

Combinational Logic

ENCS2340 - Digital Systems

Dr. Ahmed I. A. Shawahna

Electrical and Computer Engineering Department

Birzeit University

Presentation Outline

❖ **Combinational Circuits**

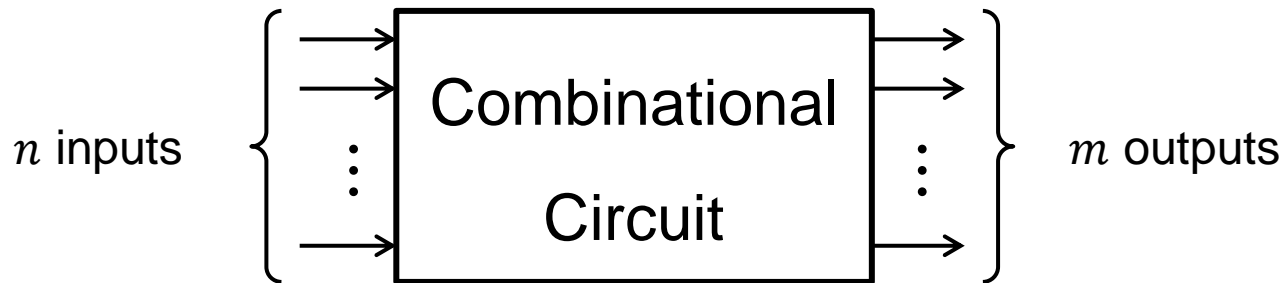
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Design Examples

Combinational Circuits

❖ A combinational circuit is a block of **logic gates** having:

n inputs: x_1, x_2, \dots, x_n

m outputs: f_1, f_2, \dots, f_m

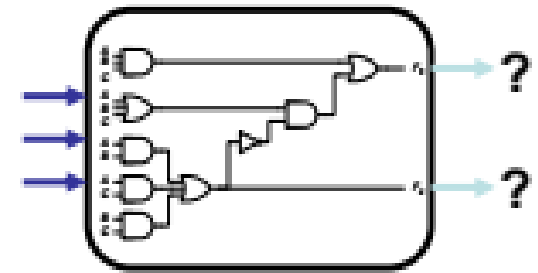


- ❖ Each output is a function of the input variables
- ❖ Each output is determined from **present combination** of inputs
- ❖ Combination circuit performs operation specified by logic gates
- ❖ The logic diagram has **no** feedback paths or memory elements

Combinational Circuits

❖ Analysis:

- ✧ Given a circuit (a logic diagram), find out its function
- ✧ Function may be expressed as:
 - Boolean function
 - Truth table



❖ Design:

- ✧ Given a desired function, determine its circuit (logic diagram)
- ✧ Function may be expressed as:
 - Boolean function
 - Truth table



Functional Blocks

- ❖ A functional block is a combinational circuit
- ❖ We will study blocks, such as decoders and multiplexers
- ❖ Functional blocks are very common and useful in design
- ❖ In the past, functional blocks were integrated circuits

SSI: Small Scale Integration = tens of gates

MSI: Medium Scale Integration = hundreds of gates

LSI: Large Scale Integration = thousands of gates

VLSI: Very Large Scale Integration = millions of gates

- ❖ Today, functional blocks are part of a design library
- ❖ Tested for correctness and reused in many projects

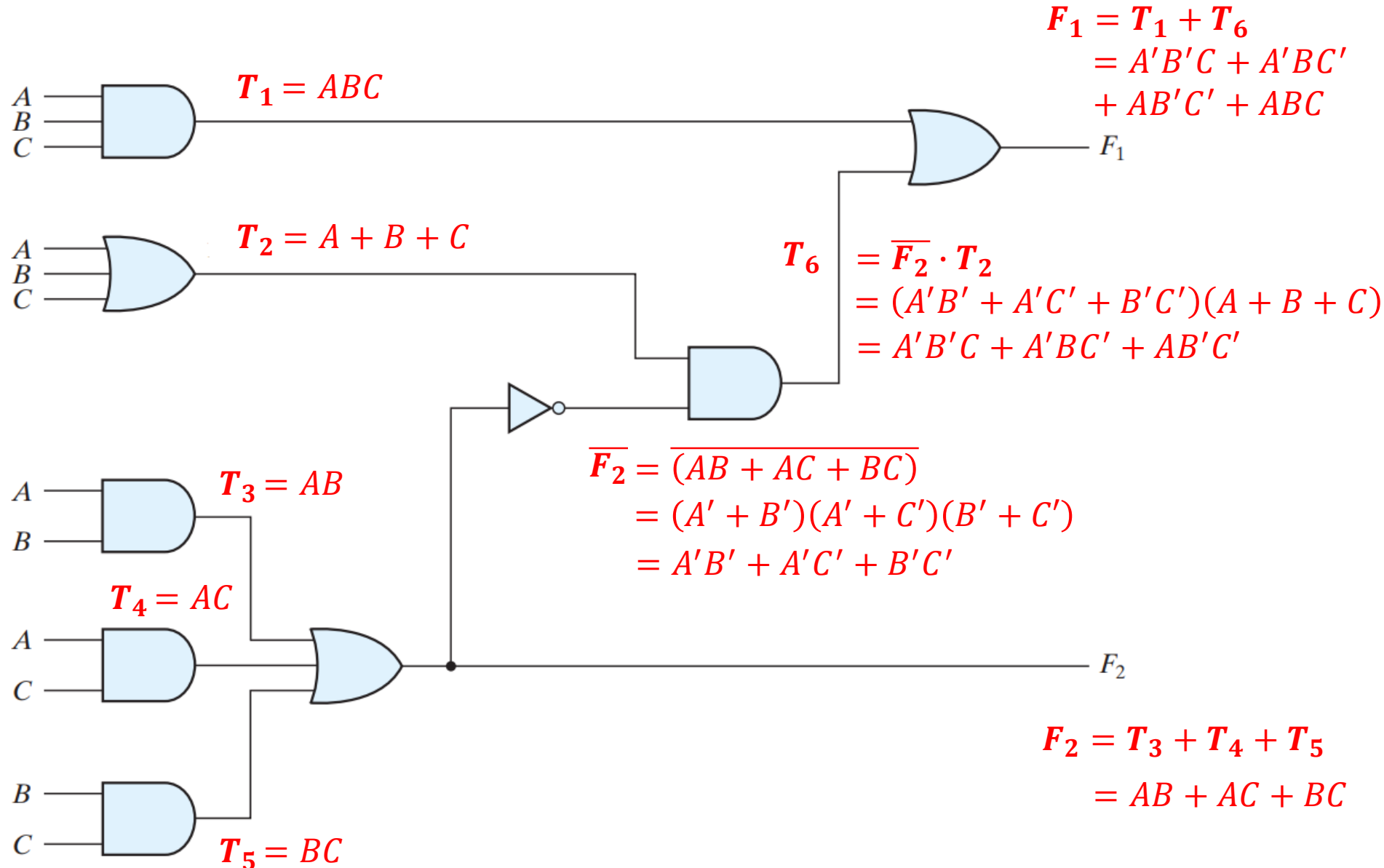
Next . . .

