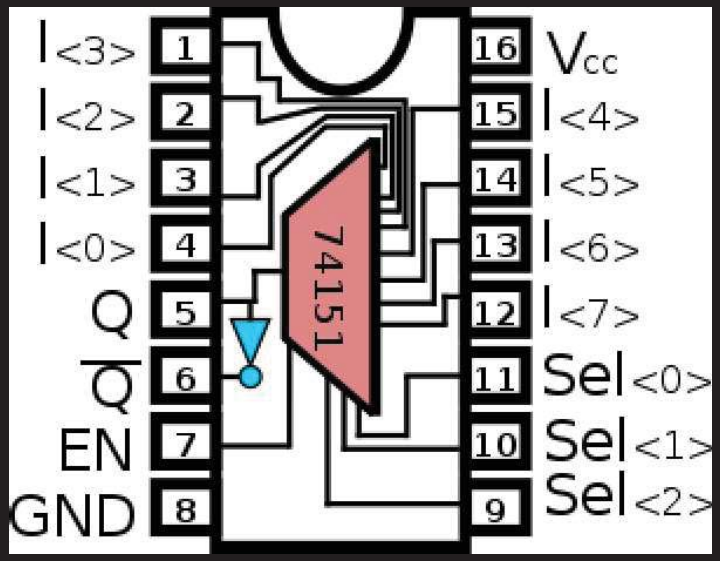
Contd....


*Thank You*

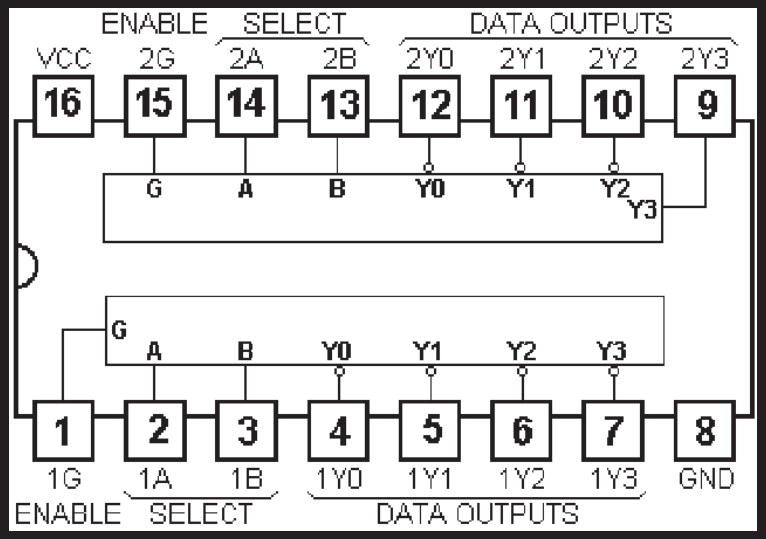
# 74XX series ICs



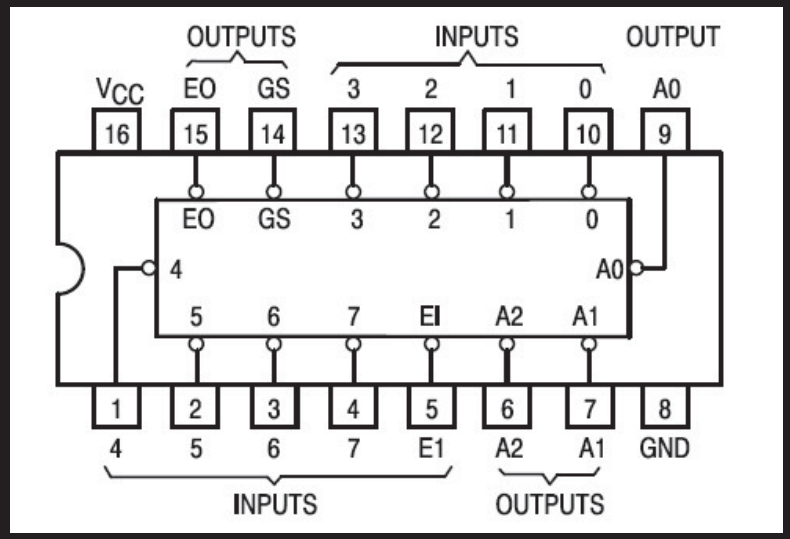

Tri-state Buffer




Mux

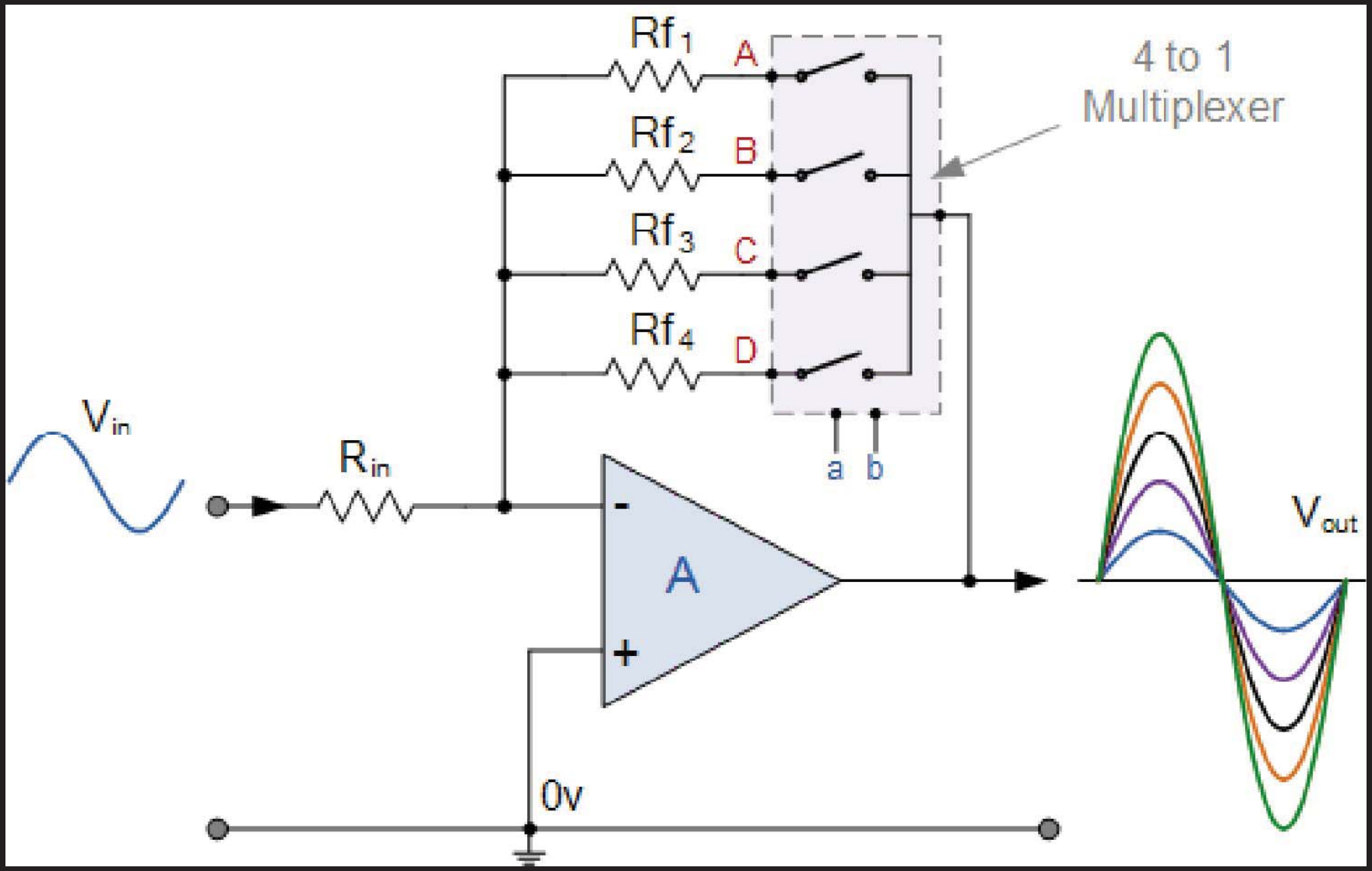



Decoder




Encoder

# Multiplexers: Realization using Op-Amp



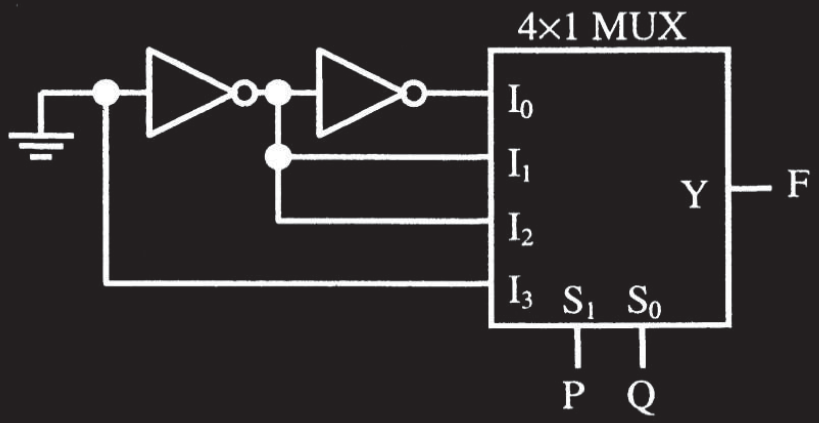
# Multiplexers

## Boolean Function implementation

1. F (X,Y,Z) = ∑ (1, 3, 5, 6, 7) using 4:1 MUX and 2:1 MUX

2. F (A, B, C, D) = ∑ (1, 3, 4, 11, 12, 13, 14, 15) Using 8:1 MUX and 4:1 MUX

3. The logic function implemented by the circuit below is (Ground implies logic '0')



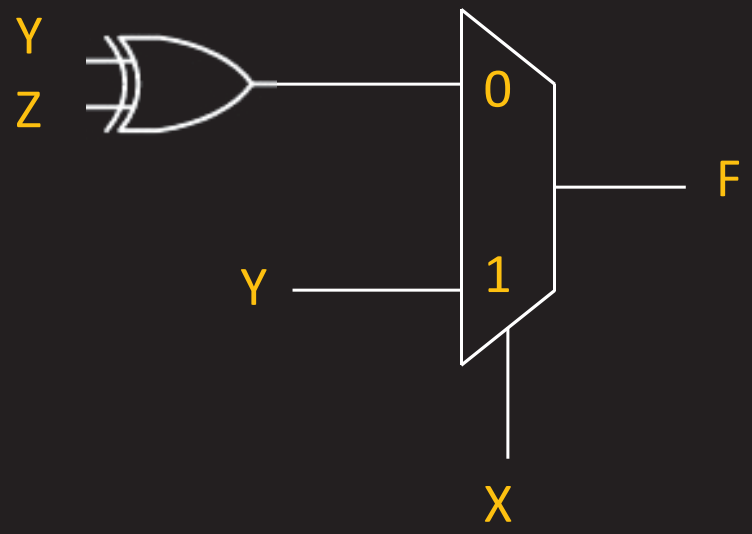
# Multiplexers

## Boolean Function implementation

$F(X,Y,Z) = \sum (1, 2, 6, 7)$

| X | Y | Z | F | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | |
| 0 | 0 | 1 | 1 | $F = Y \oplus Z$ |
| 0 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | $F = Y$ |
| 1 | 1 | 1 | 1 | |

2:1 Multiplexer



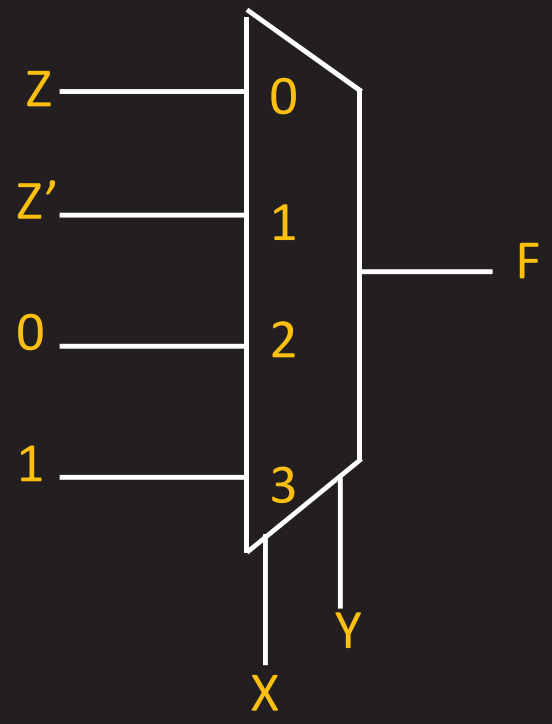
# Multiplexers

## Boolean Function implementation

$F(X,Y,Z) = \sum (1, 2, 6, 7)$

| X | Y | Z | F | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | F = Z |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | F = Z’ |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | F = 0 |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 1 | F = 1 |
| 1 | 1 | 1 | 1 | |

4:1 Multiplexer

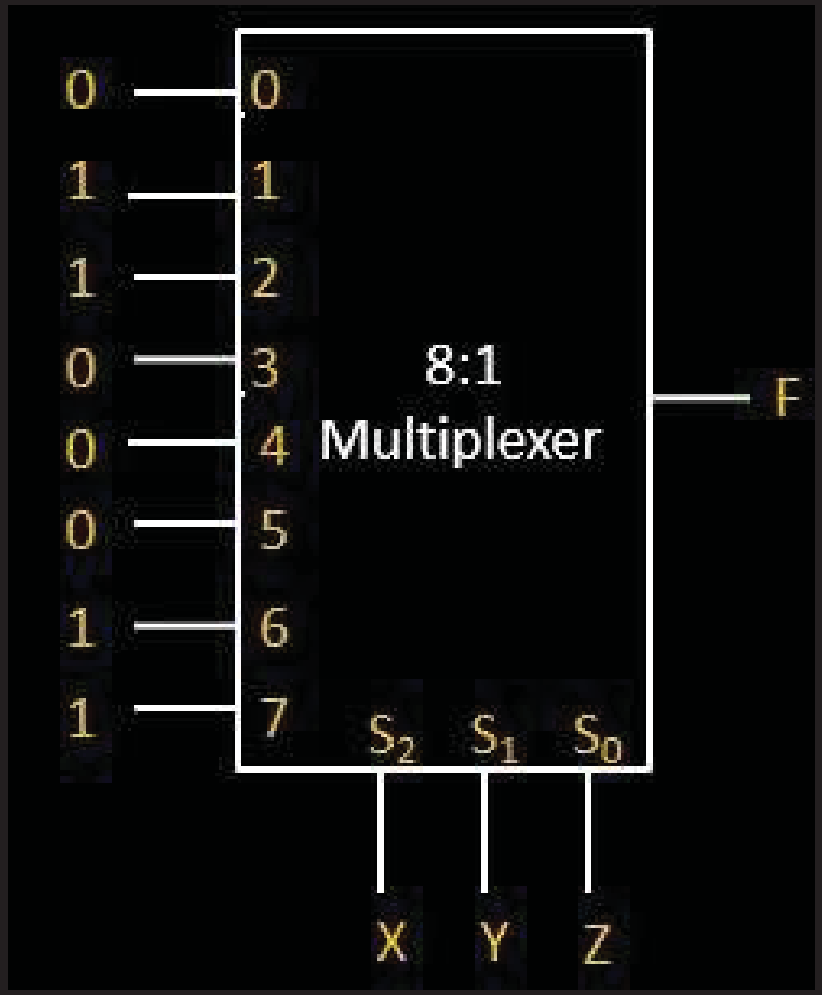


# Multiplexers

## Boolean Function implementation

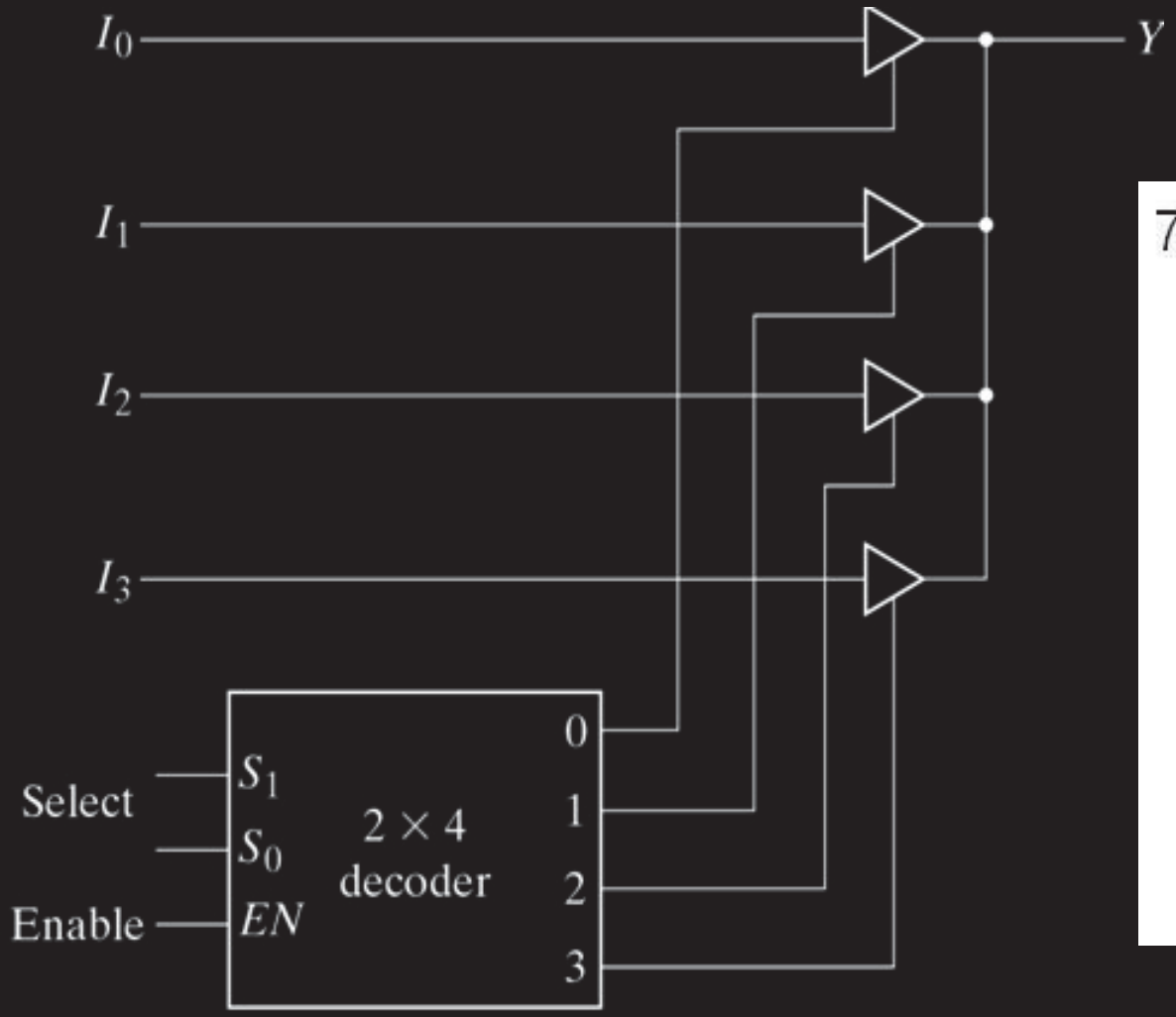
$F(X,Y,Z) = \sum (1, 2, 6, 7)$

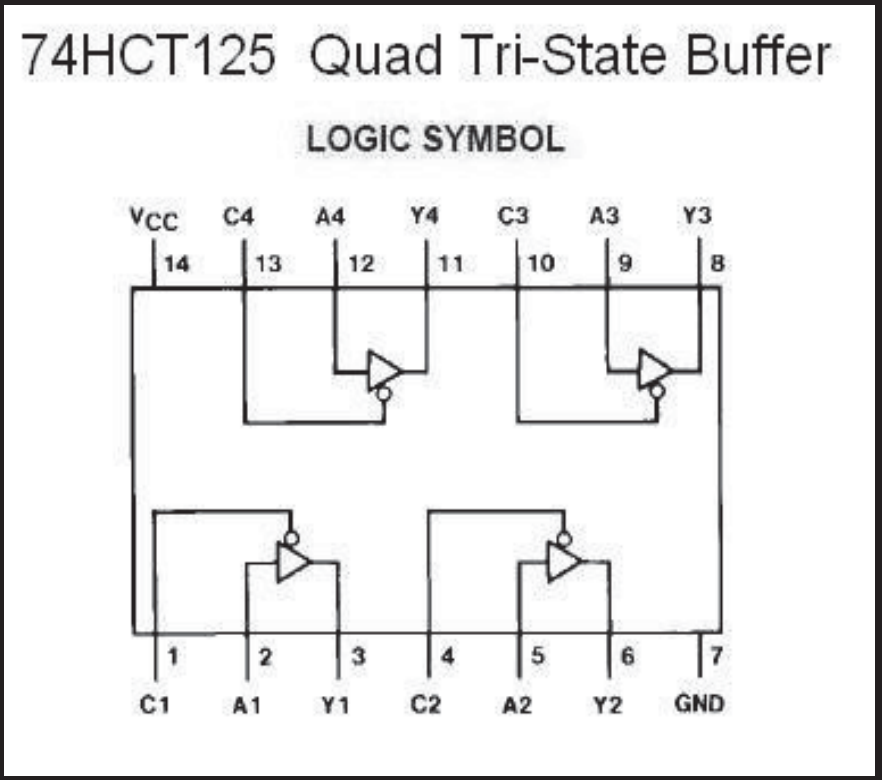
| X | Y | Z | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



# Multiplexers

## 4:1 MUX using Tri-state buffers and 2 to 4 Decoder

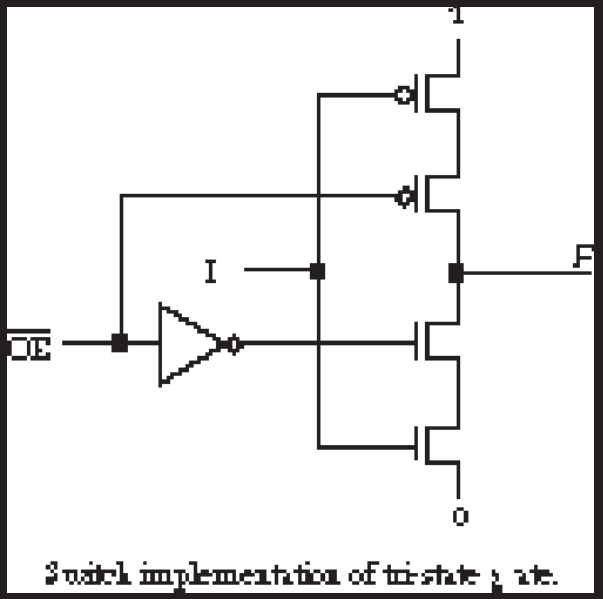


# Multiplexers

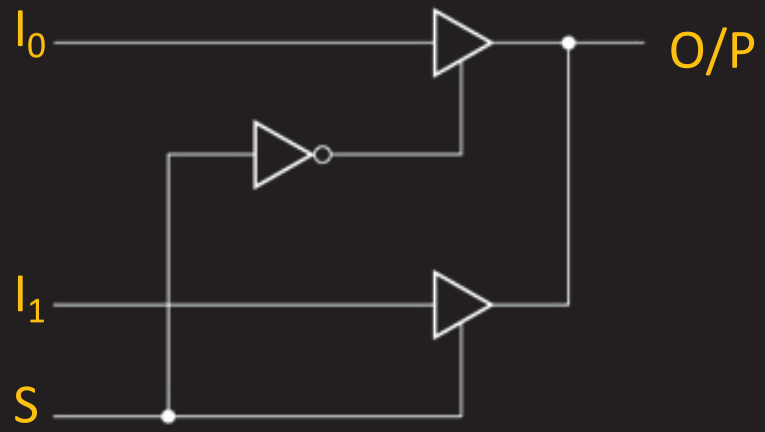
Multiplexers can be constructed using 3-state gates