- ❖ Combinational Circuits
- ❖ **Analysis Procedure**
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Design Examples

Analysis Procedure - Boolean Function

1. Label all gate outputs that are a function of input variables with symbols. Determine the Boolean function for each gate output.
2. Label the gates that are a function of input variables and previously labeled gates with other symbols. Find the Boolean functions for these gates.
3. Repeat step 2 until output of circuits are obtained.
4. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of input variables.

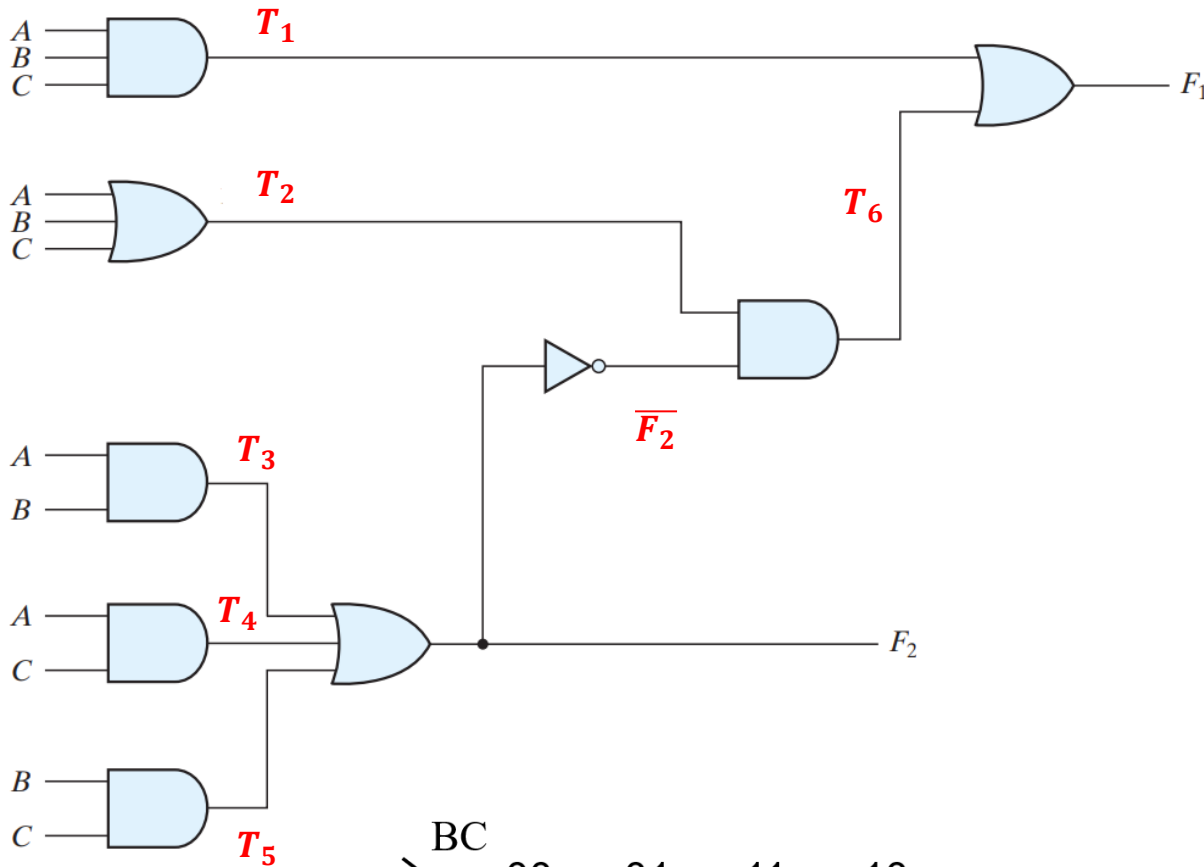
Analysis Procedure - Boolean Function



Analysis Procedure - Truth Table

1. Determine the number of input variables in the circuit. For n inputs, form the 2^n possible input combinations and list the binary numbers from **0** to **($2^n - 1$)** in a table.
2. Label the outputs of selected gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates which are a function of the input variables only.
4. Proceed to obtain the truth table for the outputs of those gates which are a function of previously defined values until the columns for all outputs are determined.

Analysis Procedure - Truth Table



Truth Table

A	B	C	T_1	T_2	T_3	T_4	T_5	F_2	$\overline{F_2}$	T_6	F_1
0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	0	1	0	0	0	0	1	1	1
0	1	0	0	1	0	0	0	0	1	1	1
0	1	1	0	1	0	0	1	1	0	0	0
1	0	0	0	1	0	0	0	0	1	1	1
1	0	1	0	1	0	1	0	1	0	0	0
1	1	0	0	1	1	0	0	1	0	0	0
1	1	1	1	1	1	1	1	1	0	0	1

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

$$F_2 = AB + AC + BC$$

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

$$F_1 = A'B'C + A'BC' + AB'C' + ABC$$

Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ **Design Procedure**
 - ✧ **Designing a BCD to Excess-3 Code Converter**
 - ✧ **Designing a BCD to 7-Segment Decoder**
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Design Examples

How to Design a Combinational Circuit

1. Specification

- ✧ Specify the inputs, outputs, and what the circuit should do

2. Formulation

- ✧ Convert the specification into truth tables or logic expressions for outputs

3. Logic Minimization

- ✧ Minimize the output functions using K-map or Boolean algebra

4. Technology Mapping

- ✧ Draw a logic diagram using ANDs, ORs, and inverters
- ✧ Map the logic diagram into the selected technology
- ✧ Considerations: cost, delays, fan-in, fan-out

5. Verification

- ✧ Verify the correctness of the design, either manually or using simulation

Verification Methods

❖ Manual Logic Analysis

- ✧ Find the logic expressions and truth table of the final circuit
- ✧ Compare the final circuit truth table against the specified truth table
- ✧ Compare the circuit output expressions against the specified expressions
- ✧ Tedious for large designs + Human Errors

❖ Simulation

- ✧ Simulate the final circuit, possibly written in HDL (such as Verilog)
- ✧ Write a test bench that automates the verification process
- ✧ Generate test cases for ALL possible inputs (exhaustive testing)
- ✧ Verify the output correctness for ALL input test cases
- ✧ Exhaustive testing can be very time consuming for many inputs

Designing a BCD to Excess-3 Code Converter

1. Specification:

- ✧ Input: BCD code for decimal digits 0 to 9
- ✧ Output: Excess-3 code for digits 0 to 9
- ✧ Convert BCD code to Excess-3 code

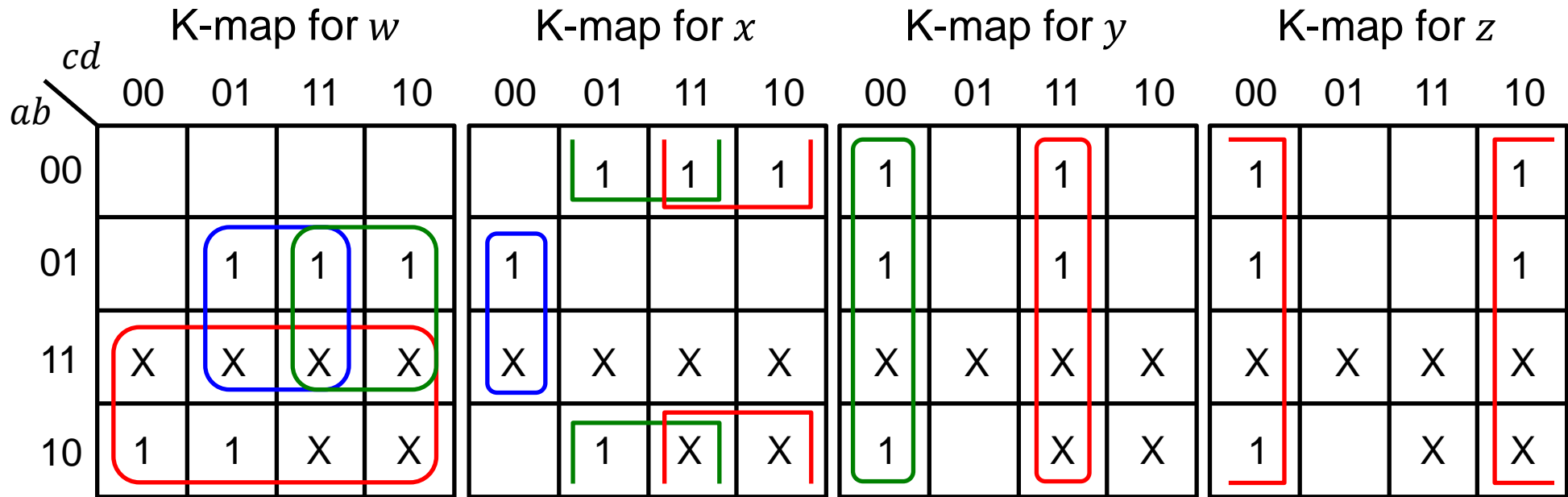
2. Formulation:

- ✧ Done easily with a truth table
- ✧ BCD input: a, b, c, d
- ✧ Excess-3 output: w, x, y, z
- ✧ Output is don't care for 1010 to 1111

BCD				Excess-3			
a	b	c	d	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1010 to 1111				X	X	X	X

Designing a BCD to Excess-3 Code Converter

3. Logic Minimization using K-maps:



Minimal Sum-of-Products expressions:

$$w = a + bc + bd, \quad x = b'c + b'd + bc'd', \quad y = cd + c'd', \quad z = d'$$