## Tri-State Buffer

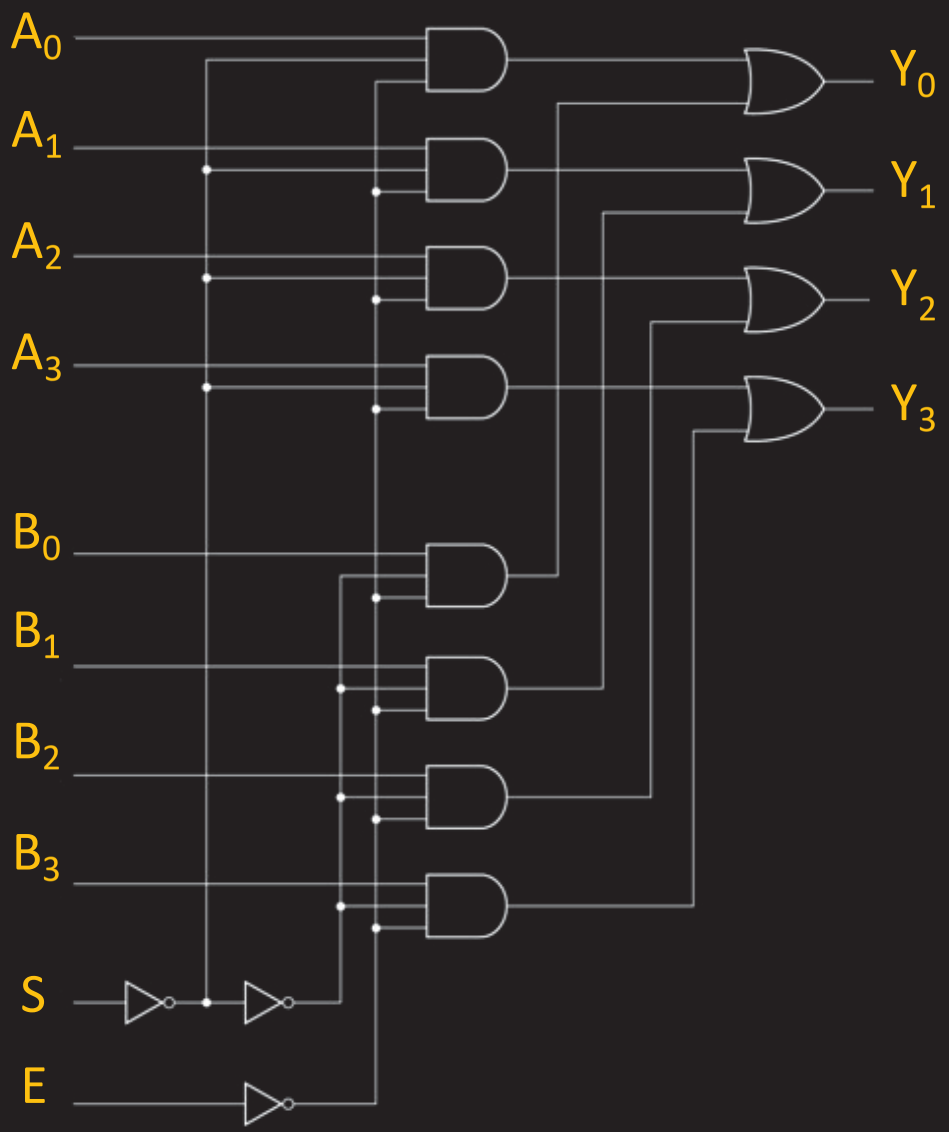



Switch implementation of tri-state gate.

## 2:1 MUX using Tri-state buffer



# Multiplexers




Multiple Bit Selection Logic

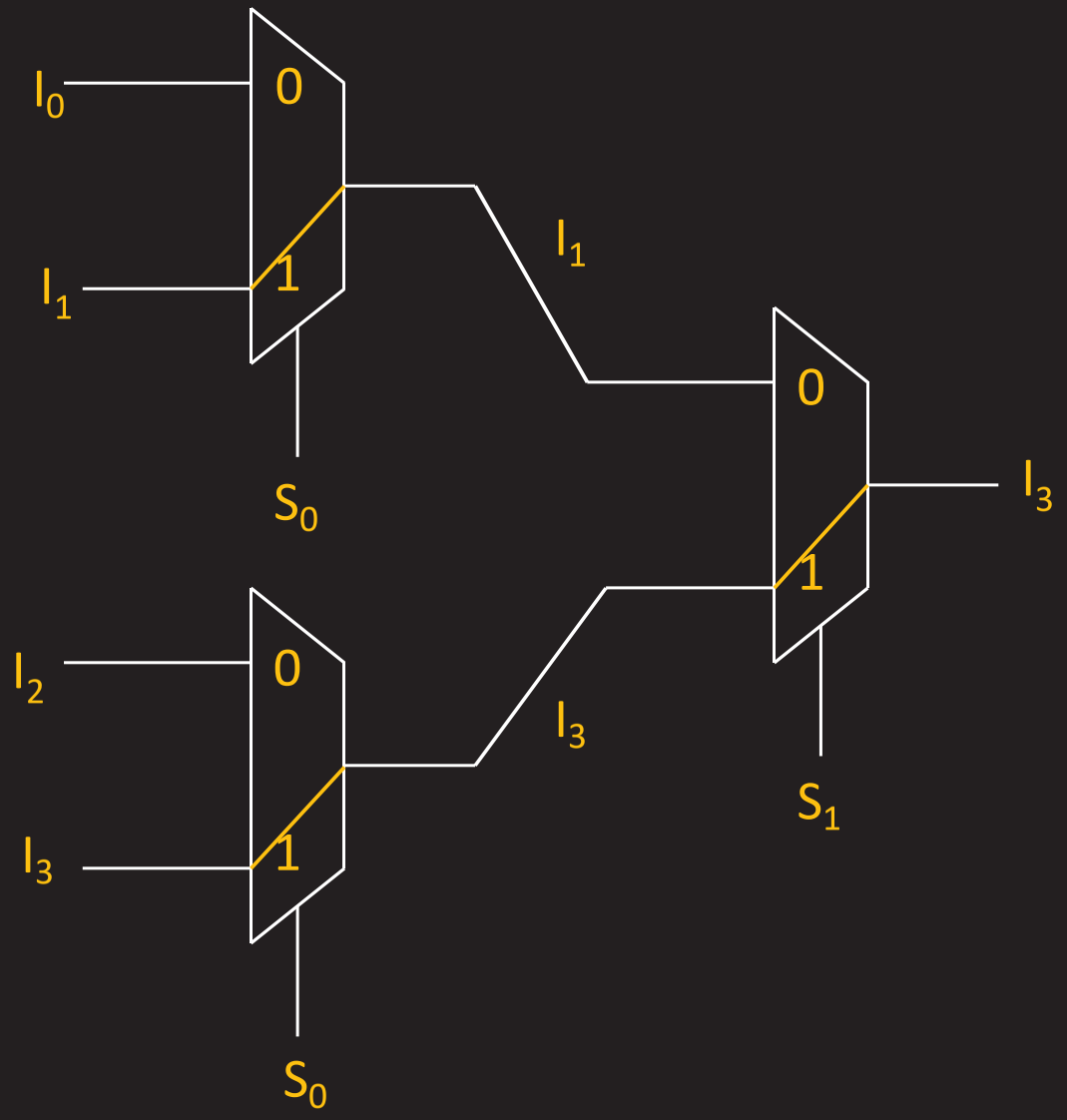
| E | S | Output Y |
|---|---|---|
| 1 | 0 | |
| 1 | 1 | |
| 0 | 0 | |
| 0 | 1 | |

Analyze and Complete the table

# Multiplexers

## 4:1 Multiplexer – 4:1 MUX Using 2:1 MUX

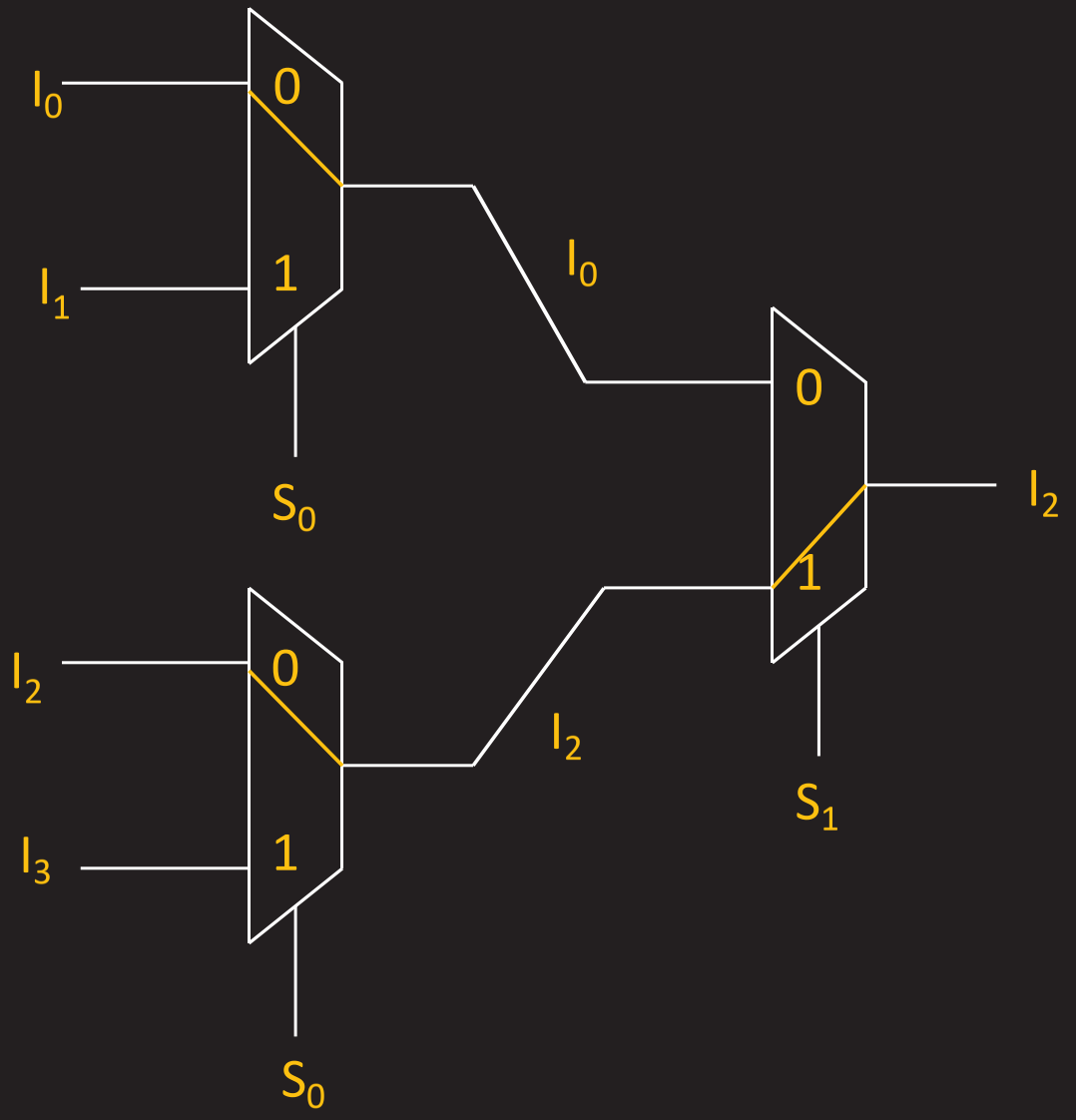
| $S_1$ | $S_0$ | O/P |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |



# Multiplexers

## 4:1 Multiplexer – 4:1 MUX Using 2:1 MUX

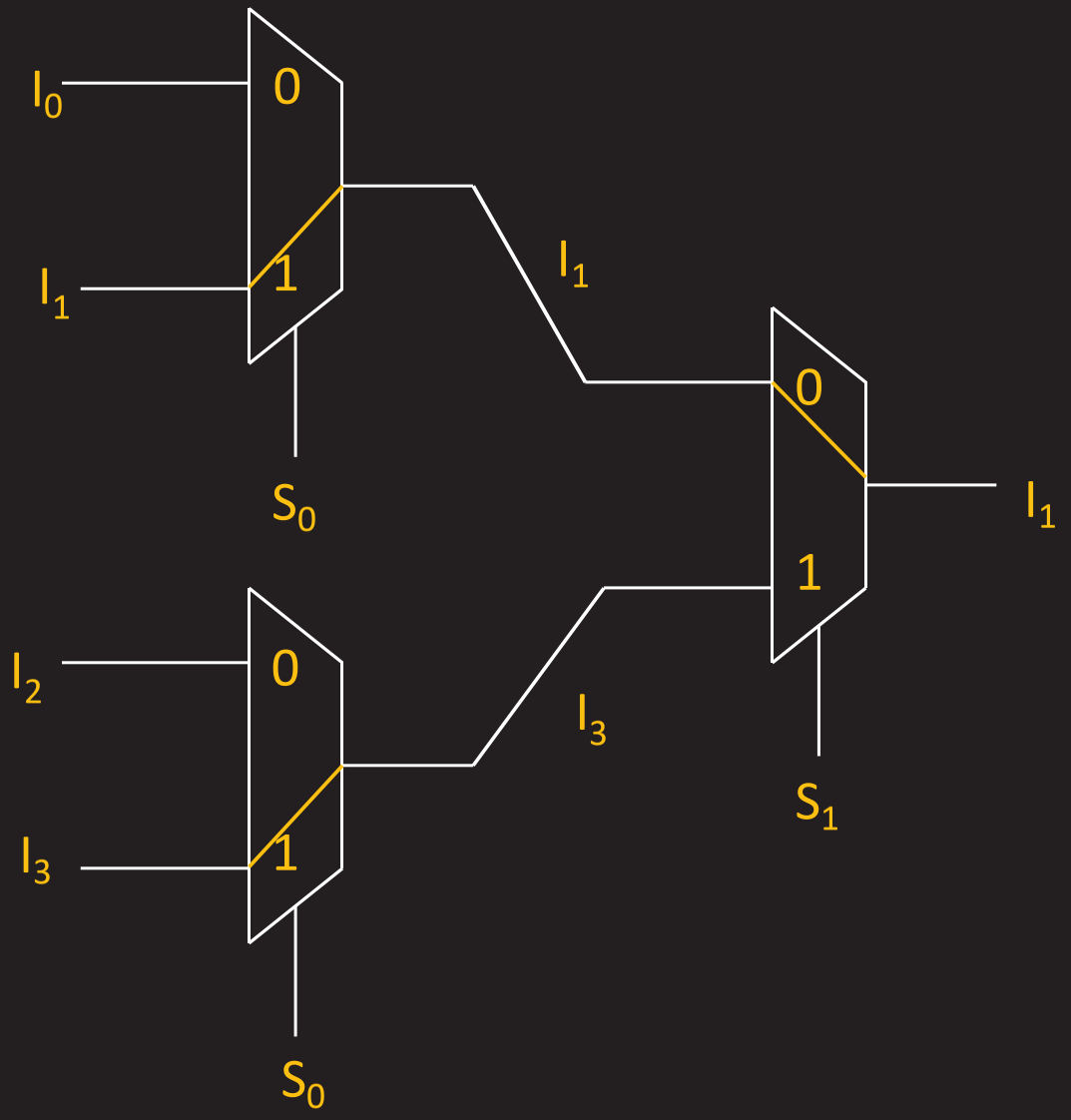
| $S_1$ | $S_0$ | O/P |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |



# Multiplexers

## 4:1 Multiplexer – 4:1 MUX Using 2:1 MUX

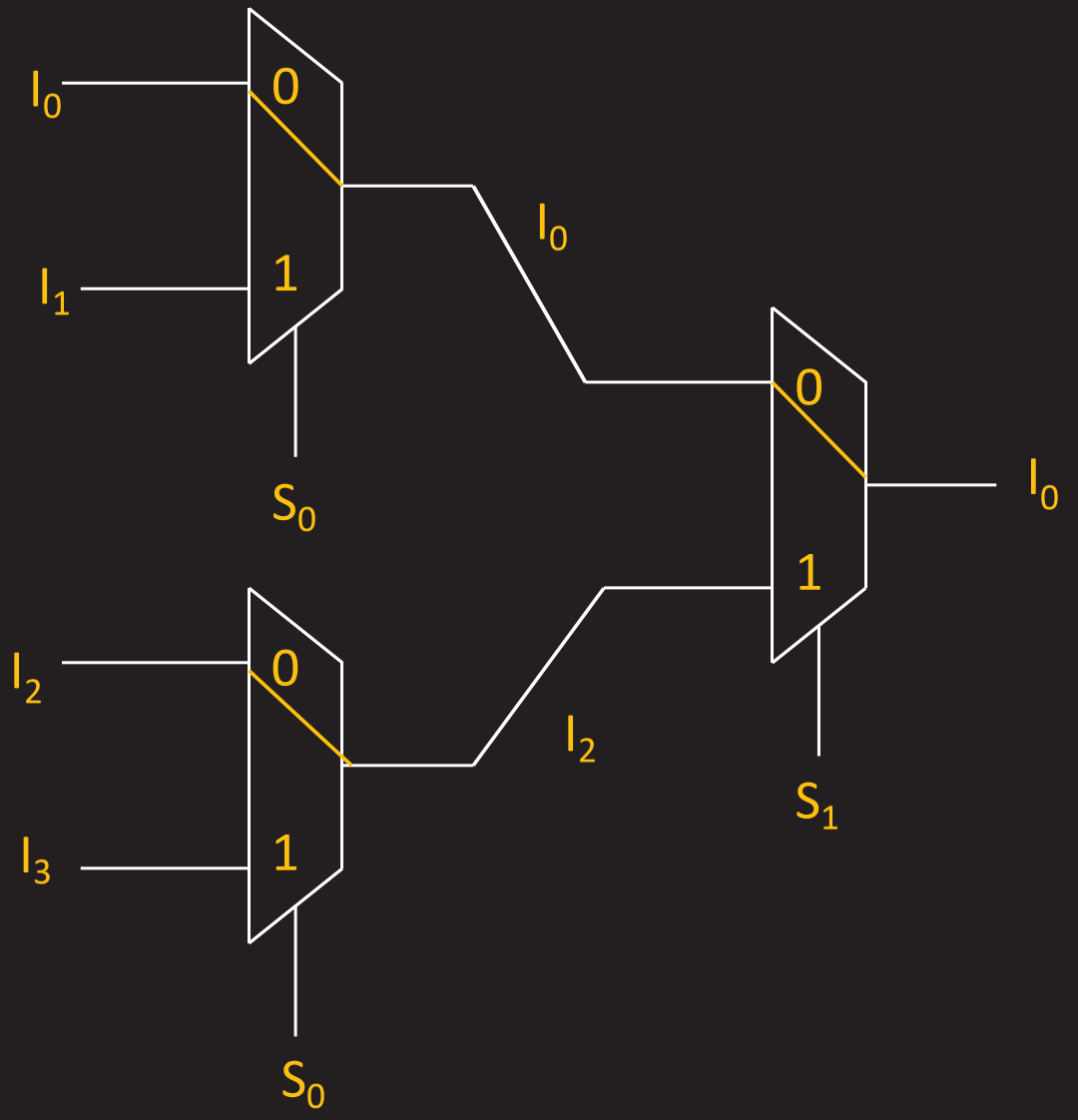
| $S_1$ | $S_0$ | O/P |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |



# Multiplexers

## 4:1 Multiplexer – 4:1 MUX Using 2:1 MUX

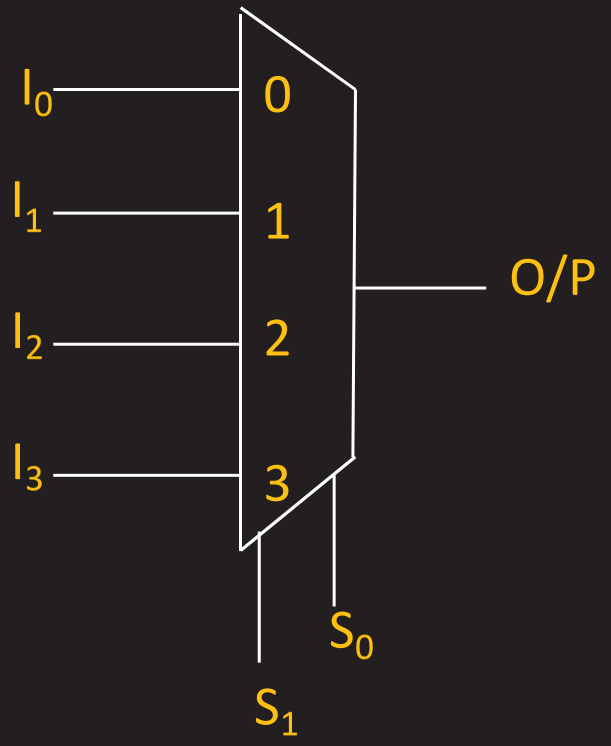
| $S_1$ | $S_0$ | O/P |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |



# Multiplexers

## 4:1 Multiplexer – 4:1 MUX

| $S_1$ | $S_0$ | O/P |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |




$O/P = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$

Implement the operation using suitable logic circuit

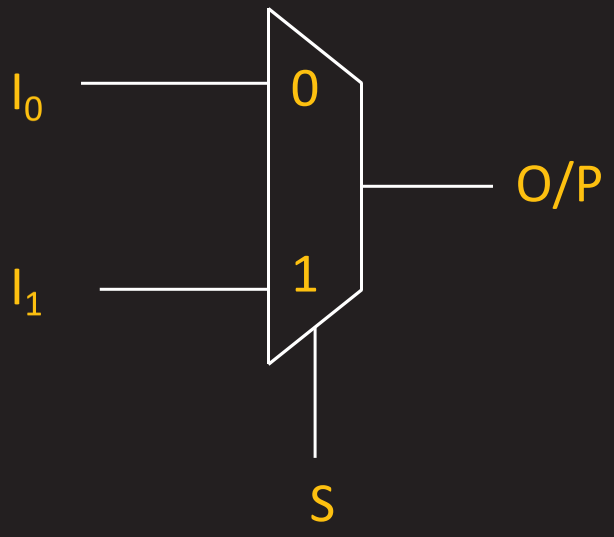
# Multiplexers

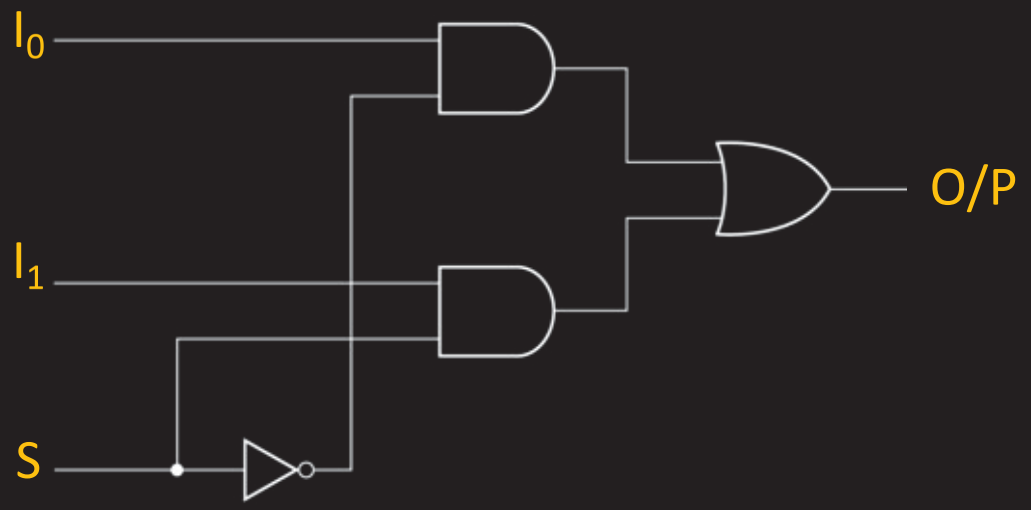
## 2:1 Multiplexer – 2:1 MUX

$I_0$ and $I_1$ are input lines and 'S' is select line

If s=0 then o/p = $I_0$
s=1 then o/p = $I_1$

$o/p = I_0.S' + I_1.S$



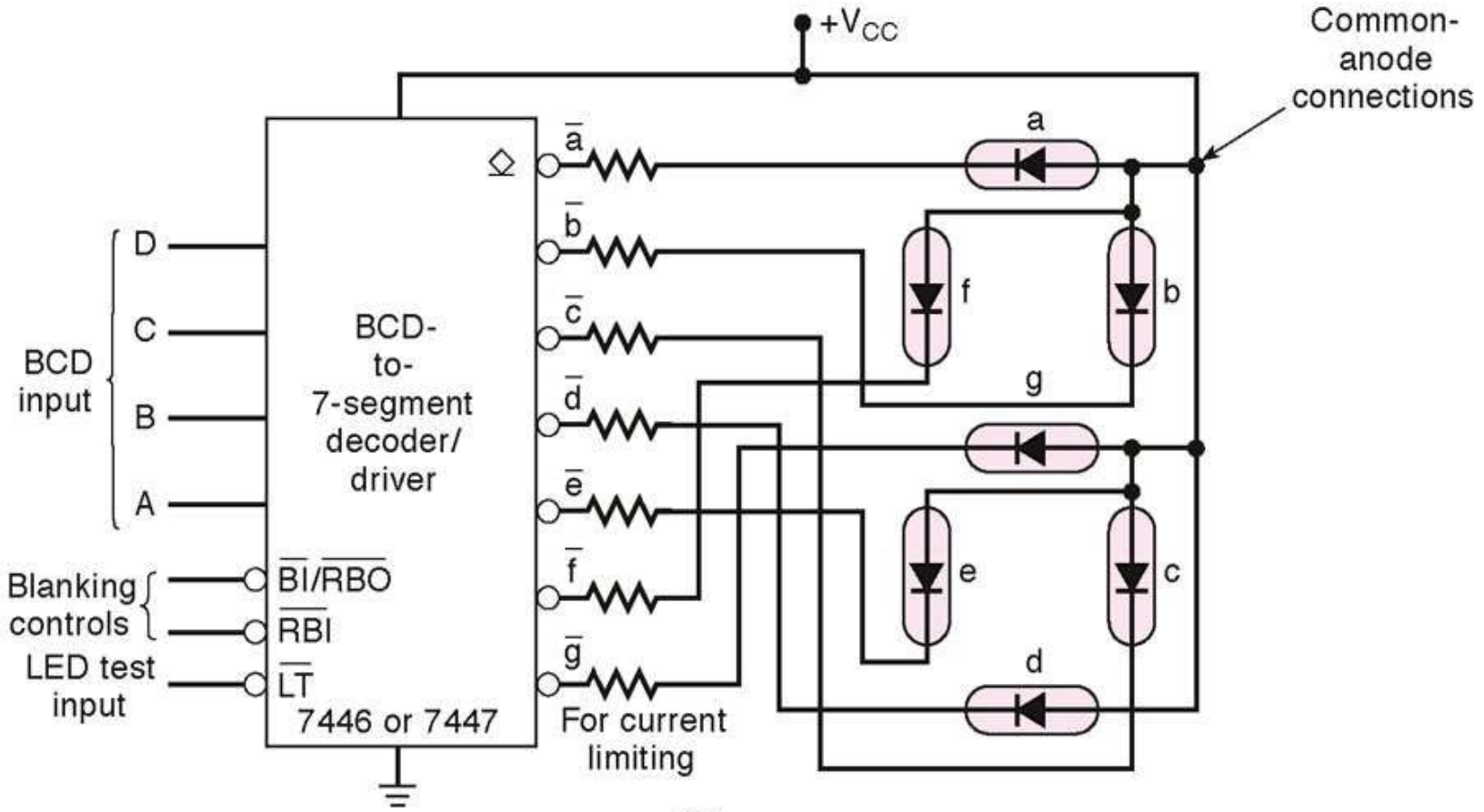



# Multiplexers

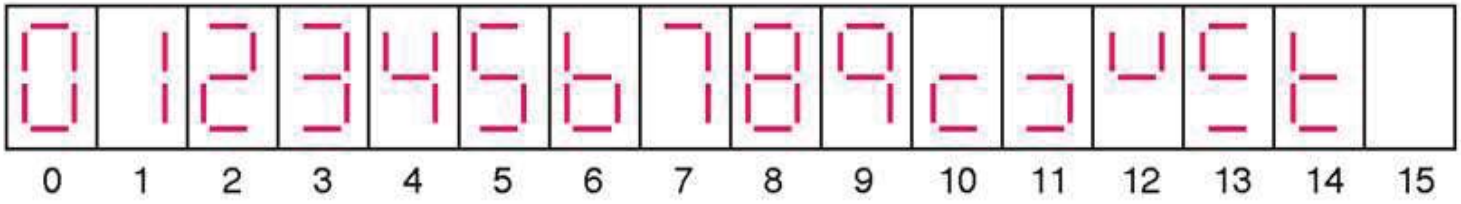
Multiplexer - combinational circuit that selects binary information from one of many input lines and directs it to single output line

The selection is controlled by Select lines

In general, $2^n$ input lines, 'n' selection lines and 1 output


+VCC
Common-anode connections
a
b
c
d
e
f
g
BCD input
D
C
B
A
BCD-to-7-segment decoder/driver
Blanking controls
BI/RBO
RBI
LED test input
LT
7446 or 7447
For current limiting

(a)

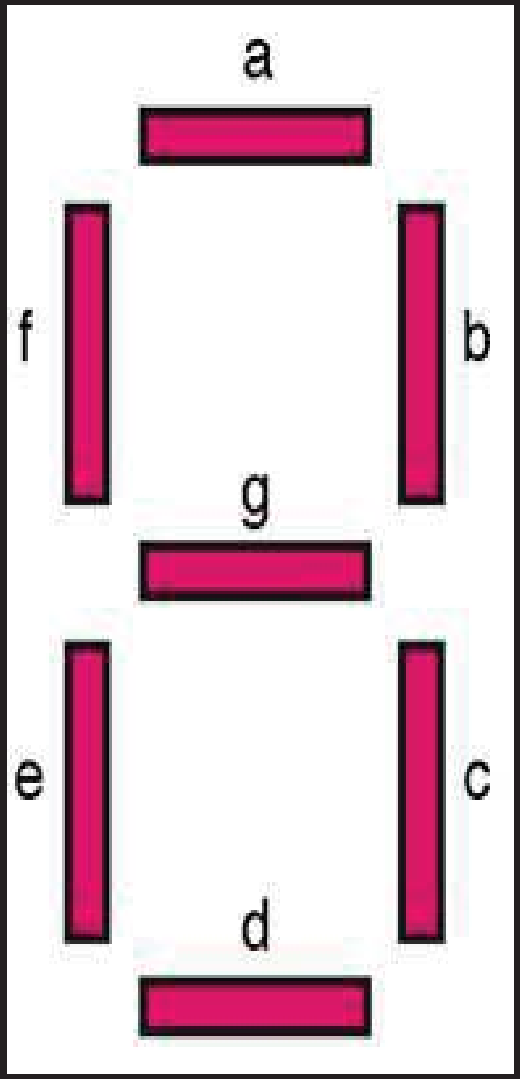

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

(b)

# BCD-7segment display using decoders

Most digital equipment has some means for displaying information in a form that can be understood by the user.

One of the simplest and most popular methods for displaying numerical digits uses a 7-segment configuration to form digital characters 0 to 9 and some times the hex characters A to F

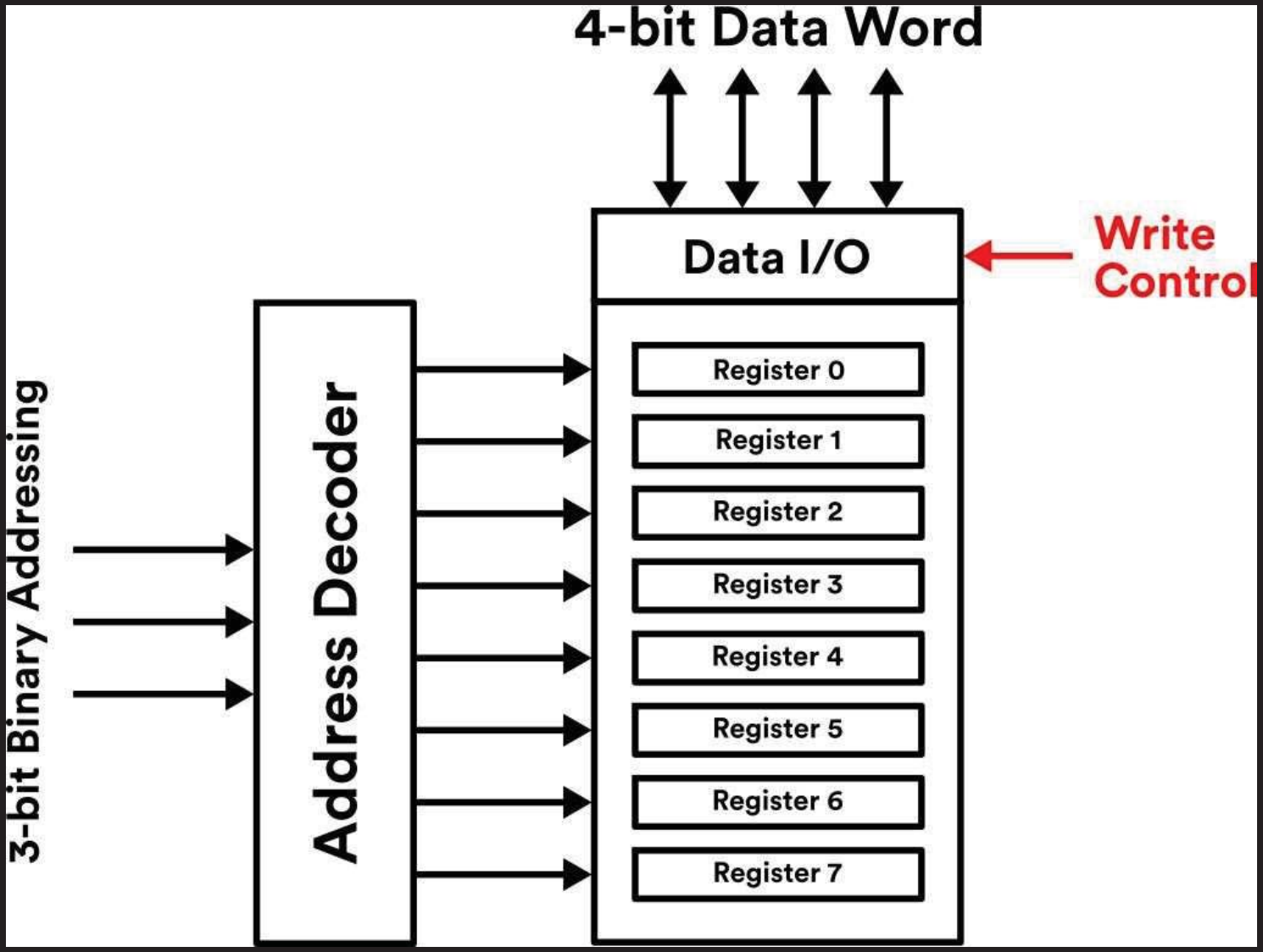


# Digital Design

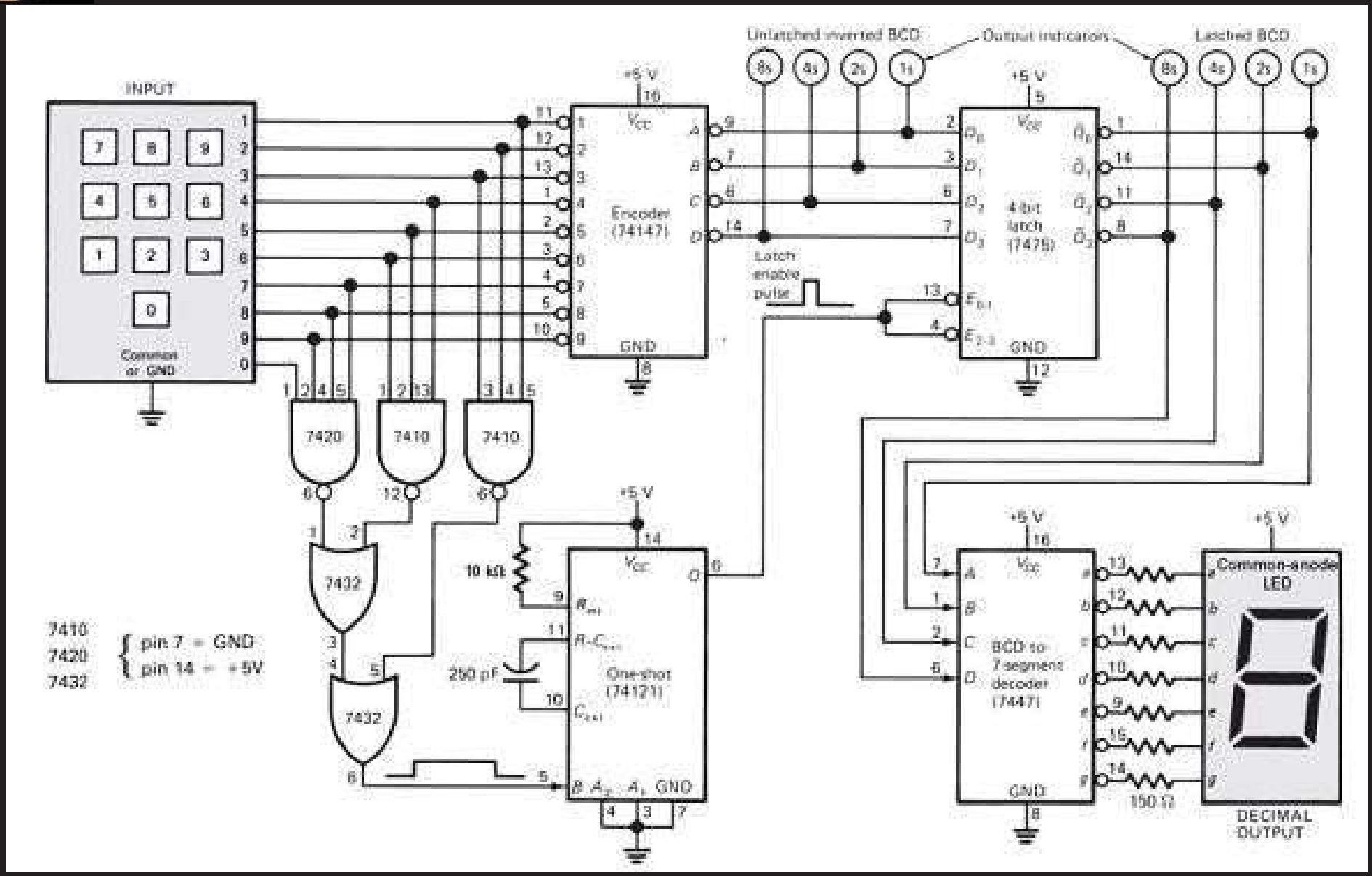
Lecture 14: Multiplexer


# Thank You

# Decoder Applications

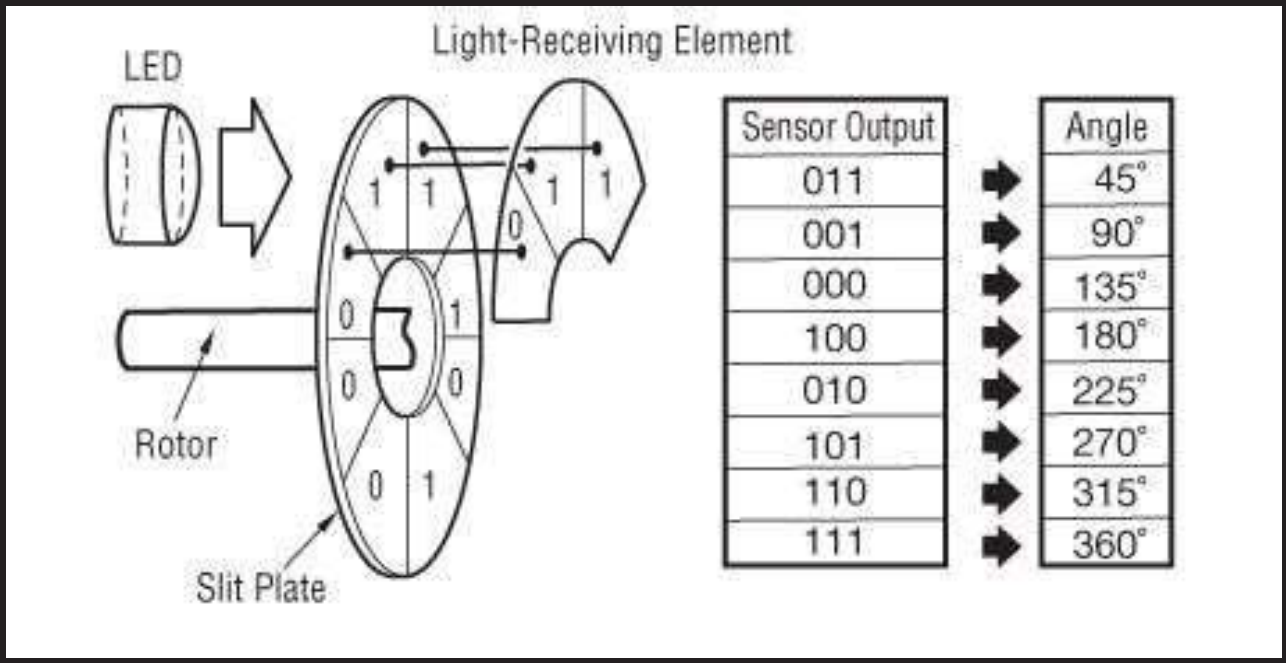


# Encoder Applications



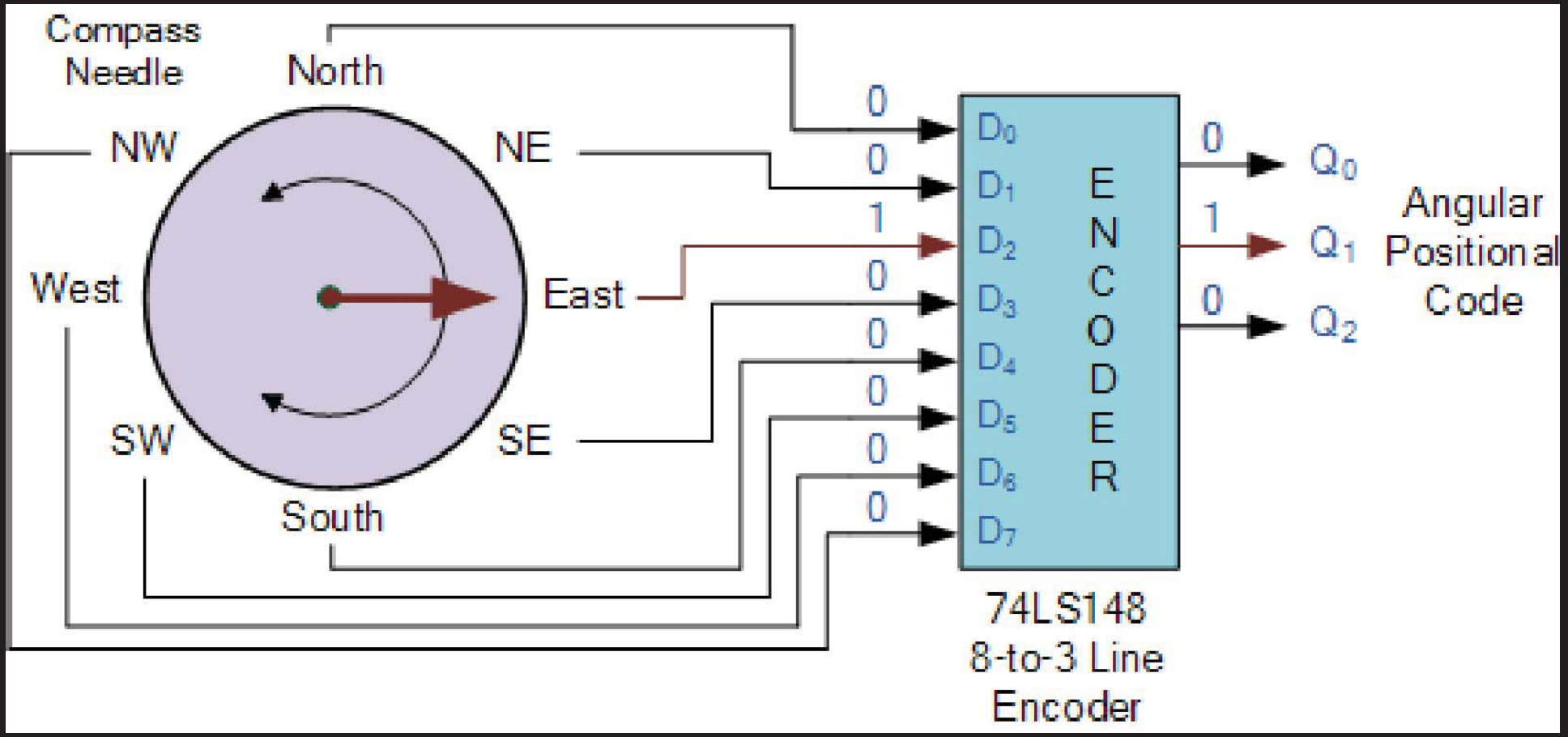
# Encoder Applications :