Additional 3-Level Optimizations: extract common term $(c + d)$

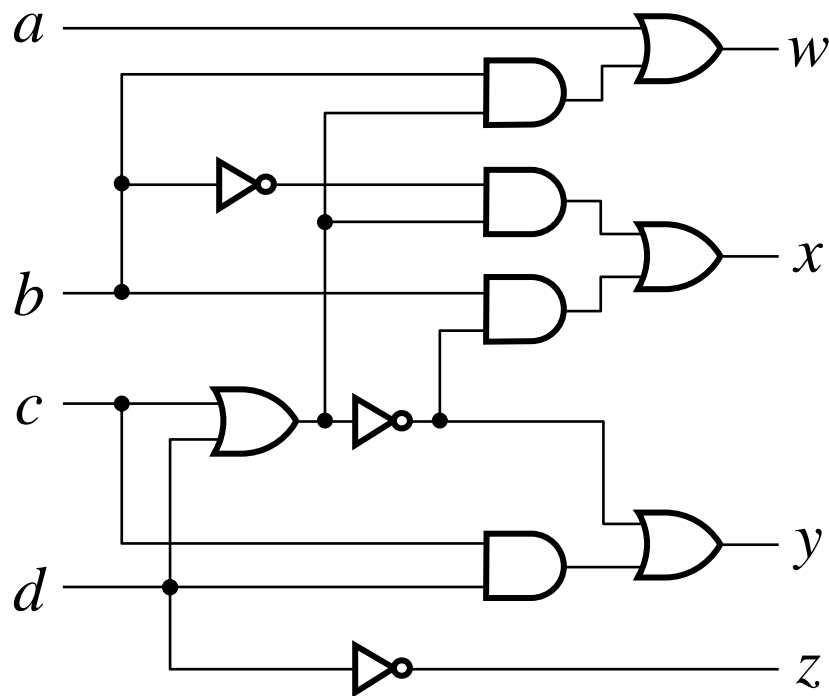
$$w = a + b(c + d), \quad x = b'(c + d) + b(c + d)', \quad y = cd + (c + d)'$$

Designing a BCD to Excess-3 Code Converter

4. Technology Mapping:

Draw a logic diagram using ANDs, ORs, and inverters

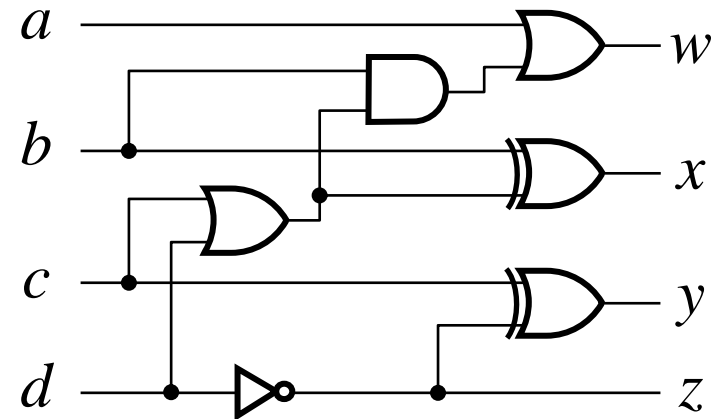
Other gates can be used, such as NAND, NOR, and XOR



Using XOR gates

$$x = b'(c + d) + b(c + d)' = b \oplus (c + d)$$

$$y = cd + c'd' = (c \oplus d)' = c \oplus d'$$



Designing a BCD to Excess-3 Code Converter

5. Verification:

Can be done manually

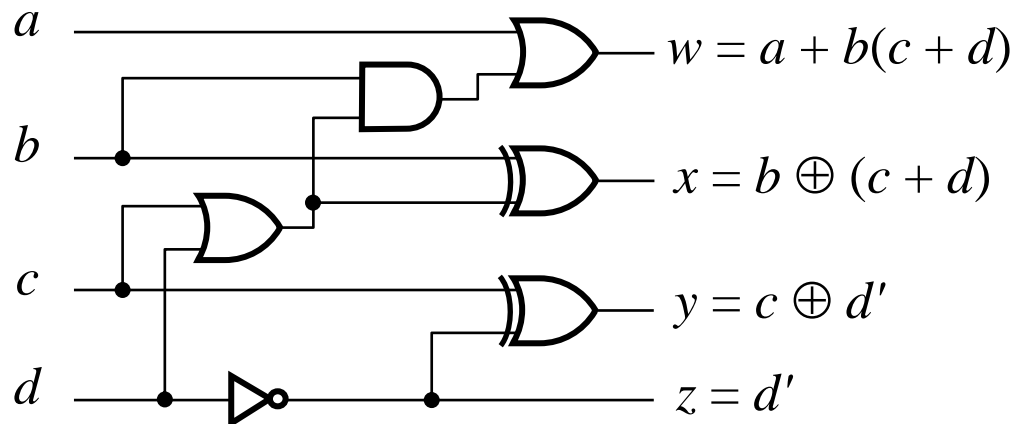
Extract output functions from circuit diagram