## Optical Encoder for motor position control

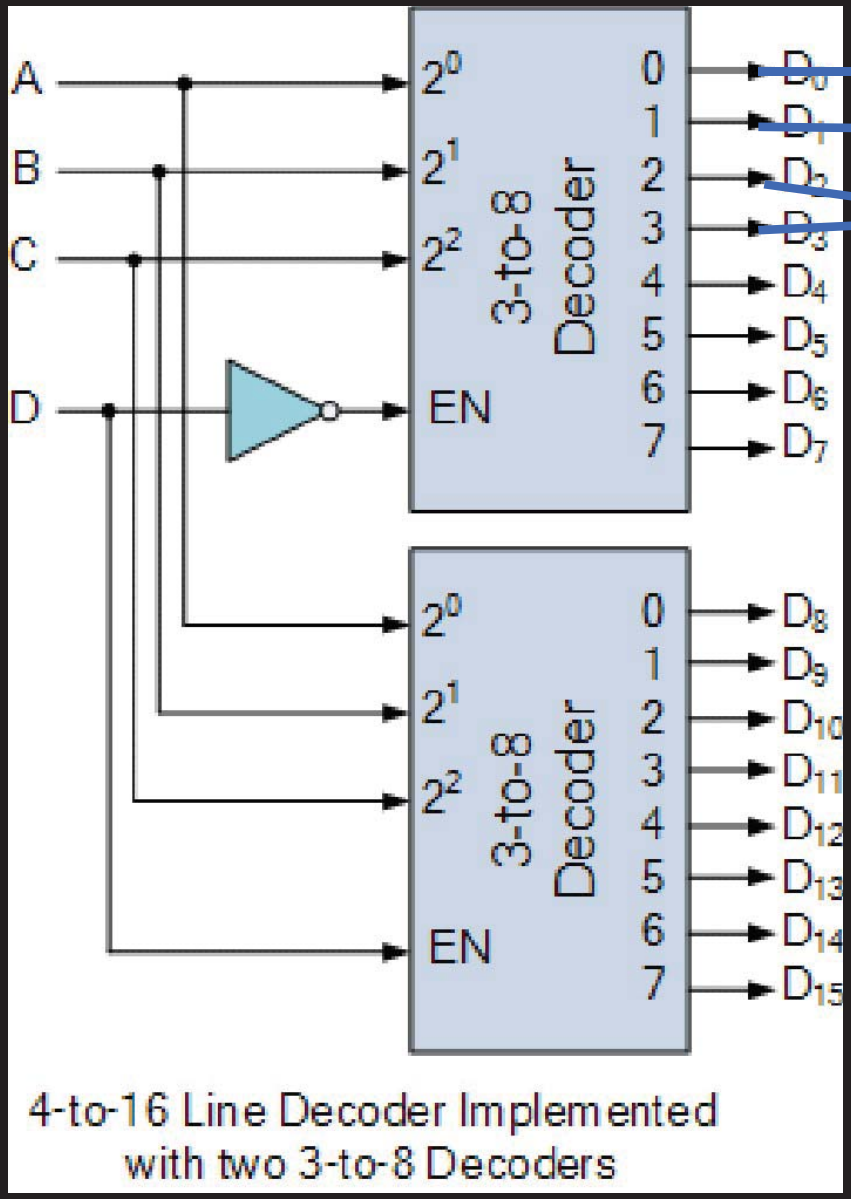



| Sensor Output | Angle |
|---|---|
| 011 | 45° |
| 001 | 90° |
| 000 | 135° |
| 100 | 180° |
| 010 | 225° |
| 101 | 270° |
| 110 | 315° |
| 111 | 360° |

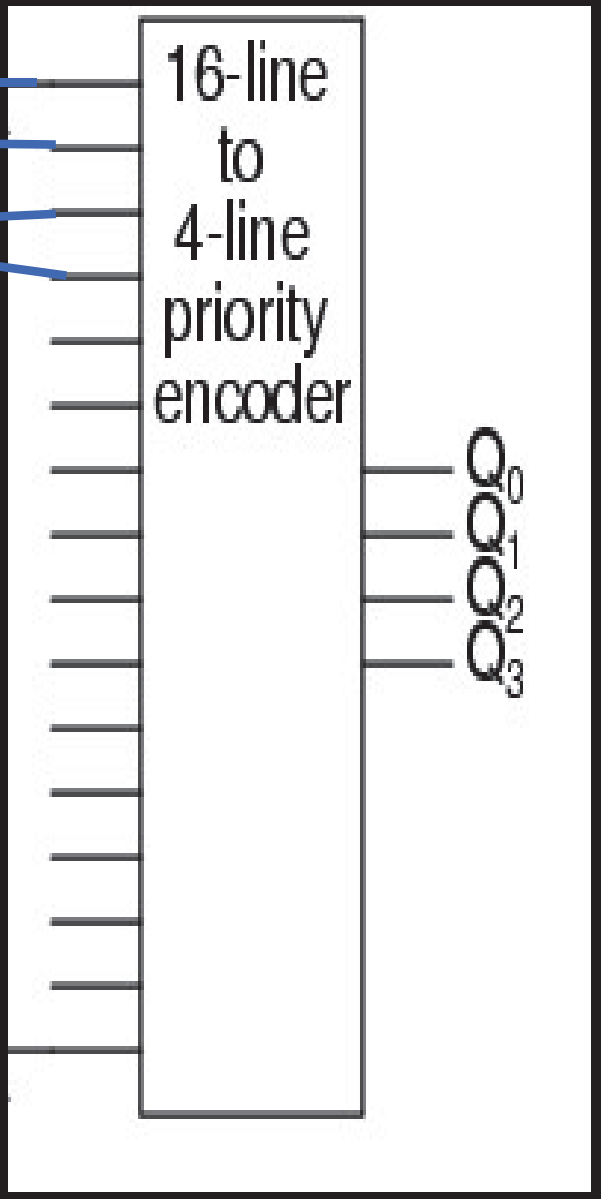
# Encoder Applications



# Binary to Gray Using Encoder-Decoder




4-to-16 Line Decoder Implemented with two 3-to-8 Decoders



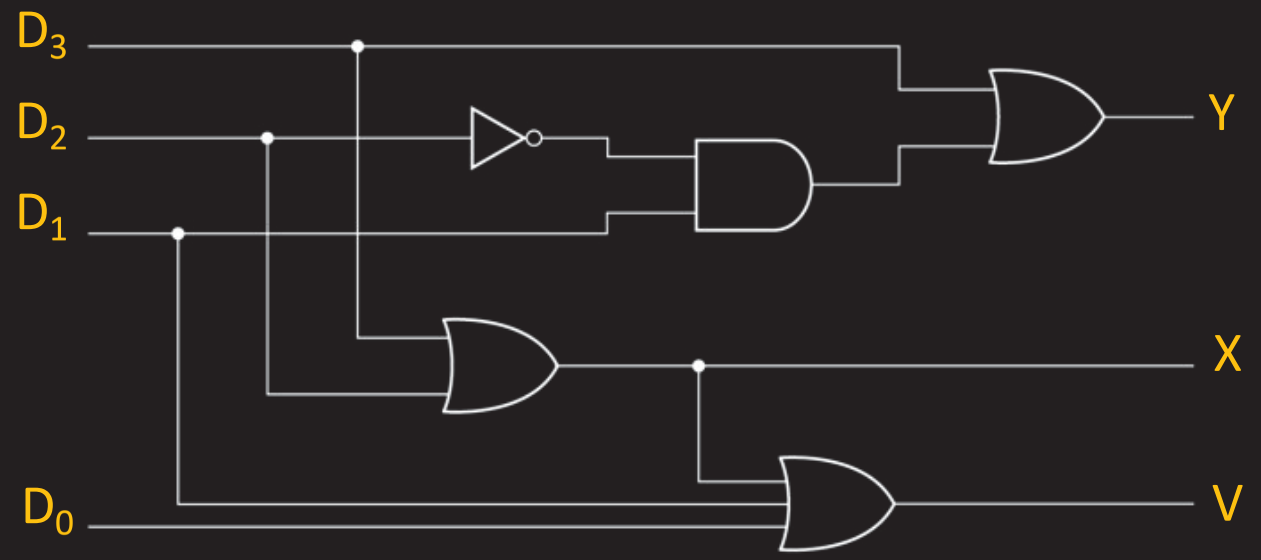
# Priority Encoder

## Priority Encoder

$X = D_2 + D_3$

$Y = D_1D_2' + D_3$

$V = D_0 + D_1 + D_2 + D_3$



# Priority Encoder

| $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

| X | Y |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

## Encoder

$X = D_2 + D_3$

$Y = D_1 + D_3$

| $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| X | 1 | 0 | 0 |
| X | X | 1 | 0 |
| X | X | X | 1 |
| 0 | 0 | 0 | 0 |

| X | Y | V |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |
| X | X | 0 |

## Priority Encoder

$X = D_2 + D_3$

$Y = D_1D_2' + D_3$

$V = D_0 + D_1 + D_2 + D_3$

# Priority Encoder

If all the inputs are zeros then output is 00 – which indicates that $D_0$ is 1

This ambiguity can be resolved by providing one more output.

Extra Output indicates whether at least one input is equal to 1

# Priority Encoder

If two ones appear i.e. if $D_1$ and $D_2$ are '1' then both X and Y will be '1' resulting in output 11

Encoder circuits must establish input priority

E.g. If high priority is given to inputs with higher subscripts

If both $D_1$ and $D_2$ are '1' then resulting o/p should be 10

# Encoder

$X = D_2 + D_3$

$Y = D_1 + D_3$

| $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

| X | Y |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

## Limitations:

If two ones appear i.e. if $D_1$ and $D_2$ are '1' then both X and Y will be '1' resulting in output 11

If all the inputs are zeros then output is 00 – which indicates that $D_0$ is 1

# Encoder

Encoder - combinational circuit that performs inverse operation of decoder

Has $2^n$ or fewer input lines and n output lines

An Example:

$X = D_2 + D_3$

$Y = D_1 + D_3$

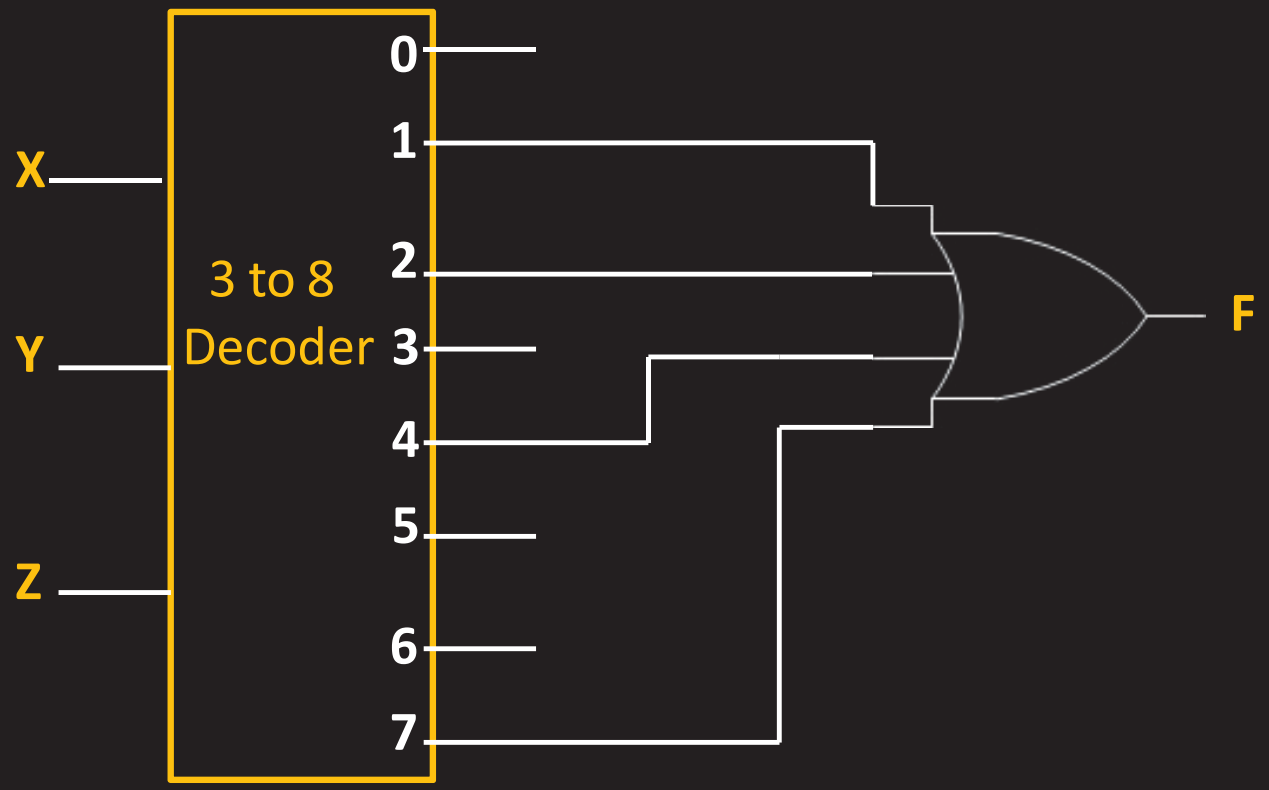
| $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

| X | Y |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

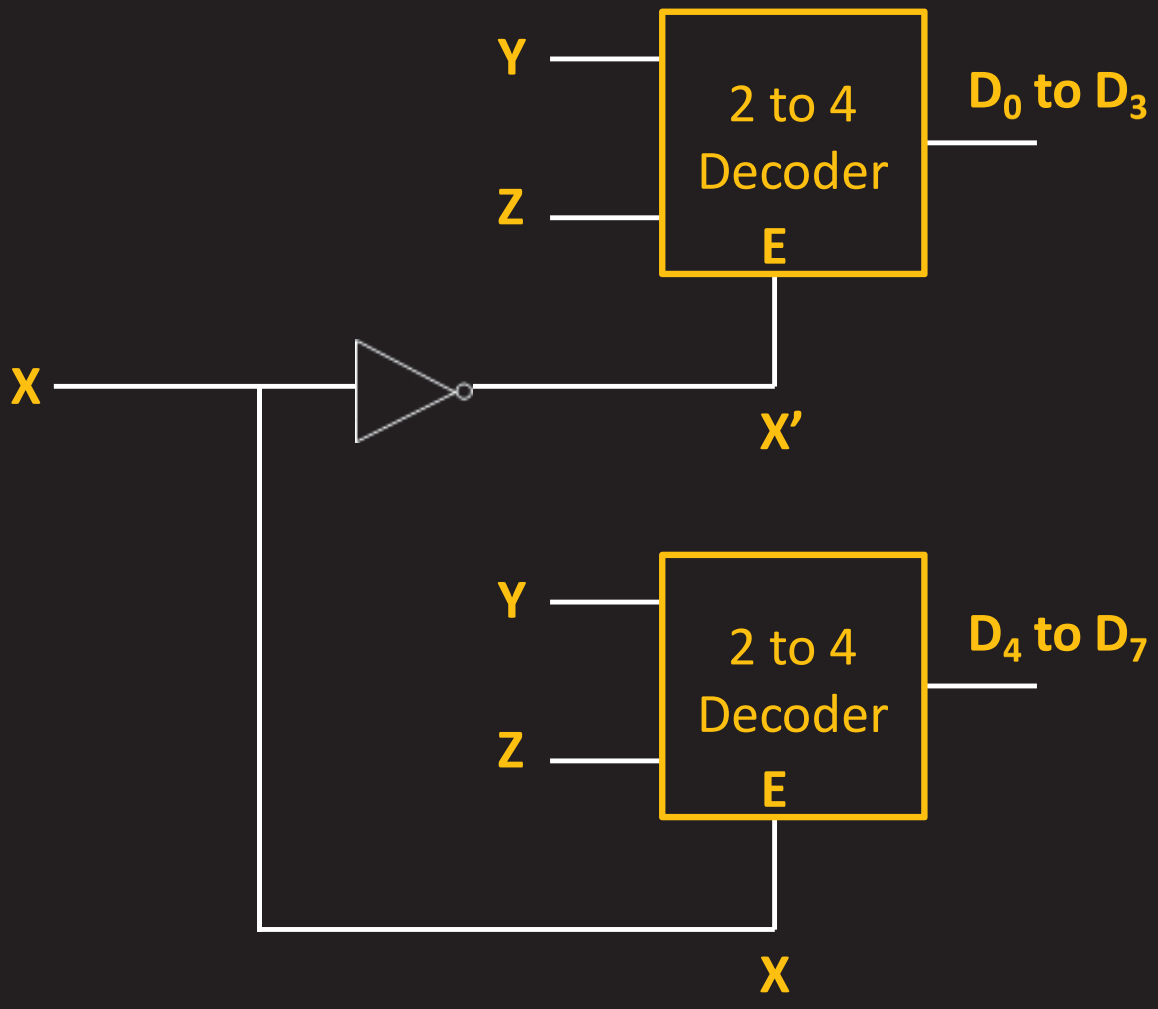
# Decoder

## Implementing function using Decoders

$F = \sum (1, 2, 4, 7) = X'Y'Z + X'YZ' + XY'Z' + XYZ$



# 3 to 8 Decoder using 2nos of 2 to 4 Decoder

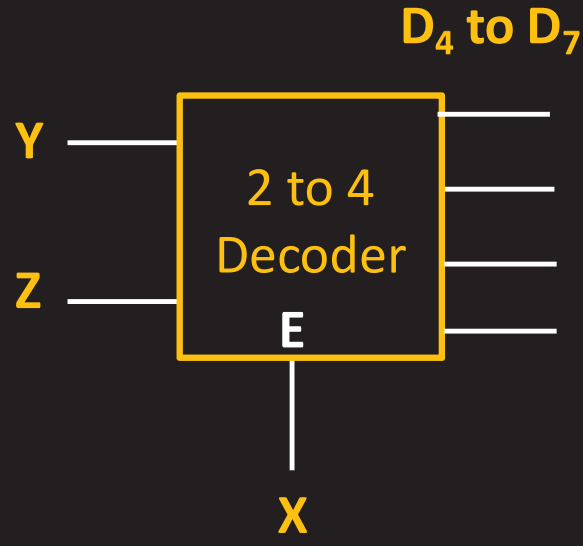


# Decoder

Can implement a 3 to 8 decoder using two 2 to 4 decoders with enable pin

| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

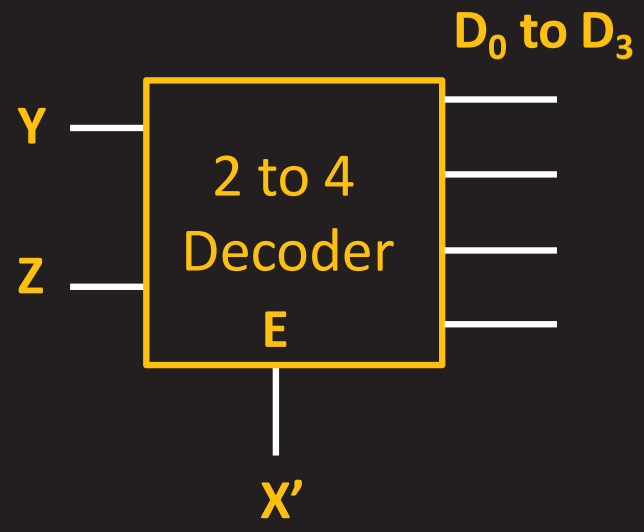
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |



# Decoder

Can you implement a 3 to 8 decoder using two 2 to 4 decoders with enable pin

| X | Y | Z | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |



# Decoder

## An Example: 3 to 8 Line Decoder

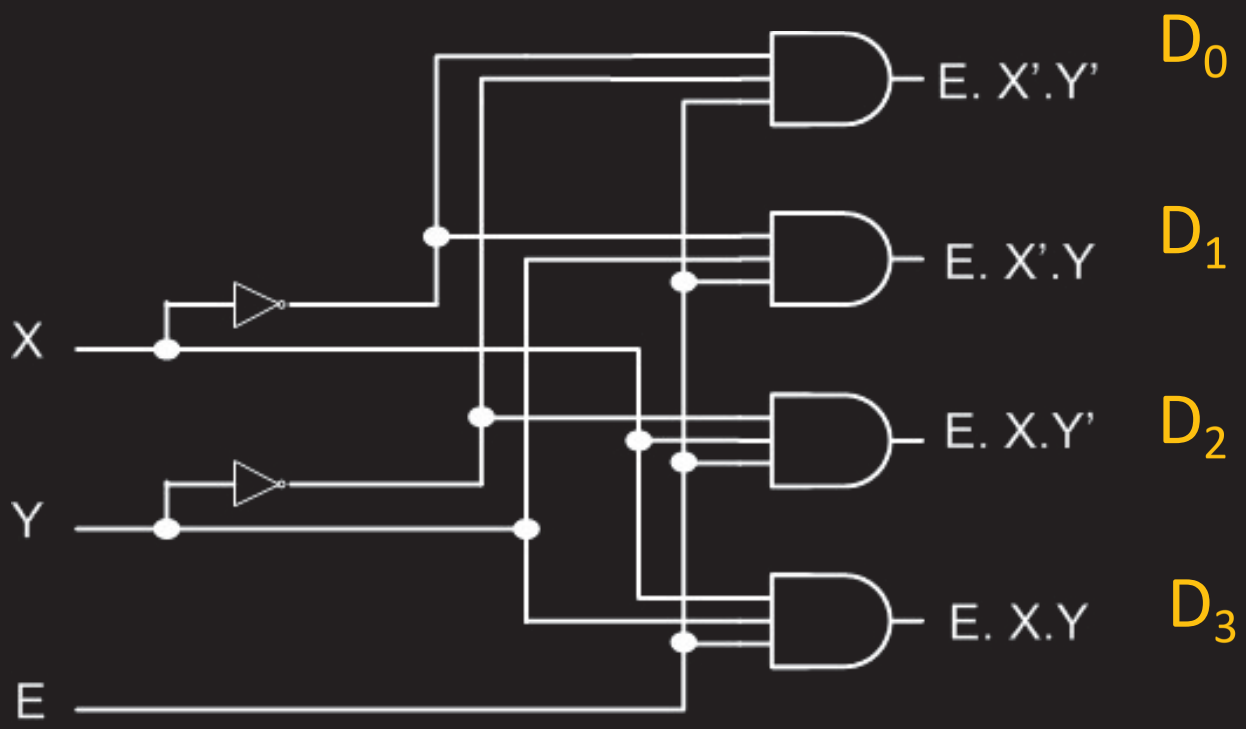
| X | Y | Z |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Direct Implementation – Home Assignment

# Decoder

An Example: 2 to 4 Line Decoder with Enable



# Decoder

## An Example: 2 to 4 Line Decoder with Enable

| E | X | Y |
|---|---|---|
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | X | X |

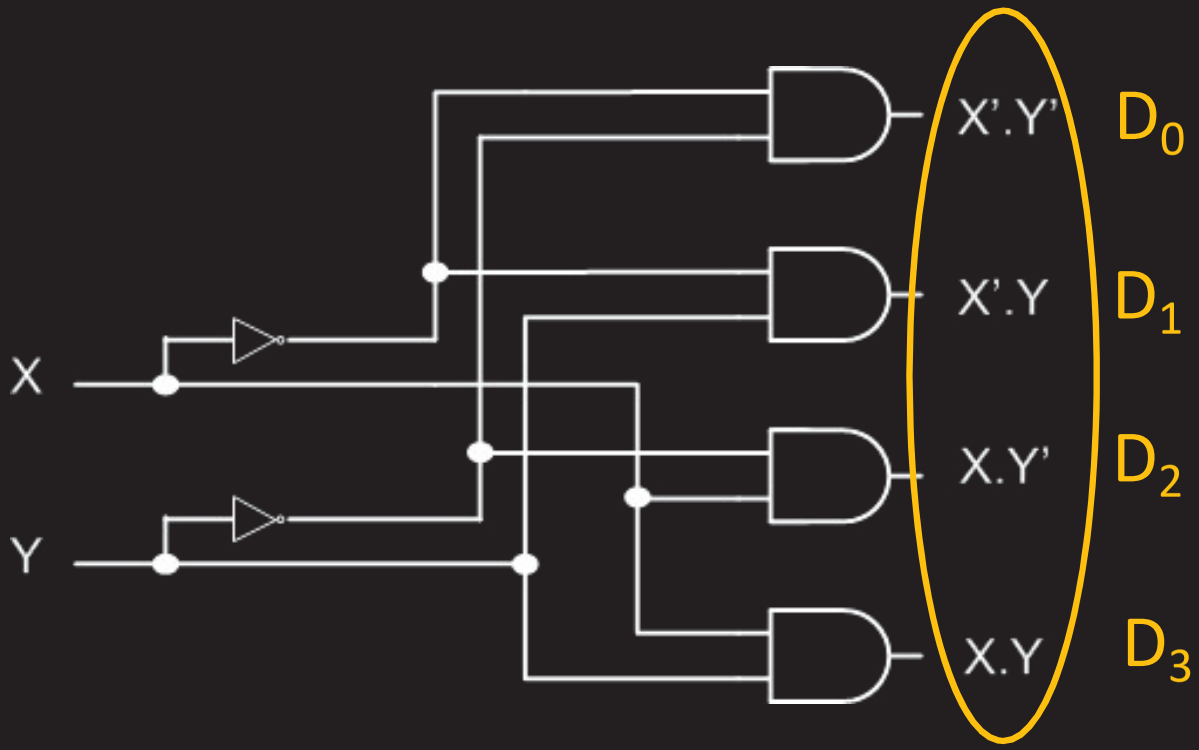
| $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |

Decoder with enable pin is also called as Demultiplexer

X and Y – Select Lines  E- Input line

# Decoder

## An Example: 2 to 4 Line Decoder




Minterms
(0, 1, 2, 3)

Minterm Generating circuit

# Decoder

## An Example: 2 to 4 Line Decoder

| X | Y |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

| $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

# Decoder

A Decoder is a combinational circuit that converts binary information from ‘n’ input lines to $2^n$ output lines

n to m decoders where $m \leq 2^n$

The name decoder is also used in conjunction with other code converters (e.g. BCD to seven segment decoder)

# Digital Design

## Lecture 13: Decoders and Encoders


# Thank You

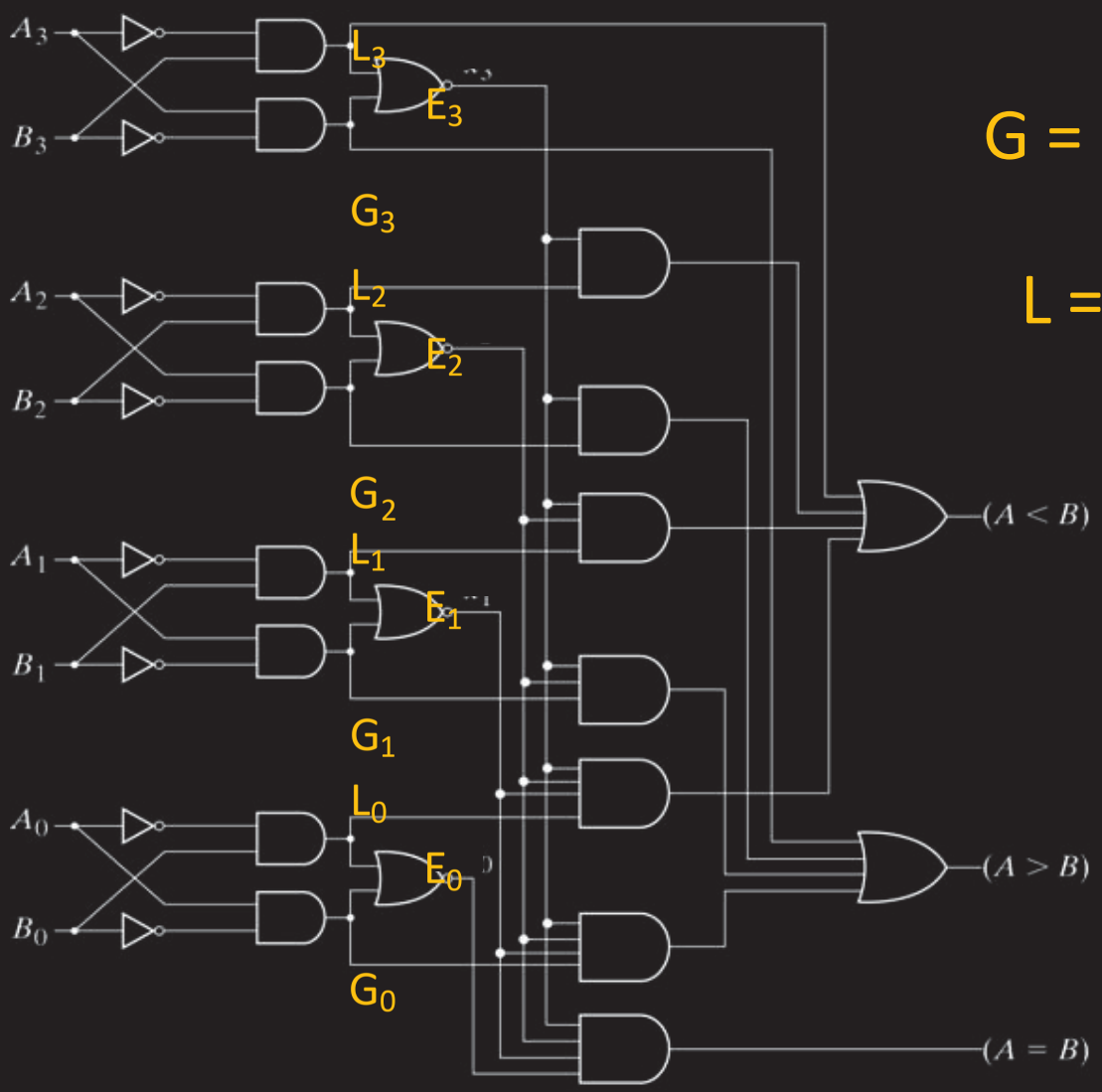
# Next Module

Decoders

Encoders

Multiplexers

# Magnitude Comparator




## 4-Bit Comparator

$$G = G_3 + E_3.G_2 + E_3.E_2.\ G_1 + E_3.E_2.\ E_1.\ G_0$$

$$L = L_3 + E_3.L_2 + E_3.E_2.\ L_1 + E_3.E_2.\ E_1.\ L_0$$

$$E = E_2\ .E_1.\ E_0$$

Can you draw the circuit ??

# Magnitude Comparator

1 Bit Comparator

$G = G_0$ $L = L_0$ $E = E_0$

2 Bit Comparator

$G = G_1 + E_1 . G_0$ $L = L_1 + E_1 . L_0$ $E = E_1 . E_0$

3 Bit Comparator

$G = G_2 + E_2 . G_1 + E_2 . E_1 . G_0$

$L = L_2 + E_2 . L_1 + E_2 . E_1 . L_0$

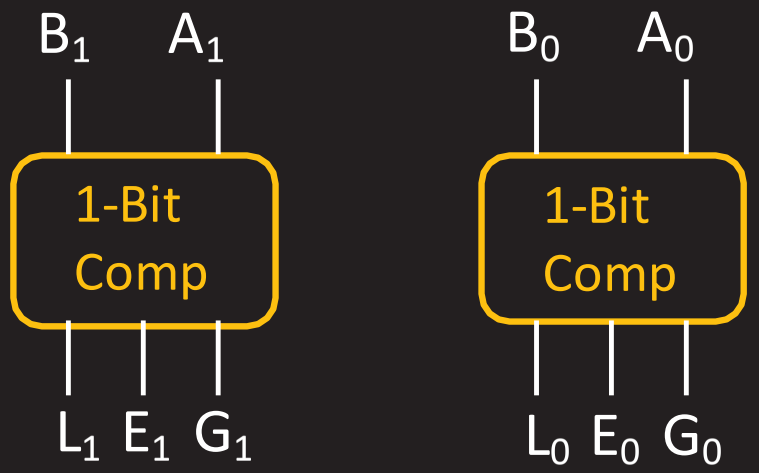
$E = E_2 . E_1 . E_0$

# Magnitude Comparator

## 2-Bit Comparator

A $A_1 A_0$

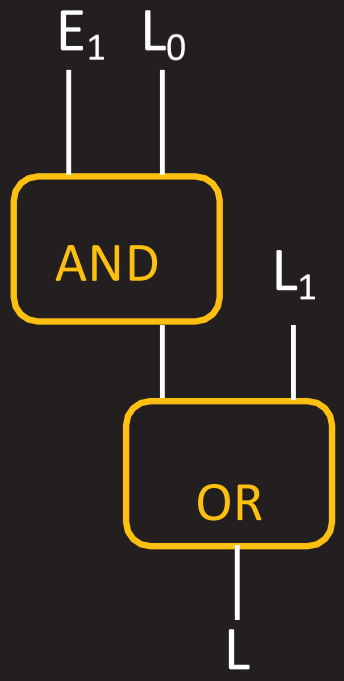
B $B_1 B_0$




A < B When $A_1 < B_1$ OR

When $A_1 = B_1$ and $A_0 < B_0$

$$L = L_1 + E_1 . L_0$$

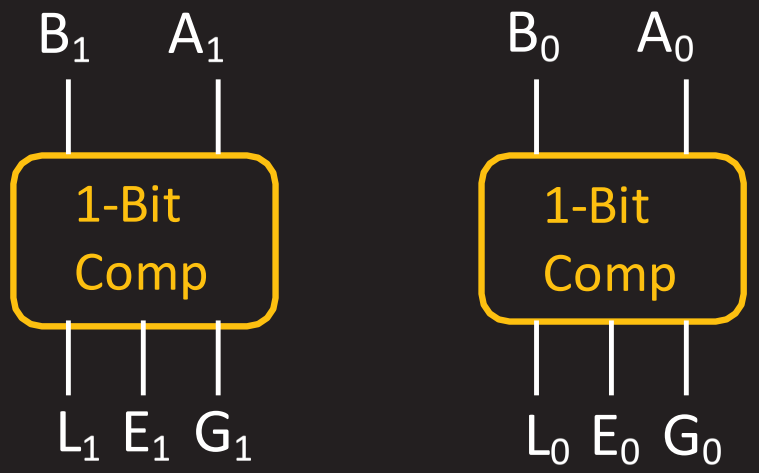


# Magnitude Comparator

## 2-Bit Comparator

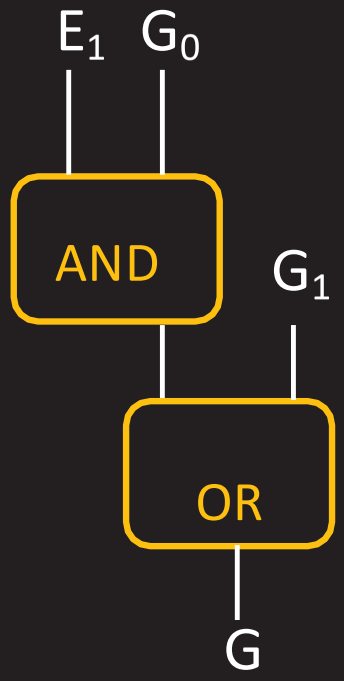
A $A_1 A_0$

B $B_1 B_0$




A > B When $A_1 > B_1$ OR
When $A_1 = B_1$ and $A_0 > B_0$

$G = G_1 + E_1 . G_0$



# Magnitude Comparator

## 2-Bit Comparator

A $A_1 A_0$

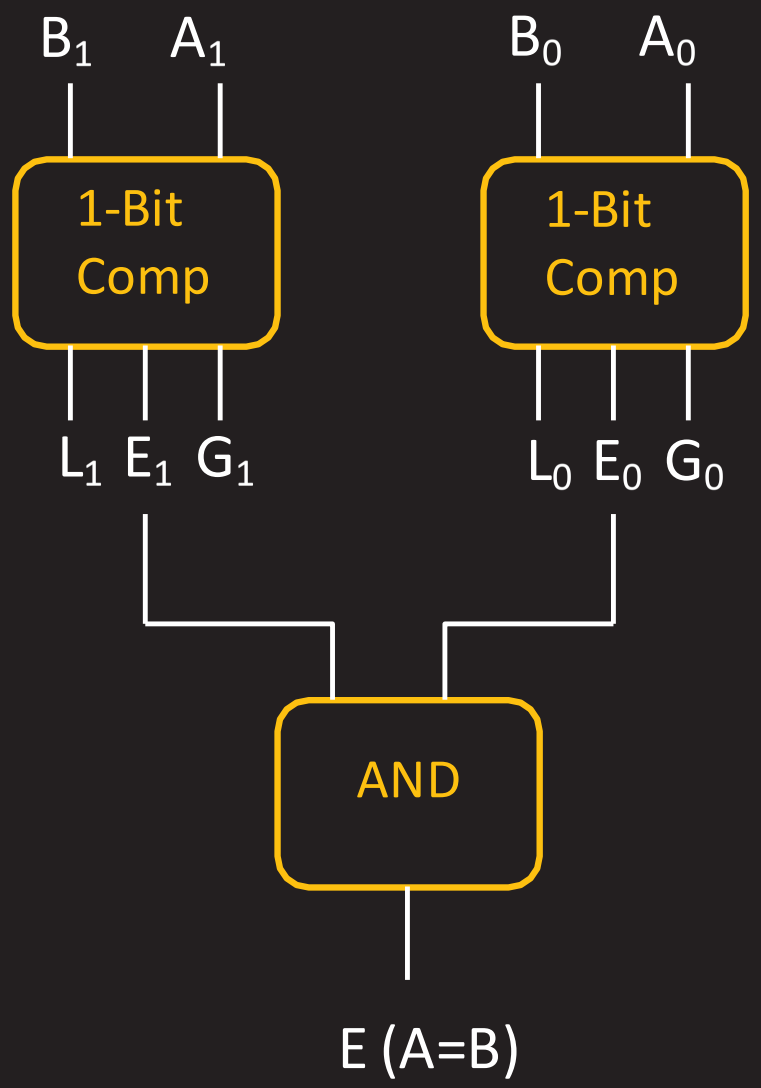
B $B_1 B_0$

# Magnitude Comparator

## 2-Bit Comparator

A $A_1 A_0$

B $B_1 B_0$




A = B When $A_0 = B_0$ and $A_1 = B_1$

$E = E_0 . E_1$

# Magnitude Comparator

1-Bit Comparator

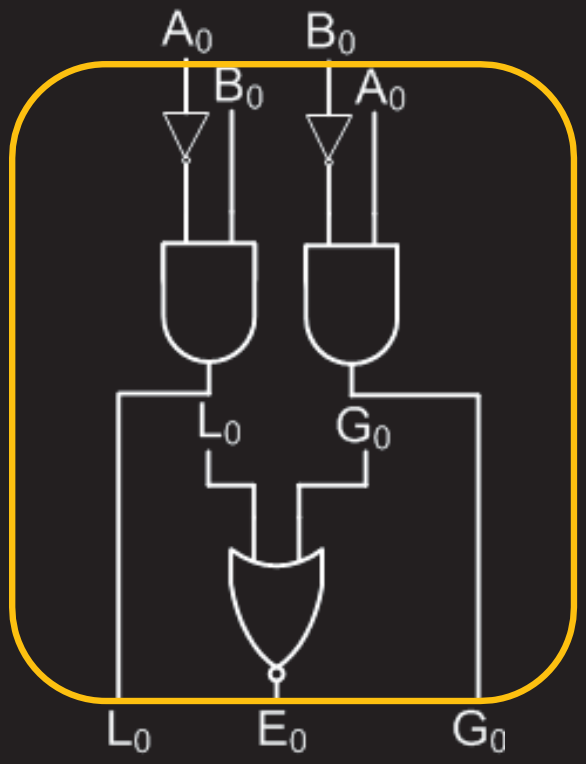
$A_0$

$B_0$

$G_0 = A_0B_0'$

$L_0 = A_0'B_0$

$E_0 = (G_0 + L_0)' = (A_0 \oplus B_0)'$




# Magnitude Comparator

1-Bit Comparator

$A_0$

$B_0$

Let $G_0$ indicate greater, $L_0$ indicate Lesser and $E_0$ indicate equal

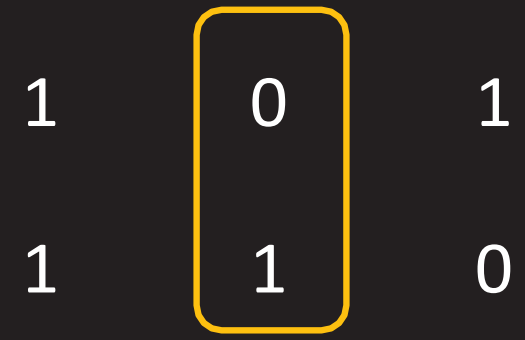
| $A_0$ | $B_0$ | $G_0$ | $L_0$ | $E_0$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 |

$G_0 = A_0B_0'$

$L_0 = A_0'B_0$

$E_0 = (A_0 \oplus B_0)' = (A_0 \odot B_0) = (G_0 + L_0)'$

# Magnitude Comparator



# Magnitude Comparator

| | | |
|---|---|---|
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Magnitude Comparator

If most significant digits are equal ??

| | | |
|---|---|---|
| 2 | 3 | 4 |
| 2 | 6 | 4 |

Compare the next significant digit

If next significant digit of first number is greater than the next significant digit of second number then

First Number > Second number

# Magnitude Comparator

How do you compare two numbers?

| | | |
|---|---|---|
| 2 | 3 | 4 |
| 5 | 6 | 4 |

Compare most significant digit

If most significant digit of first number is greater than the most significant digit of second number then

First Number > Second number

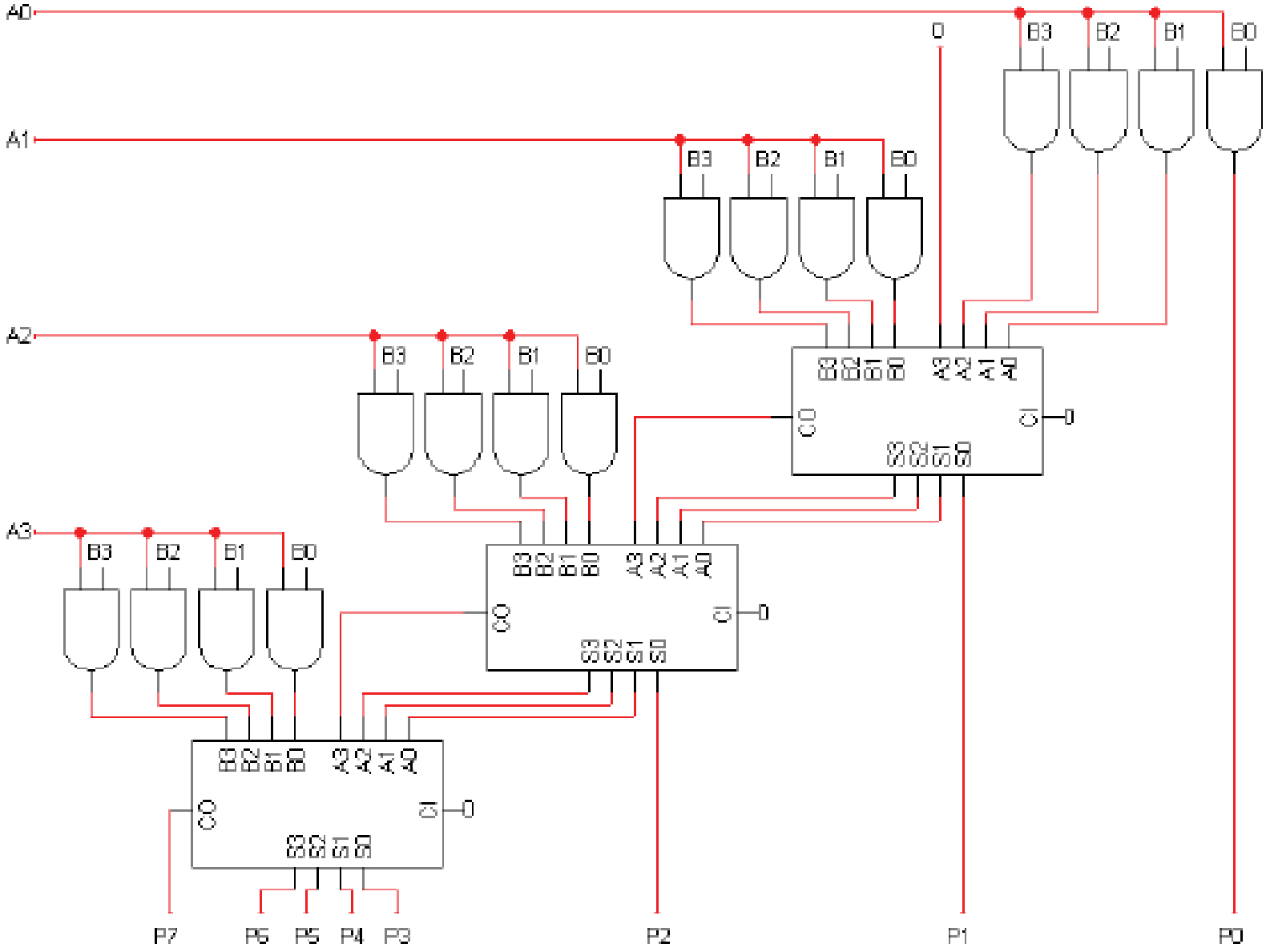
# Magnitude Comparator

A Magnitude comparator is a combinational circuit that compares two numbers A and B .

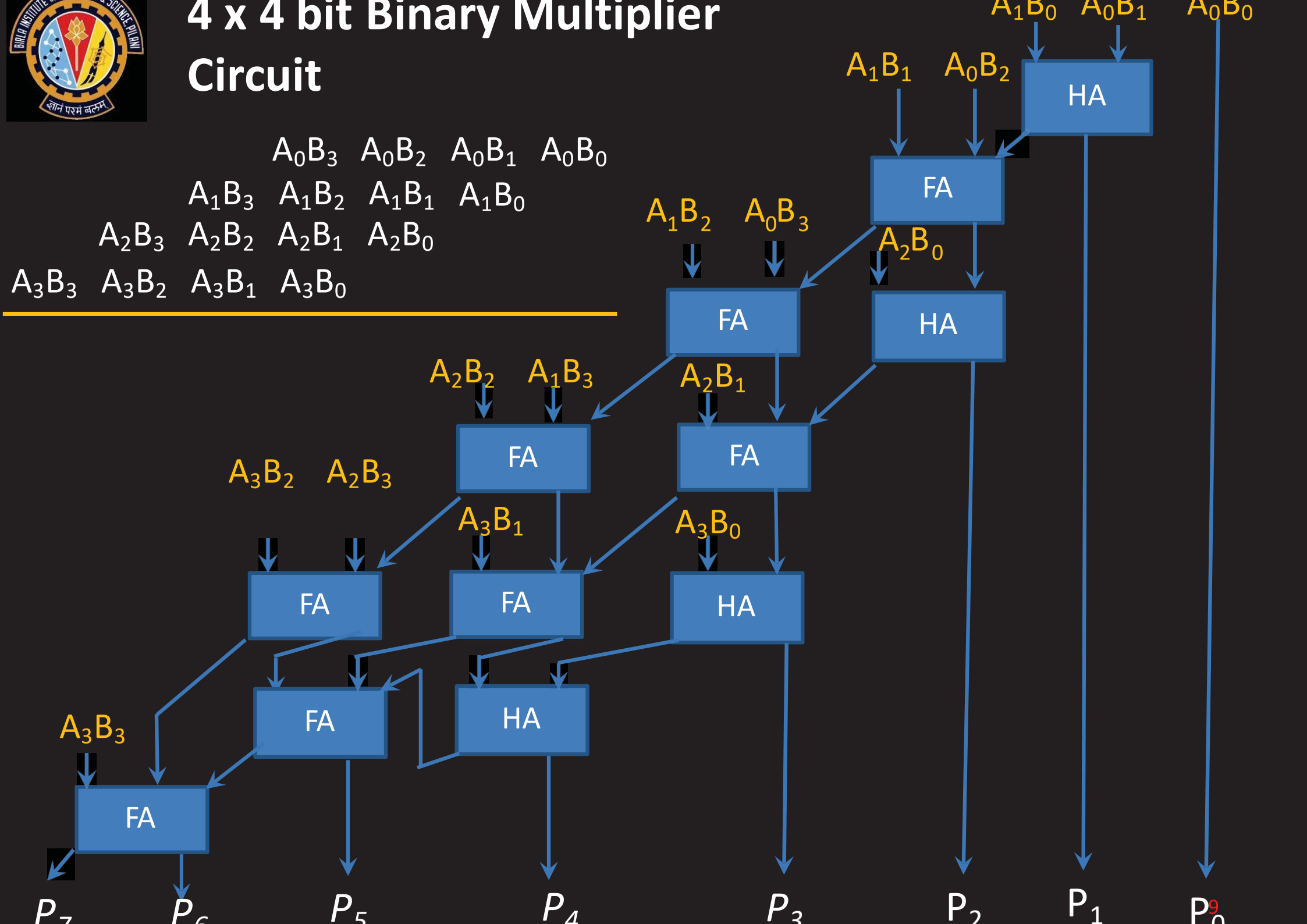
The output of the magnitude comparator are three signals that indicate if (A>B), (A<B), (A=B)

Even a 3-bit comparator results in 6 inputs – Too complex to solve using K-maps

# A 4×4 binary multiplier



# 4 x 4 bit Binary Multiplier Circuit



# Binary Multiplier (4-bit x 4-bit)

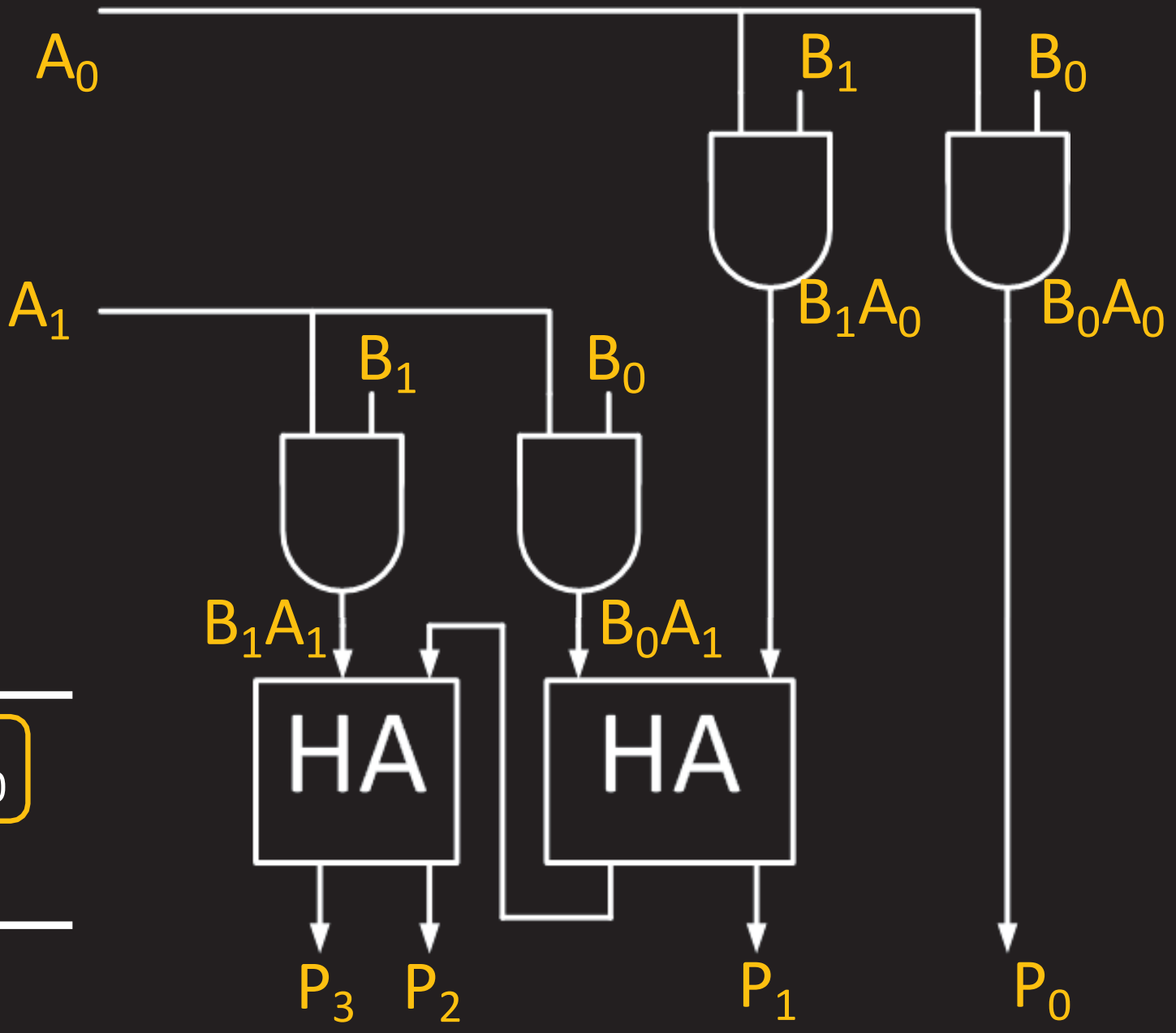
(Multiplicand) $B_3\ B_2\ B_1\ B_0$

(Multiplier) $A3\ A_2\ A_1\ A_0$

| | | | | |
|---|---|---|---|---|
| | | $A_0B_3$ | $A_0B_2$ | $A_0B_1$ | $A_0B_0$ |
| | $A_1B_3$ | $A_1B_2$ | $A_1B_1$ | $A_1B_0$ | |
| $A_2B_3$ | $A_2B_2$ | $A_2B_1$ | $A_2B_0$ | | |
| $A_3B_3$ | $A_3B_2$ | $A_3B_1$ | $A_3B_0$ | | |

# Binary Multiplier Circuit (2-bit x 2-bit)

$$
\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\times & & & \\
\hline
 & & B_1A_0 & B_0A_0 \\
 & B_1A_1 & B_0A_1 & \\
\hline
P_3 & P_2 & P_1 & P_0
\end{array}
$$



# Binary Multiplier (2-bit x 2-bit)

$B_1$ $B_0$

x $A_1$ $A_0$

$B_1A_0$ $B_0A_0$

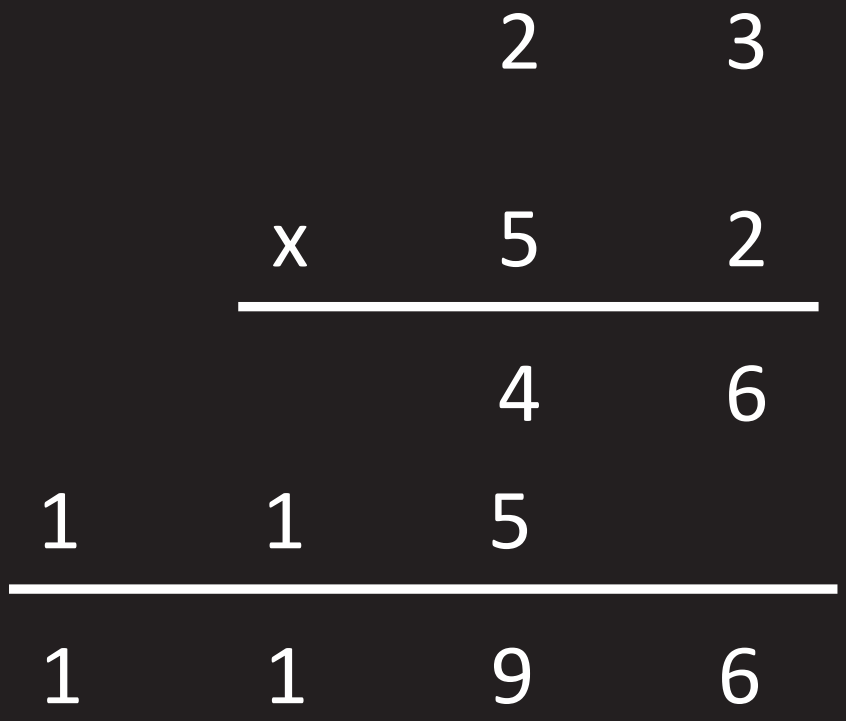
$B_1A_1$ $B_0A_1$

$P_3$ $P_2$ $P_1$ $P_0$

Gate for 1-bit Multiplication ($B_0A_0$) ?

| $A_0$ | $B_0$ | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

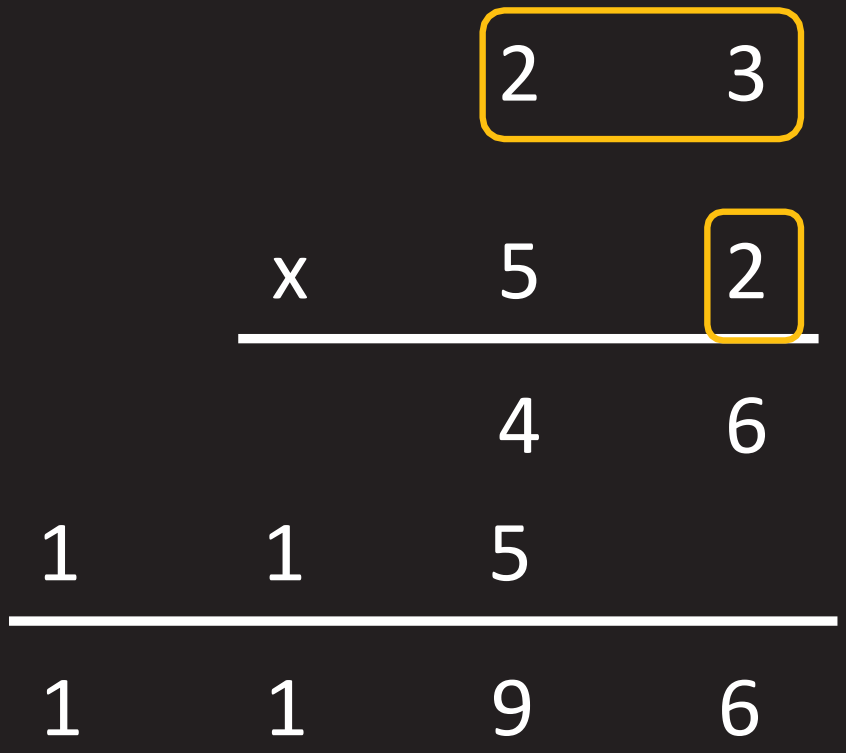
# Binary Multiplier





2 x 3 = 6

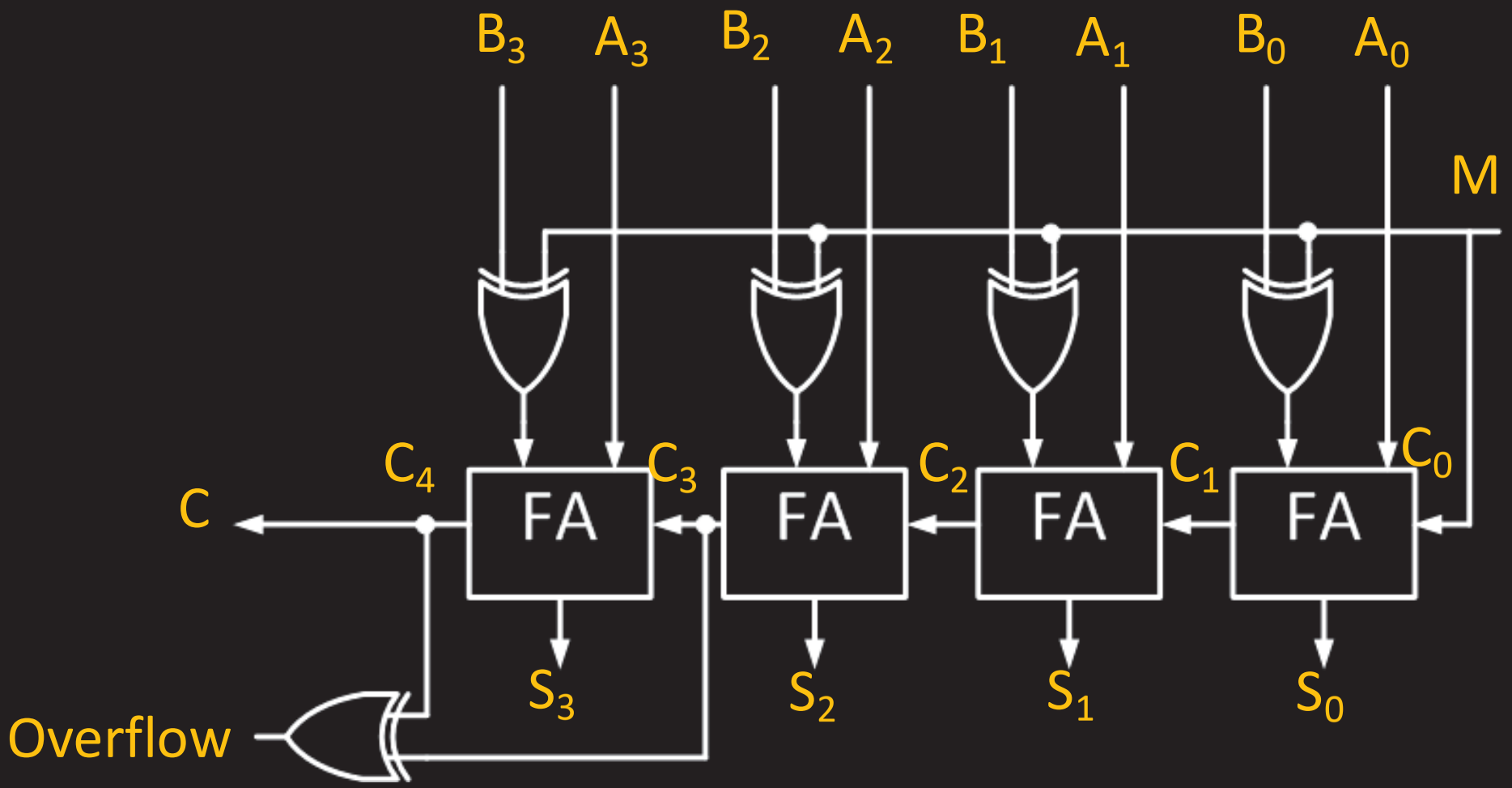
# Binary Multiplier




# Multiplier

| | | |
|---|---|---|
| | 2 | 3 |
| x | 5 | 2 |

# 4-bit Adder subtractor with overflow detection

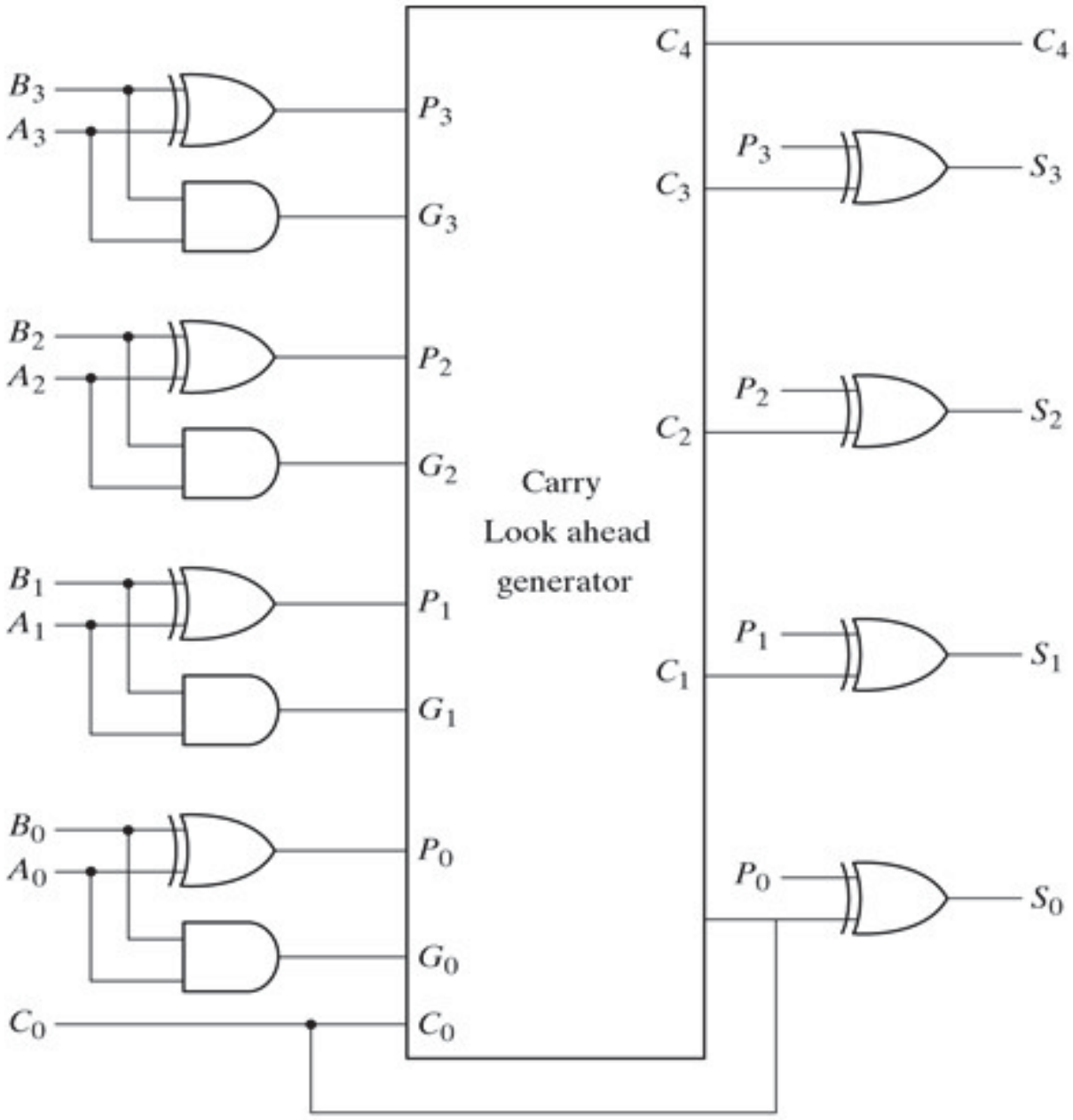


# Digital Design

## Lecture 12:

## Binary Multipliers and Magnitude Comparators

# *Thank you*

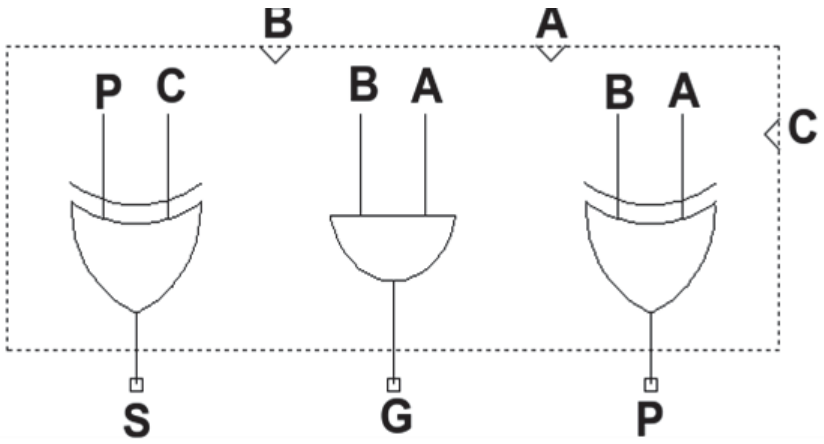


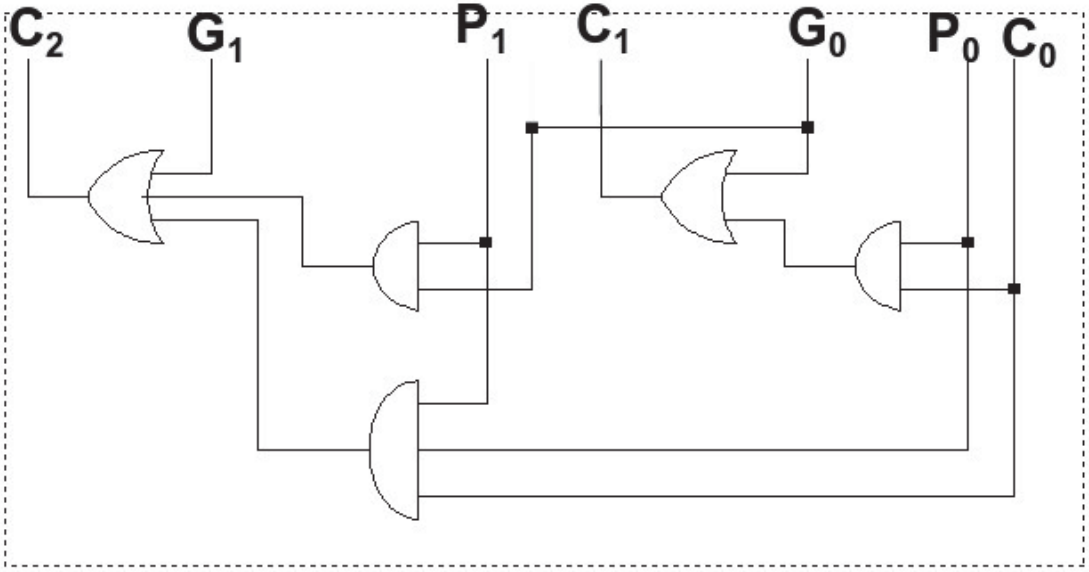

4-Bit Adder with Carry Lookahead

- Let's look at the carry out equations for specific bits, using the general equation from the previous page $C_{i+1} = G_i + P_i C_i$.

$$C_1 = G_0 + P_0 C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1 C_1 \\ &= G_1 + P_1 (G_0 + P_0 C_0) \\ &= G_1 + P_1 G_0 + P_1 P_0 C_0 \end{aligned}$$



# A faster way to compute carry outs

- Instead of waiting for the carry out from each previous stage, we can minimize the delay by computing it directly with a two-level circuit.
- First we'll define two functions.
  - The "generate" function $G_i$ produces 1 when there *must* be a carry out from position i (i.e., when $A_i$ and $B_i$ are both 1).

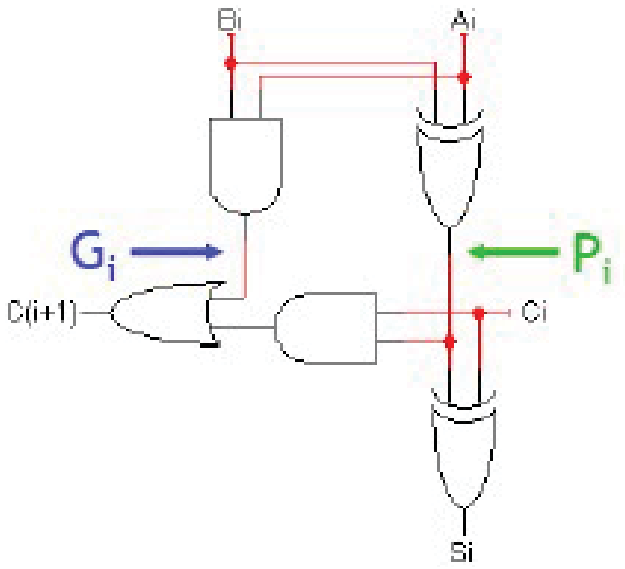
$$G_i = A_i B_i$$

  - The "propagate" function $P_i$ is true when an incoming carry is propagated (i.e, when $A_i$=1 or $B_i$=1, but not both).

$$P_i = A_i \oplus B_i$$

- Then we can rewrite the carry out function.