Find the truth table of the circuit diagram

Match it against the specification truth table

Verification process can be automated

Using a simulator for complex designs



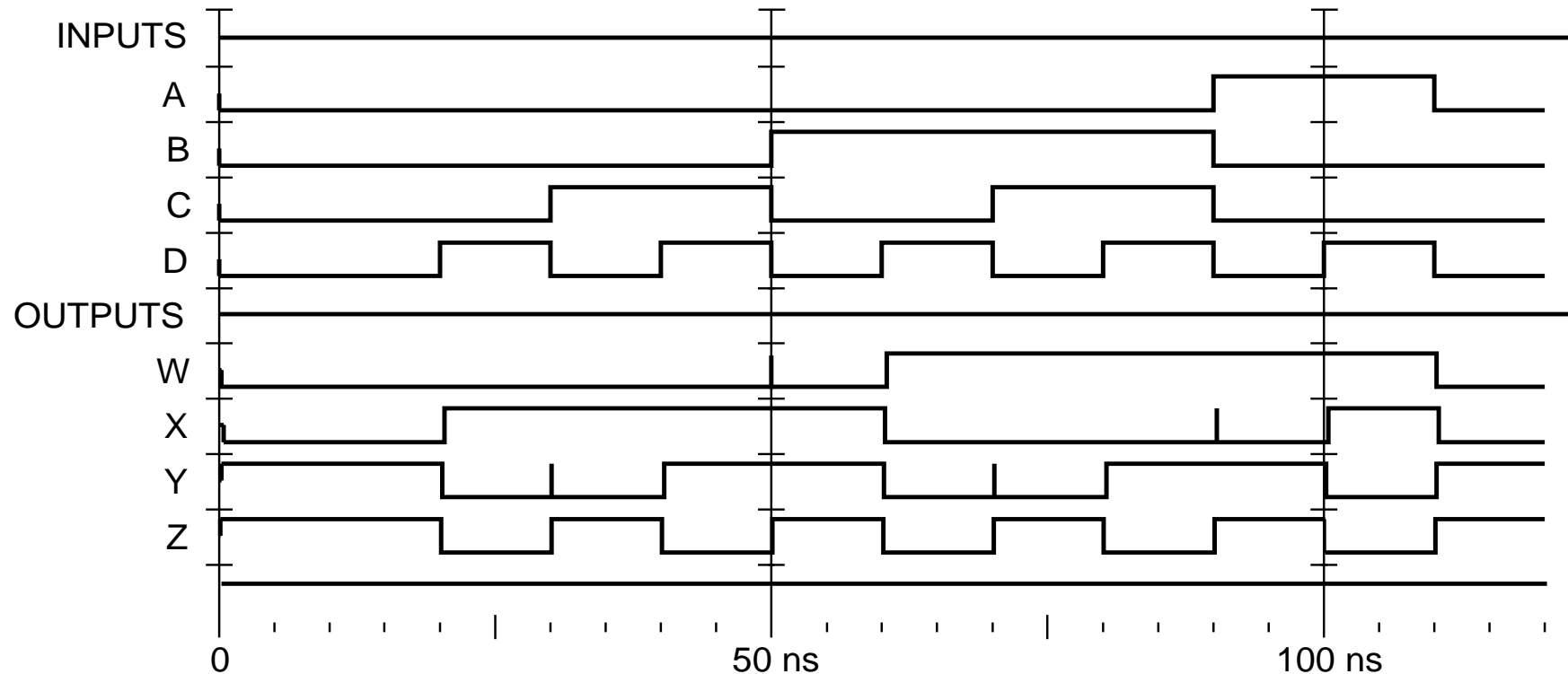
Truth Table of the
Circuit Diagram

BCD a b c d	c+d	b(c+d)	Excess-3 w x y z
0 0 0 0	0	0	0 0 1 1
0 0 0 1	1	0	0 1 0 0
0 0 1 0	1	0	0 1 0 1
0 0 1 1	1	0	0 1 1 0
0 1 0 0	0	0	0 1 1 1
0 1 0 1	1	1	1 0 0 0
0 1 1 0	1	1	1 0 0 1
0 1 1 1	1	1	1 0 1 0
1 0 0 0	0	0	1 0 1 1
1 0 0 1	1	0	1 1 0 0

Designing a BCD to Excess-3 Code Converter

5. Verification:

Run the simulation of the circuit

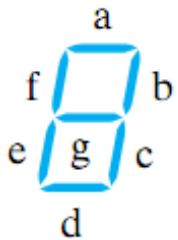
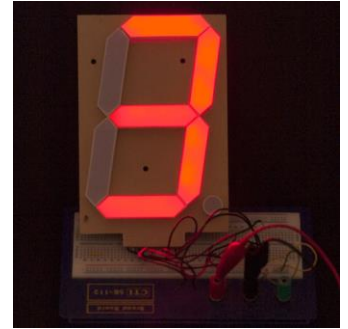


Do the simulation output combinations match the original specification truth table?

BCD to 7-Segment Decoder

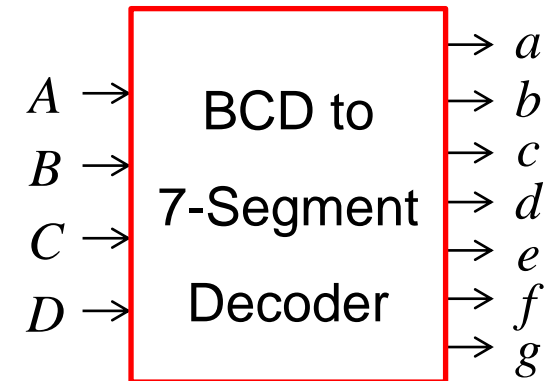
❖ Seven-Segment Display:

- ✧ Made of Seven segments: light-emitting diodes (LED)
- ✧ Found in electronic devices: such as clocks, calculators, etc.



❖ BCD to 7-Segment Decoder

- ✧ Accepts as input a BCD decimal digit (0 to 9)
- ✧ Generates output to the seven LED segments to display the BCD digit
- ✧ Each segment can be turned on or off separately



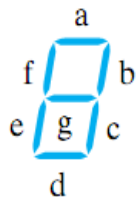
Designing a BCD to 7-Segment Decoder

1. Specification:

- ✧ Input: 4-bit BCD (A, B, C, D)
- ✧ Output: 7-bit (a, b, c, d, e, f, g)
- ✧ Display should be OFF for Non-BCD input codes

2. Formulation:

- ✧ Done with a truth table
- ✧ Output is zero for 1010 to 1111



0 1 2 3 4 5 6 7 8 9

Truth Table

BCD input				7-Segment decoder						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1010 to 1111				0	0	0	0	0	0	0