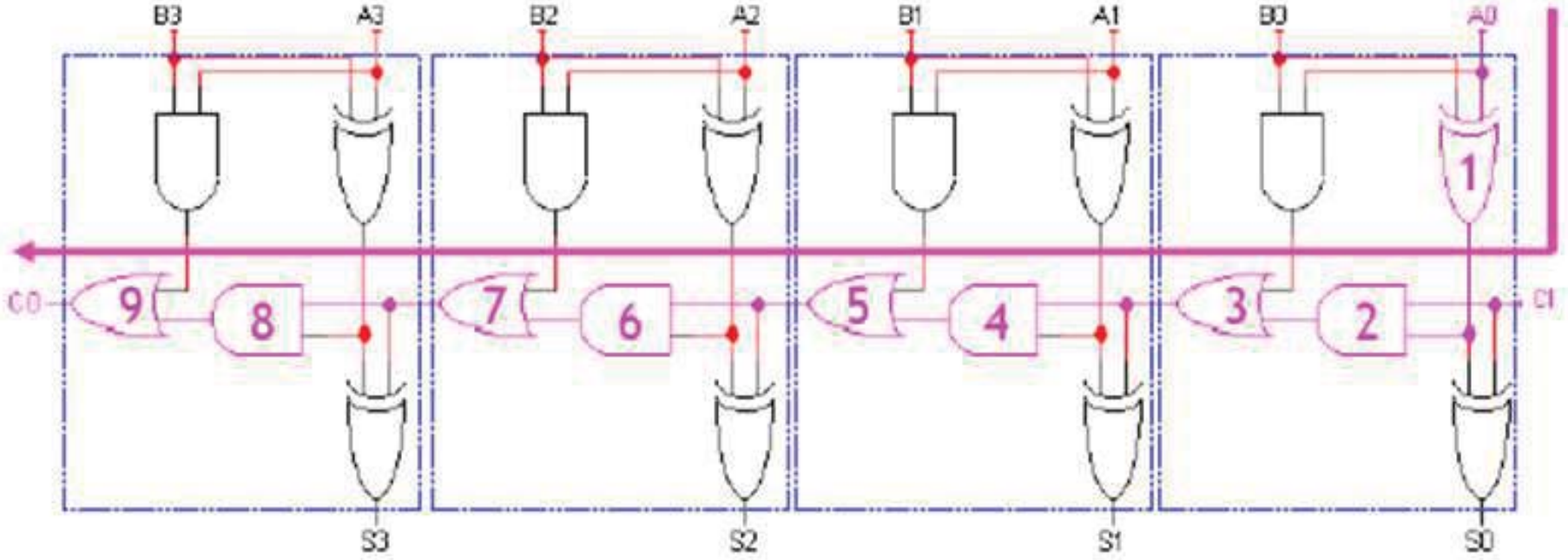
$$C_{i+1} = G_i + P_i C_i$$



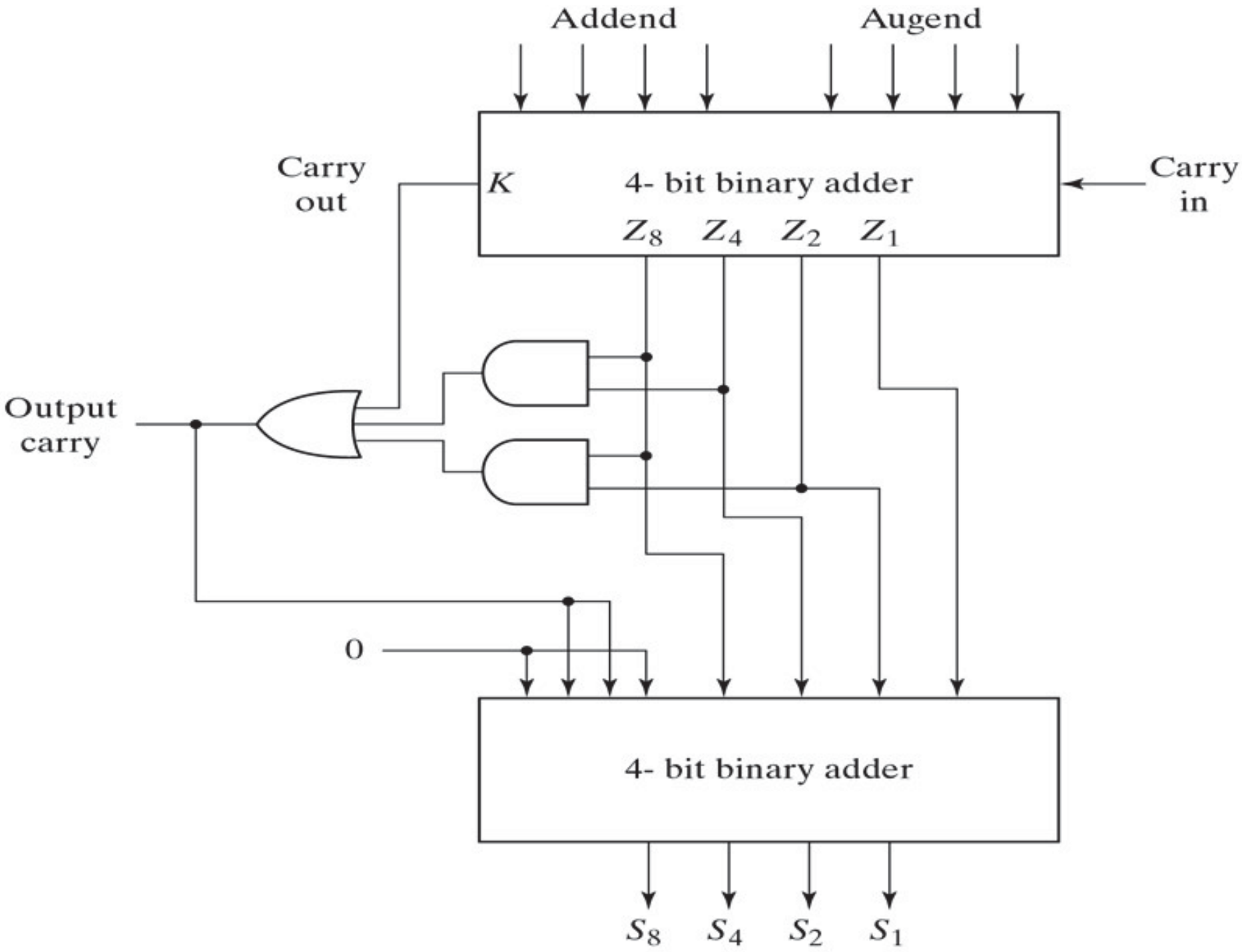

| $A_i$ | $B_i$ | $C_i$ | $C_{i+1}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

# Ripple carry delays

- This is called a ripple carry adder, because the inputs A0, B0 and CI "ripple" leftwards until CO and S3 are produced.
- Ripple carry adders are slow!
  - There is a very long path from A0, B0 and CI to CO and S3.
  - For an *n*-bit ripple carry adder, the longest path has $2n+1$ gates.
  - The longest path in a 64-bit adder would include 129 gates!



# Design of a Carry Look Ahead adder




Block Diagram of a BCD Adder

# BCD Adder

**Principle:**

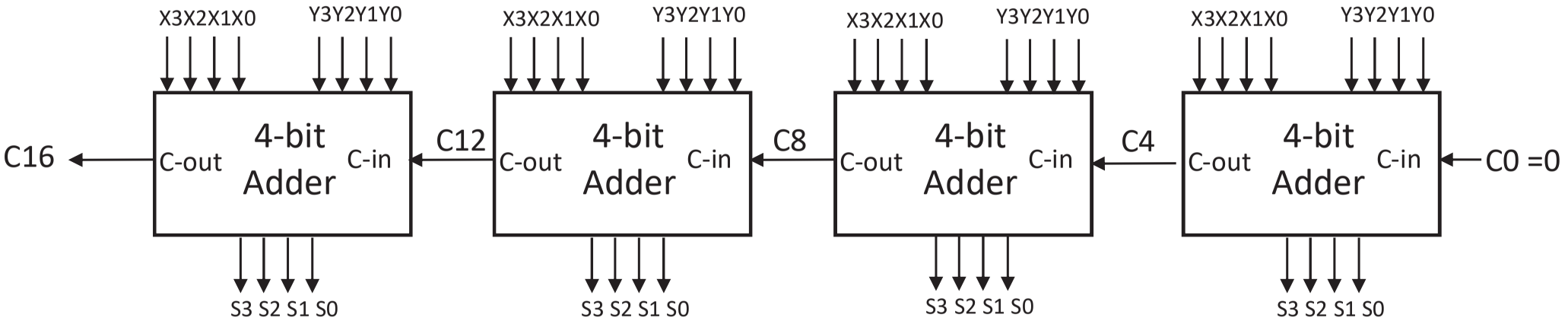
- **Is Binary sum less than or equal to 1001 (9 in decimal)?**
- **For Binary sum more than 1001, add 0110 (6 in decimal) as correction factor**
- **Circuit needs modification accordingly**

# Larger Adder

## Example: 16-bit adder using 4, 4-bit adders

**Adds two 16-bit inputs X (bits X0 to X15), Y (bits Y0 to Y15) producing a 16-bit Sum S (bits S0 to S15) and a carry out C1 from most significant position.**

**Data inputs to be added X (X0 to X15) , Y (Y0-Y15)**

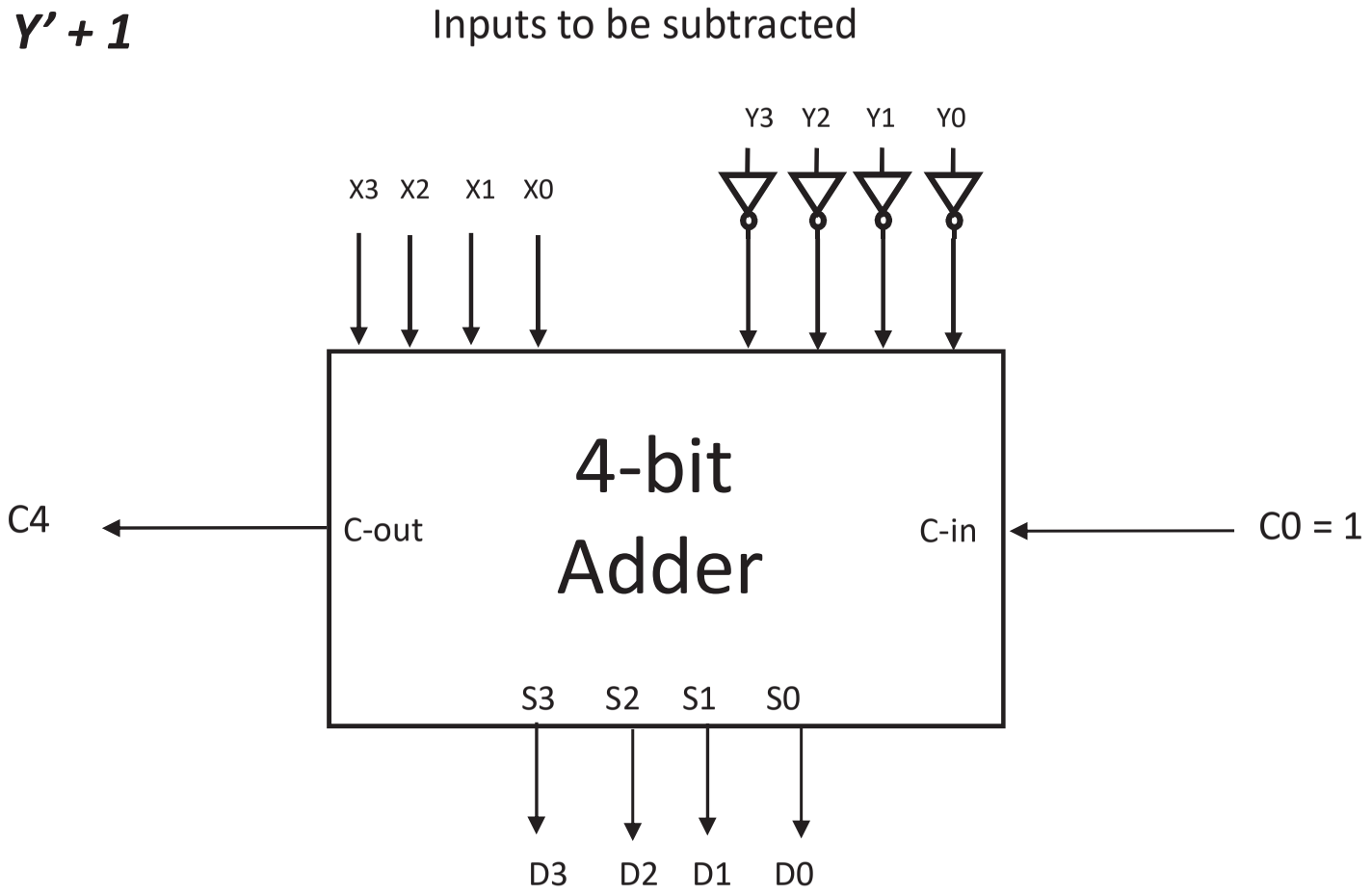



**Sum output S (S0 to S15)**

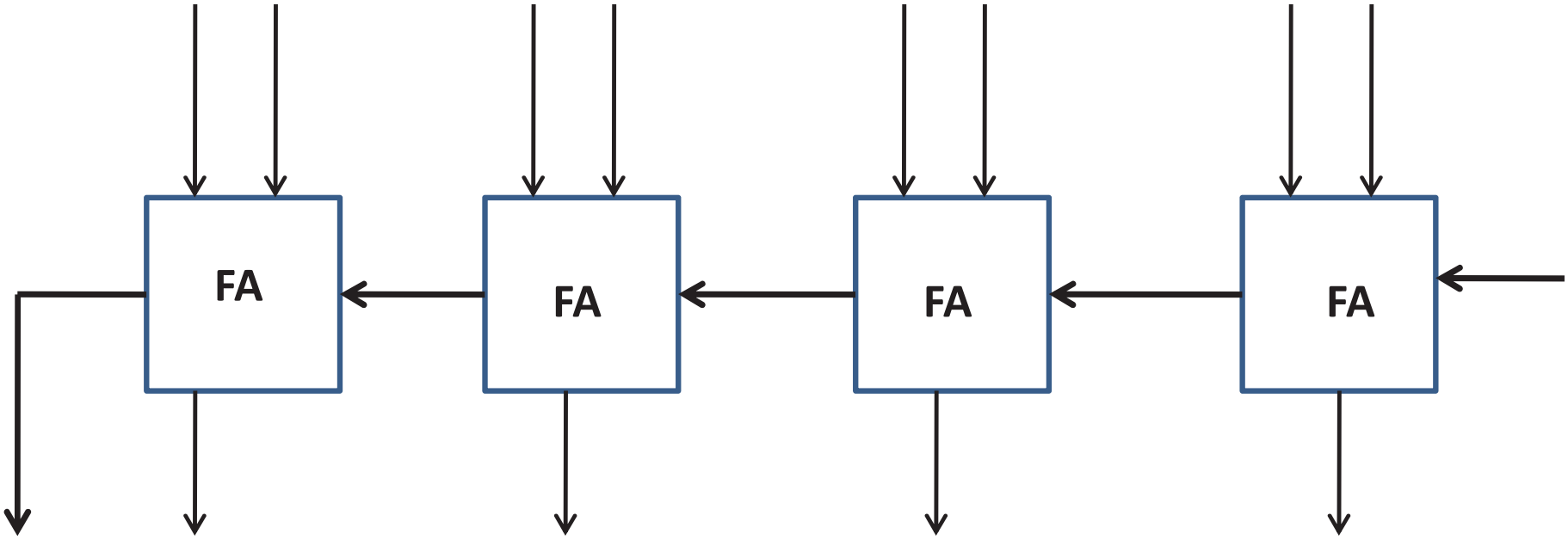
# Implementation of a Subtractor using Adder

**The subtraction *X-Y* can be performed by taking the 2's complement of *Y* and adding to *X*.**

$$X-Y = X+ 2C(Y)$$
$$= X+ 1C(Y) + 1$$
$$= X + Y' + 1$$



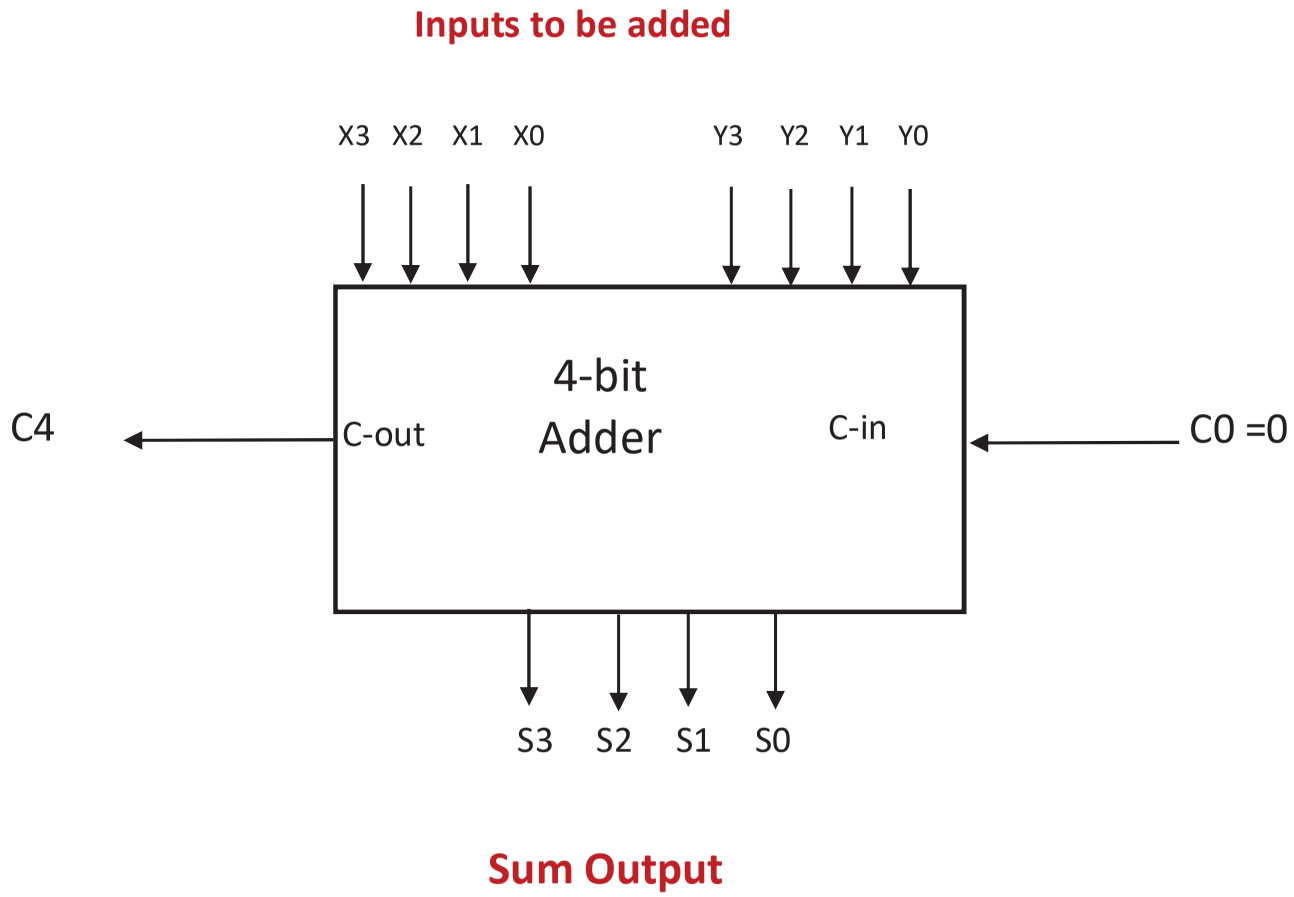
# 4 bit Binary Adder




*Each full adder represents a bit position ‘j’ (from 0 to n-1).

*Each carry out (C_out) from a full adder at position j is connected to the carry in C-in of the full adder at the higher position j+1.

# 4 bit Binary Adder



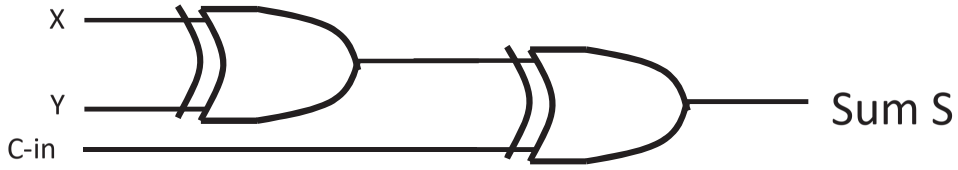

**A 4-bit Adder is used to add two 4-bit binary numbers can be built by connecting in series/cascaded 4 full adders.**

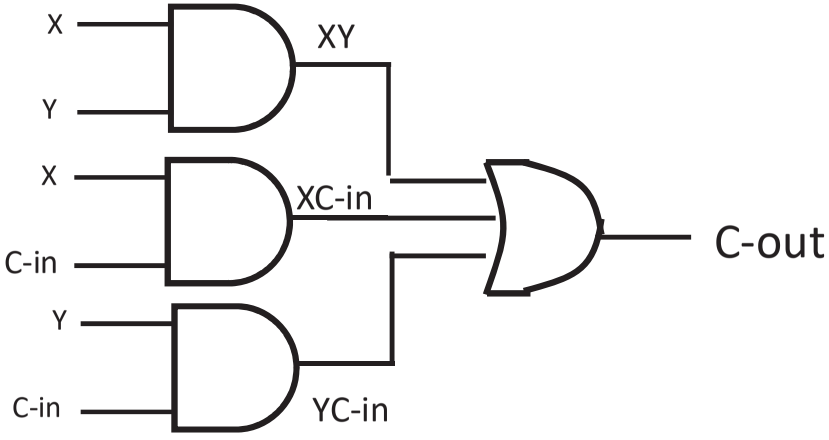
# FULL ADDER

Adding two single-bit binary values, X, Y and a carry input bit C_in, produces a sum bit S and a carry out C_out bit.

**( X + Y + Cin )**

| X | Y | C-in | S | C-out |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |



# Building blocks - contd

Subtracting a single-bit binary value Y from anther X (i.e. X -Y ) and producing a difference bit D and a borrow out bit B-out. This operation is called half addition – **Half Subtractor**

## Half Subtractor Truth Table

| Inputs | | Outputs | |
|---|---|---|---|
| X | Y | D | B_out |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

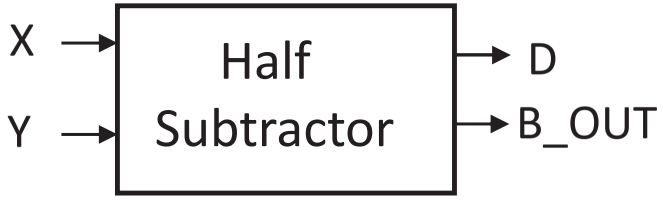
$D(X,Y) = \Sigma\,(1,2)$

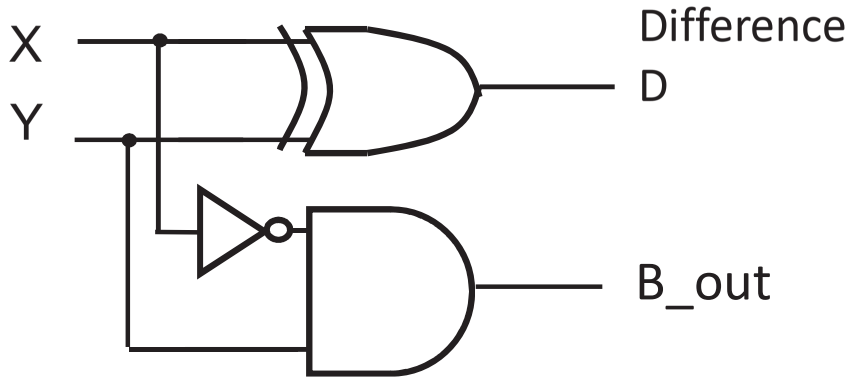
$D = X'Y + XY'$

$D = X \oplus Y$

$B\_out(x, y)$

$B\_out = X'Y$



# Adders : Building blocks

Adding two single-bit binary values, X, Y and produces a sum S and a carry out C_out

This operation is called half addition – **Half Adder**

## Half Adder Truth Table

| Inputs | | Outputs | |
|---|---|---|---|
| X | Y | S | Cout |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

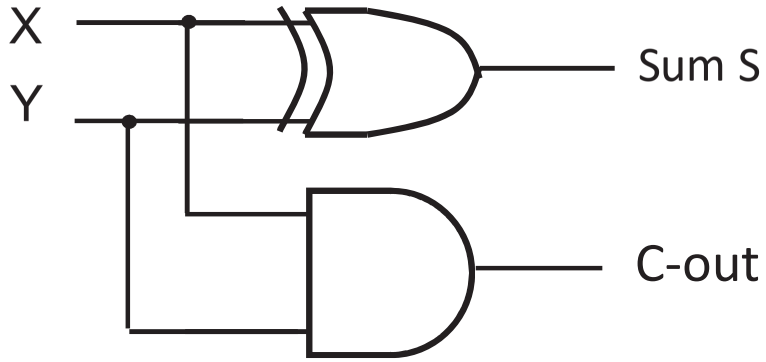
$S(X,Y) = \Sigma\ (1,2)$

$S = X'Y + XY'$

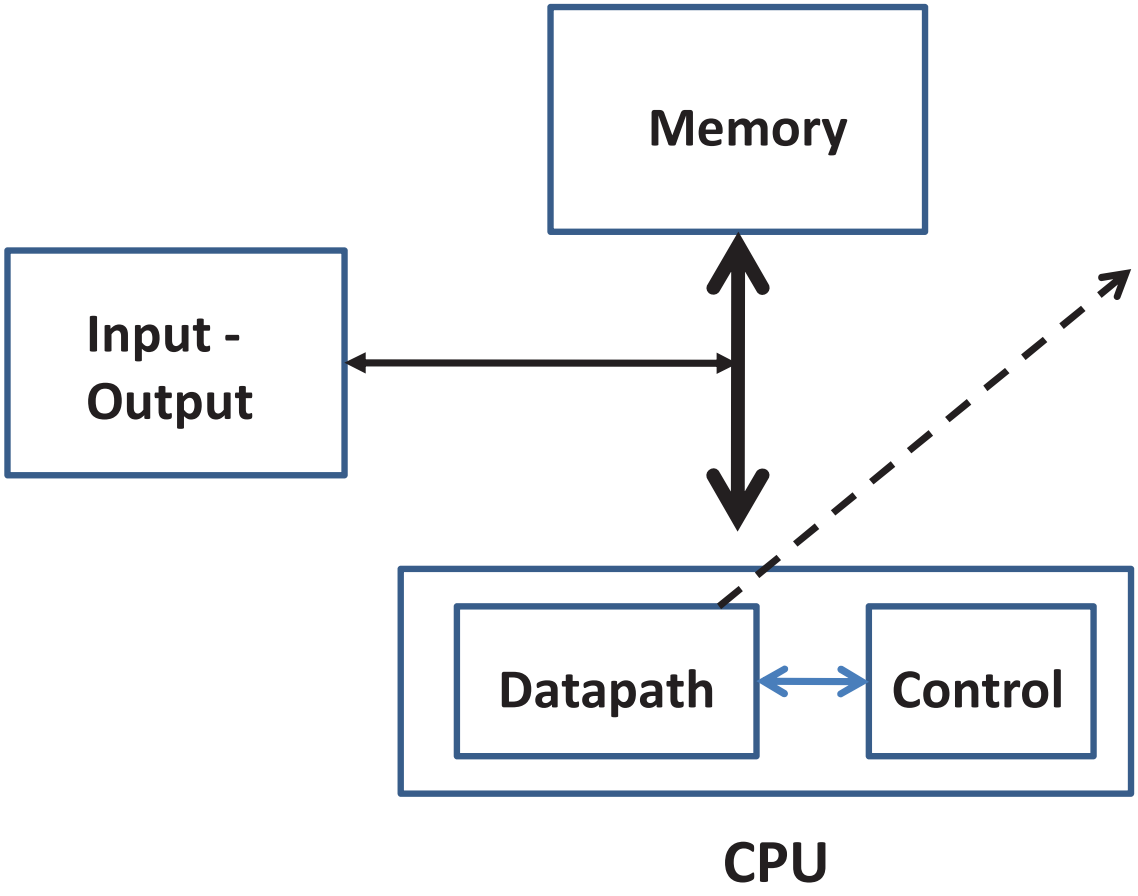
$S = X \oplus Y$

C-out(x, y)

C-out = XY

# Blocks of a Digital Processor




These circuits perform **Arithmetic & Logic Operations** which are core to central processing unit in a processor

# ARITHMETIC LOGIC CIRCUITS DESIGN

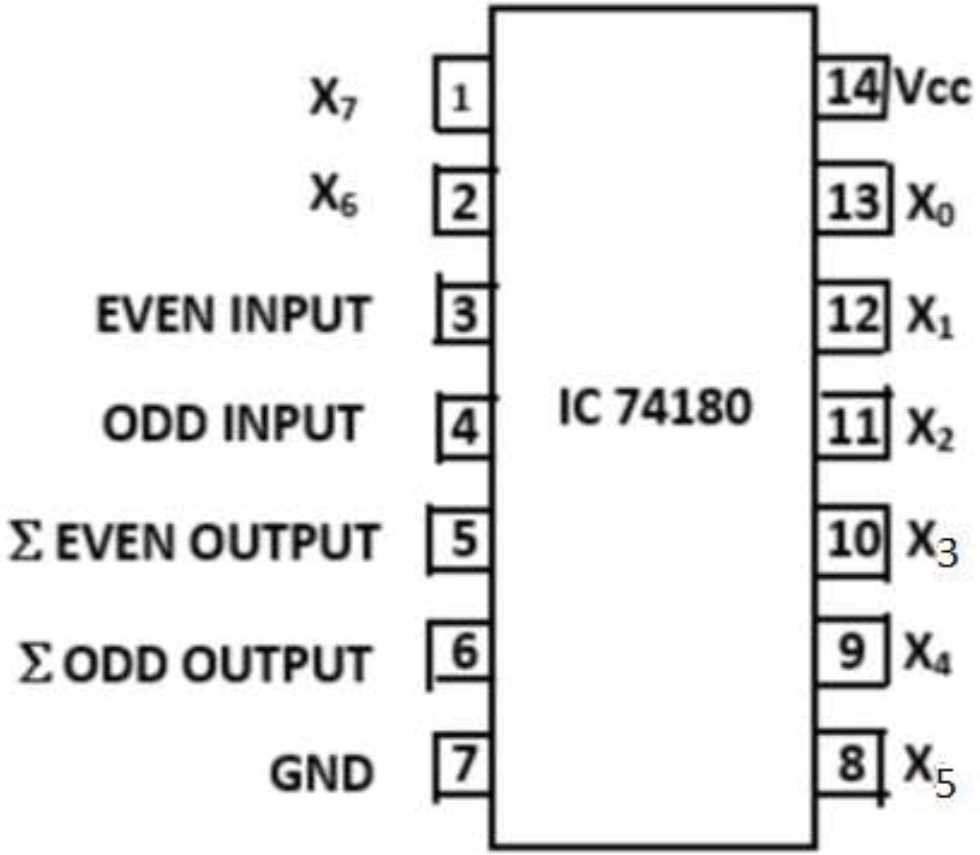
# Digital Design

## Lecture 11:
## Combinational Logic and Arithmetic Circuits

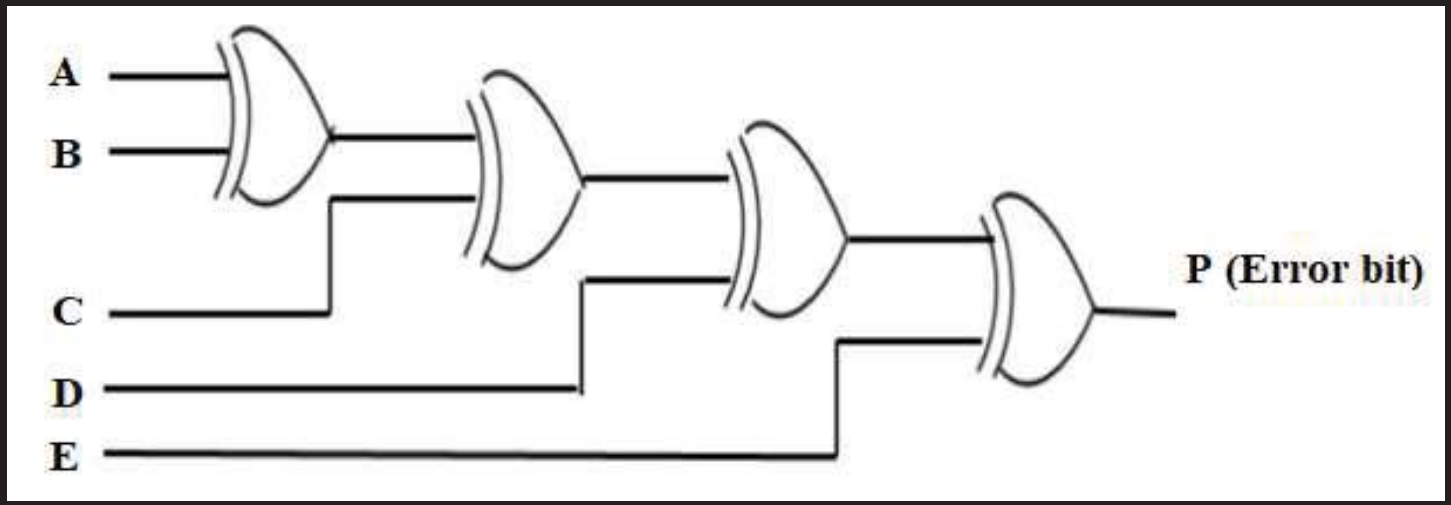
*Thank you*

## Parity Generator/Checker using IC's

The IC 74180 does the function of parity generation as well as checking. The 9 bit (8 data bits, 1 parity bit) Parity Generator/Checker is shown in the below figure.

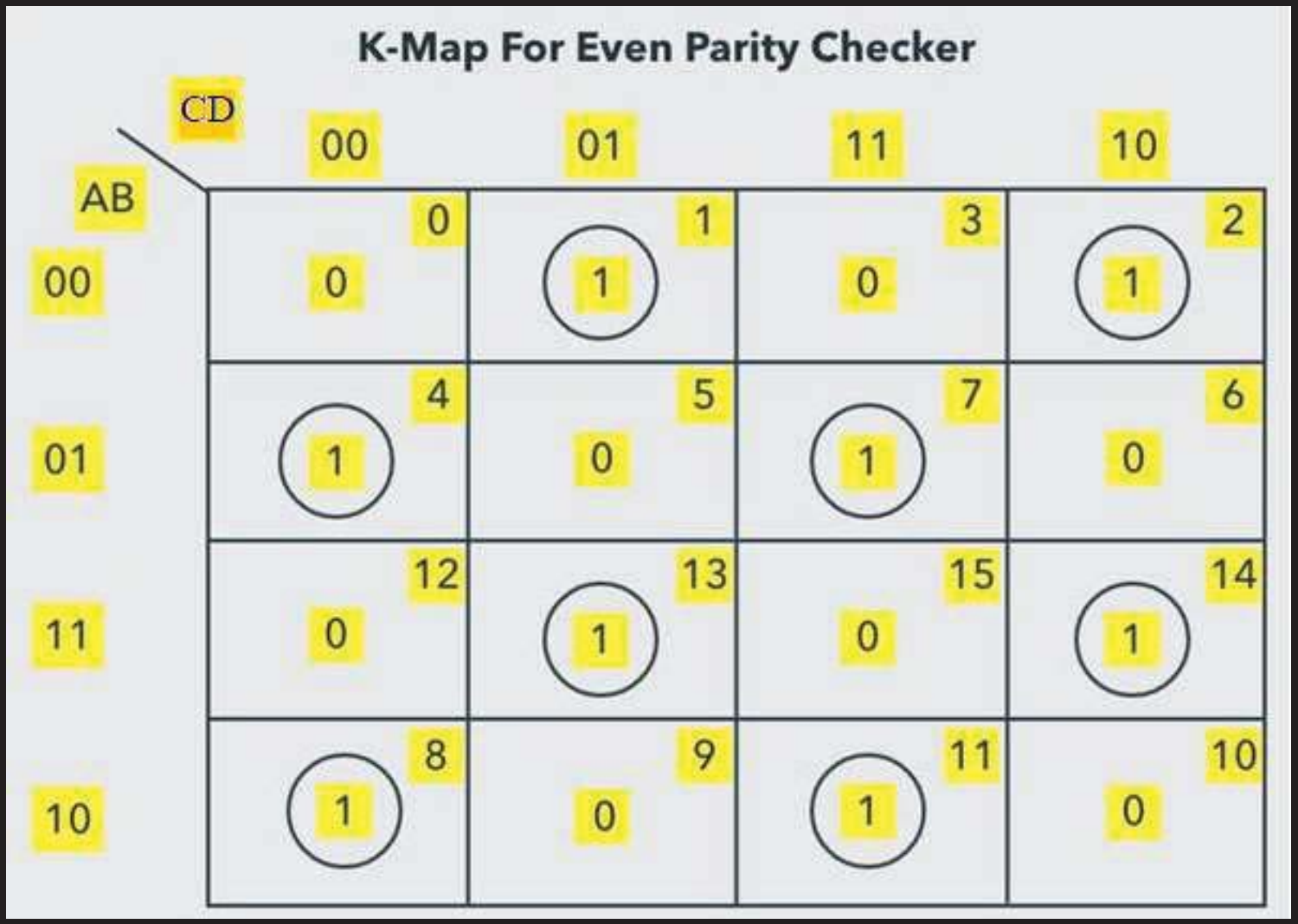


# Parity checker Logic Circuit for 5-bit received message




**An additional 4th EX-OR Gate is introduced**

## K-Map For Even Parity Checker




$$= (A \oplus B \oplus C \oplus D)$$

# Example: A 4 bit Even Parity Checker Truth Table

| 4-bit received message | | | | Parity error check P |
|---|---|---|---|---|
| A | B | C | D | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

## Even Parity Checker

In even parity checker if the error bit (E) is equal to '1', then we have an error. If error bit E=0 then indicates there is no error.

Error Bit (E) =1, error occurs

Error Bit (E) =0, no error

## Odd Parity Checker

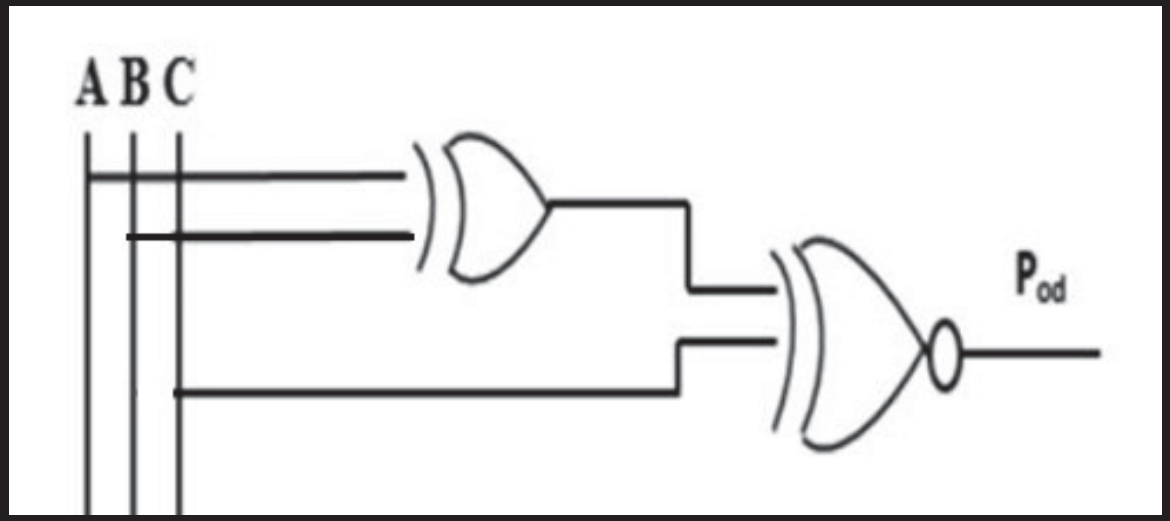
In odd parity checker if an error bit (E) is equal to '1', then it indicates there is no error. If an error bit E=0 then indicates there is an error.

Error Bit (E) =1, no error

Error Bit (E) =0, error occurs

# K- map for Odd Parity generator and its reduced expression for logic circuit design

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 (0) | 1 | 1 (3) | 2 |
| 1 | 4 | 1 (5) | 7 | 1 (6) |



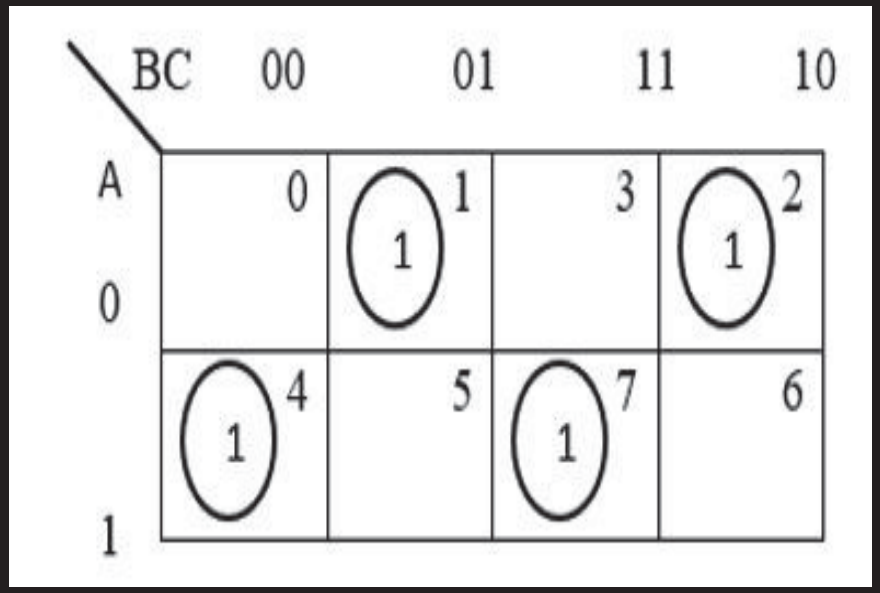

$$
\begin{aligned}
P_{od} &= (\bar{A}\,\bar{B}\,\bar{C} + A\,\bar{B}\,C) + (\bar{A}\,B\,C + A\,B\,\bar{C}) \\
&= \bar{C}\,(A\,B + \bar{A}\,\bar{B}) + B\,(A\,\bar{B} + \bar{A}\,B) \\
&= \bar{C}\,(\overline{A \oplus B}) + C(A \oplus B) \\
&= \bar{C}\,\bar{X} + C\,X = C \odot X = C \odot A \oplus B
\end{aligned}
$$

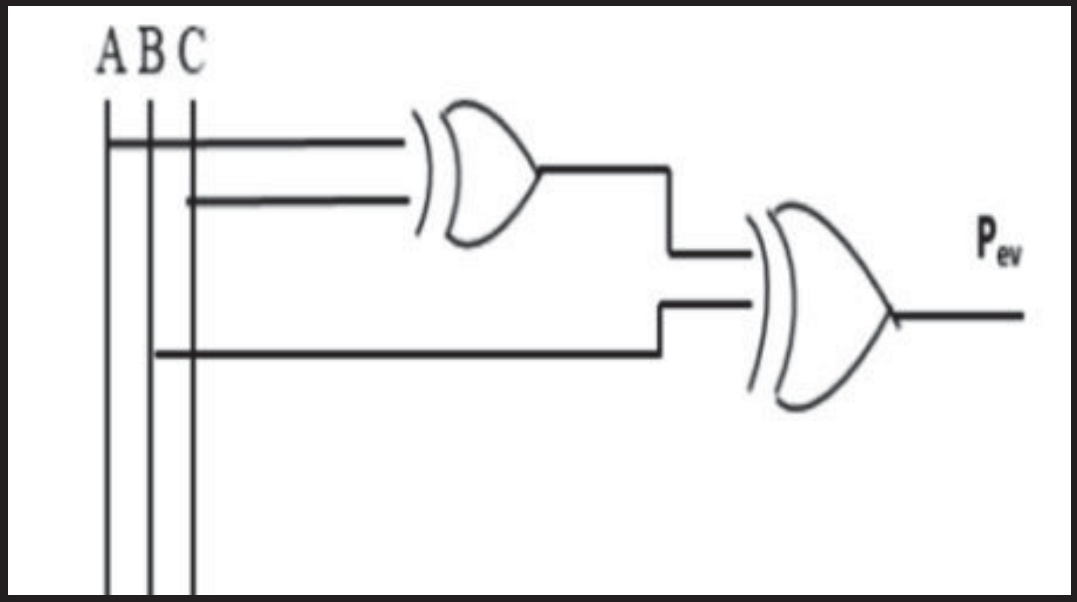
# Odd Parity Generator

**Odd Parity Generator Truth Table**

| A B C | Odd Parity |
|---|---|
| 0 0 0 | 1 |
| 0 0 1 | 0 |
| 0 1 0 | 0 |
| 0 1 1 | 1 |
| 1 0 0 | 0 |
| 1 0 1 | 1 |
| 1 1 0 | 1 |
| 1 1 1 | 0 |

# K- map for Even Parity generator and its reduced expression for logic circuit design




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

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

$$= \bar{B}\,(A \oplus C) + B\,(\overline{A \oplus C}) = \bar{B}x + B\bar{x} = \mathrm{B} \oplus \mathrm{x} = \mathrm{B} \oplus \mathrm{A} \oplus \mathrm{C}$$

# Even Parity Generator

**Even Parity Generator Truth Table**

| A B C | Even Parity |
|---|---|
| 0 0 0 | 0 |
| 0 0 1 | 1 |
| 0 1 0 | 1 |
| 0 1 1 | 0 |
| 1 0 0 | 1 |
| 1 0 1 | 0 |
| 1 1 0 | 0 |
| 1 1 1 | 1 |

# Parity Generation and Checking:

- Parity is used to detect errors in transmitted data caused by noise or other disturbances.
- A parity bit is an extra bit that is added to a data word and can be either odd or even parity.
- In an even parity system, the sum of all the bits (including the parity bit) is an even number
- In an odd parity system the sum of all the bits must be an odd number.
- The circuit that creates the parity bit at the transmitter is called the parity generator.
- The circuit that determines if the received data is correct is the parity checker.
- Parity is good for detecting a single bit error only.
- The parity generator and the parity checker can both be built using Exclusive-OR gates.

- A combinational circuit consists of logic gates whose outputs at any time are determined by combining the values of the inputs.
- For *n* input variables, there are $2^n$ possible binary input combinations.
- For each binary combination of the input variables, there is one possible output.

# Digital Design

## CS / EEE / ECE/ INSTR F215

## Lecture 10: Combinational Circuits