Designing a BCD to 7-Segment Decoder

3. Logic Minimization Using K-Maps:

K-map for a

CD \ AB	00	01	11	10
00	1		1	1
01		1	1	1
11				
10	1	1		

K-map for b

CD \ AB	00	01	11	10
00	1	1	1	1
01	1		1	
11				
10	1	1		

K-map for c

CD \ AB	00	01	11	10
00	1	1	1	
01	1	1	1	1
11				
10	1	1		

$$a = A'C + A'BD + AB'C' + B'C'D'$$

$$b = A'B' + B'C' + A'C'D' + A'CD$$

$$c = A'B + B'C' + A'D$$

Extracting common terms

$$\text{Let } T_1 = A'B, T_2 = B'C', T_3 = A'D$$

Optimized Logic Expressions

$$a = A'C + T_1 D + T_2 A + T_2 D'$$

$$b = A'B' + T_2 + A'C'D' + T_3 C$$

$$c = T_1 + T_2 + T_3$$

T_1, T_2, T_3 are **shared gates**

Designing a BCD to 7-Segment Decoder

3. Logic Minimization Using K-Maps

K-map for d

CD \ AB	00	01	11	10
00	1		1	1
01		1		1
11				
10	1	1		

K-map for e

CD \ AB	00	01	11	10
00	1			1
01				1
11				
10	1			

K-map for f

CD \ AB	00	01	11	10
00	1			
01	1	1		1
11				
10	1	1		

K-map for g

CD \ AB	00	01	11	10
00			1	1
01	1	1		1
11				
10	1	1		

Common AND Terms

→ Shared Gates

$$T_4 = AB'C', T_5 = B'C'D'$$

$$T_6 = A'B'C, T_7 = A'CD'$$

$$T_8 = A'BC', T_9 = A'BD'$$

Optimized Logic Expressions

$$d = T_4 + T_5 + T_6 + T_7 + T_8 + D$$

$$e = T_5 + T_7$$

$$f = T_4 + T_5 + T_8 + T_9$$

$$g = T_4 + T_6 + T_8 + T_9$$

Designing a BCD to 7-Segment Decoder

4. Technology Mapping:

Many Common AND terms: T_0 thru T_9

$$T_0 = A'C, T_1 = A'B, T_2 = B'C'$$

$$T_3 = A'D, T_4 = AB'C', T_5 = B'C'D'$$

$$T_6 = A'B'C, T_7 = A'CD'$$

$$T_8 = A'BC', T_9 = A'BD'$$

Optimized Logic Expressions

$$a = T_0 + T_1 D + T_4 + T_5$$

$$b = A'B' + T_2 + A'C'D' + T_3 C$$

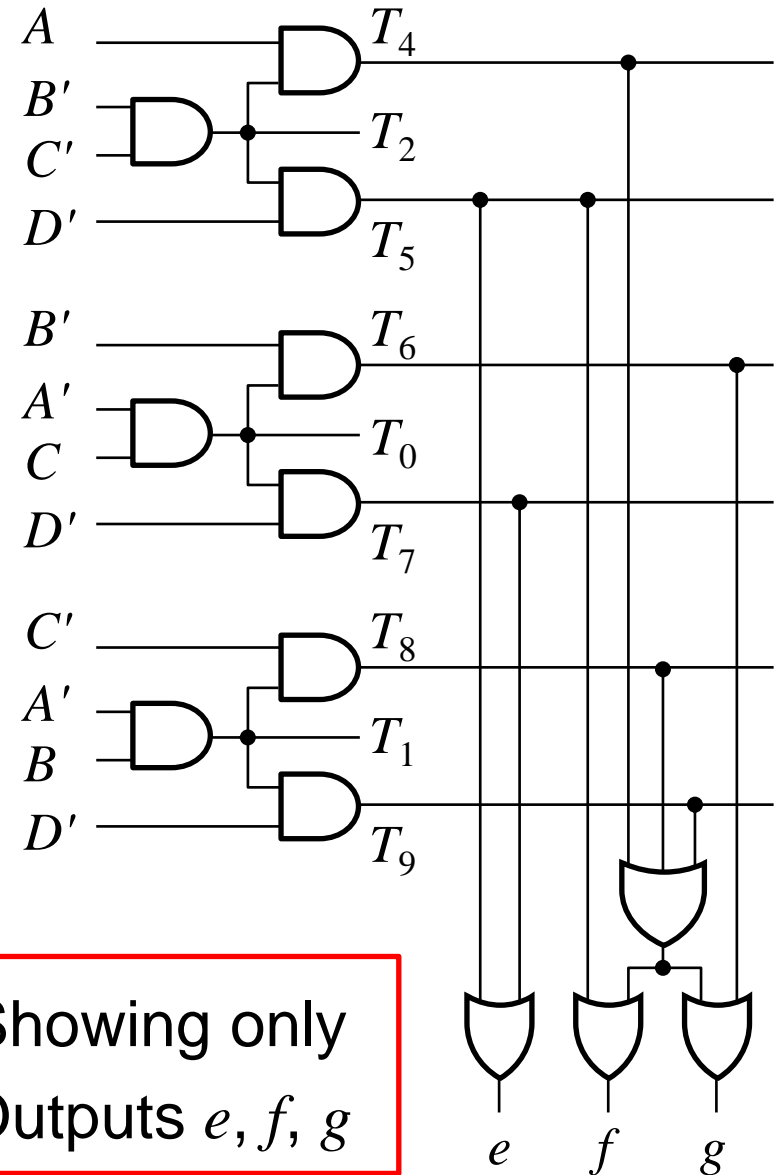
$$c = T_1 + T_2 + T_3$$

$$d = T_4 + T_5 + T_6 + T_7 + T_8 D$$

$$e = T_5 + T_7$$

$$f = T_4 + T_5 + T_8 + T_9$$

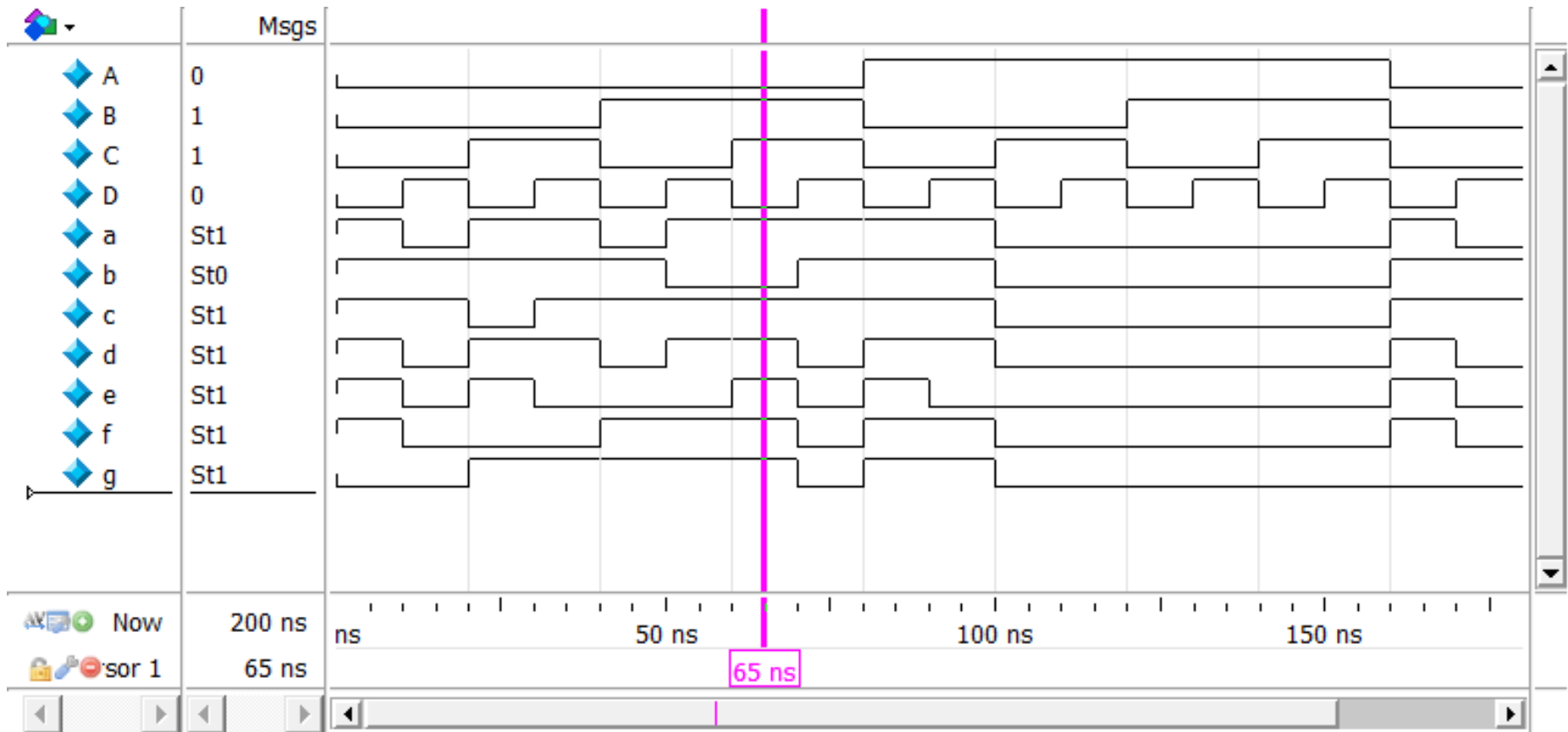
$$g = T_4 + T_6 + T_8 + T_9$$



Designing a BCD to 7-Segment Decoder

5. Verification:

Run the simulation of the circuit. All sixteen input test cases of A, B, C, D are generated between $t=0$ and $t=160\text{ns}$. Verify that outputs a to g match the truth table.



Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ **Binary Adder-Subtractor**
 - ✧ **Half Adder and Full Adder**
 - ✧ **Binary Adder (Ripple Carry Adder and Carry Lookahead Adder)**
 - ✧ **Incrementor**
 - ✧ **Binary Subtractor**
 - ✧ **Adder/Subtractor Design Examples**
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers

Hierarchical Design

❖ Why Hierarchical Design?

To simplify the implementation of a complex circuit

❖ What is Hierarchical Design?

Decompose a complex circuit into smaller pieces called **blocks**

Decompose each block into even smaller blocks

Repeat as necessary until the blocks are small enough

Any block not decomposed is called a **primitive block**

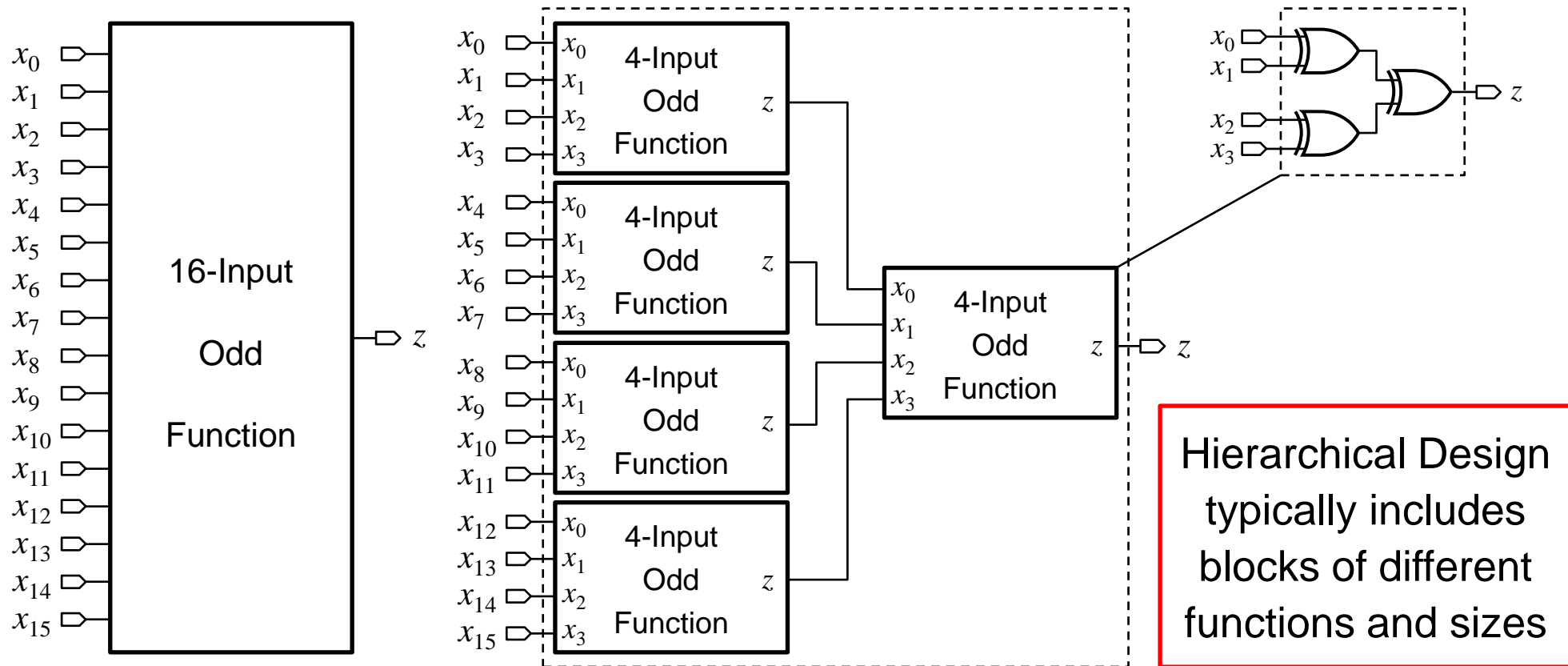
The hierarchy is a tree of blocks at different levels

❖ The blocks are verified and well-document

❖ They are placed in a library for future use

Example of Hierarchical Design

- ❖ Top Level: 16-input odd function: 16 inputs, one output
 - ✧ Implemented using Five 4-input odd functions
- ❖ Second Level: 4-input odd function that uses three XOR gates



Hierarchical Design typically includes blocks of different functions and sizes

Testing Hierarchical Design

- ❖ Exhaustive testing can be very time consuming (or impossible)
 - ✧ For a 16-bit input, there are $2^{16} = 65,536$ test cases (combinations)
 - ✧ For a 32-bit input, there are $2^{32} = 4,294,967,296$ test cases
 - ✧ For a 64-bit input, there are $2^{64} = 18,446,744,073,709,551,616$ test cases!
- ❖ Testing a hierarchical design requires a different strategy
- ❖ Test each block in the hierarchy separately
 - ✧ For smaller blocks, exhaustive testing can be done
 - ✧ It is easier to detect errors in smaller blocks before testing complete circuit
- ❖ Test the top-level design by applying selected test inputs
- ❖ Make sure that the test inputs exercise all parts of the circuit

Top-Down versus Bottom-Up Design

- ❖ A **top-down design** proceeds from a high-level specification to a more and more detailed design by decomposition and successive refinement
- ❖ A **bottom-up design** starts with detailed primitive blocks and combines them into larger and more complex functional blocks
- ❖ Design usually proceeds top-down to a known set of building blocks, ranging from complete processors to primitive logic gates

Half Adder

❖ Half-adder adds 2 bits: x and y

❖ Two output bits:

1. Carry bit: C

2. Sum bit: S

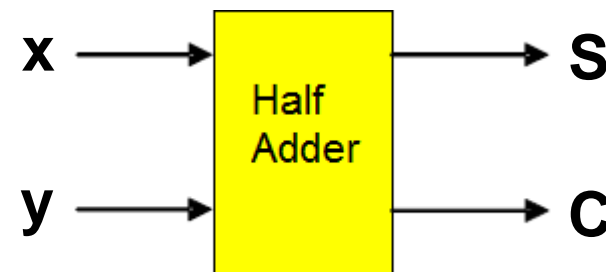
$$\begin{array}{r} x \\ + y \\ \hline C \ S \end{array}$$

❖ Sum bit is 1 if the number of 1's in the input is odd (odd function)

$$S = x'y + xy' = x \oplus y$$

❖ Carry bit is 1 only when both inputs are 1

$$C = x y$$

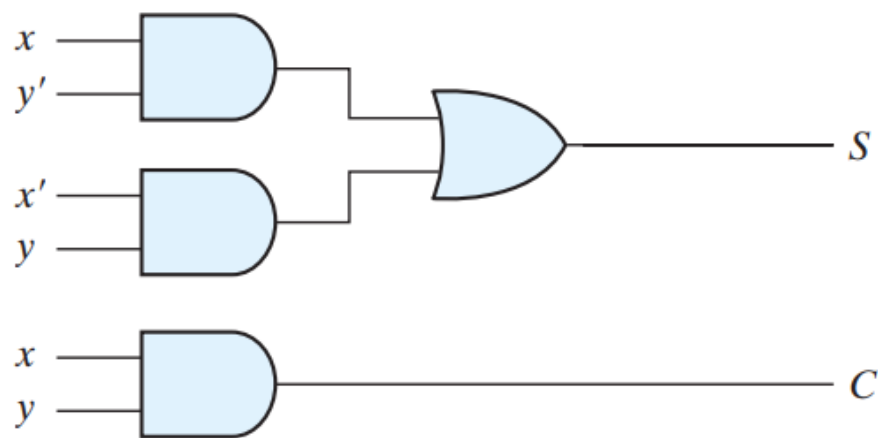


Truth Table

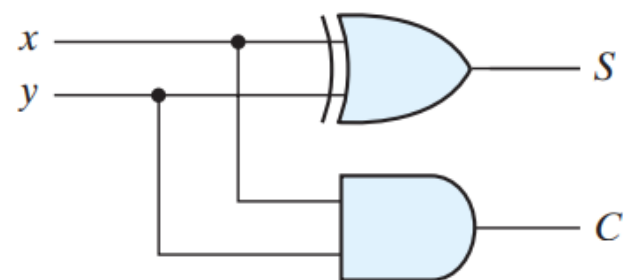
x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half Adder

- ❖ The logic diagram of the half adder implemented in sum-of-products is shown in (a). It can be also implemented with an exclusive-OR and an AND gate as shown in (b):



$$(a) \begin{aligned} S &= xy' + x'y \\ C &= xy \end{aligned}$$



$$(b) \begin{aligned} S &= x \oplus y \\ C &= xy \end{aligned}$$

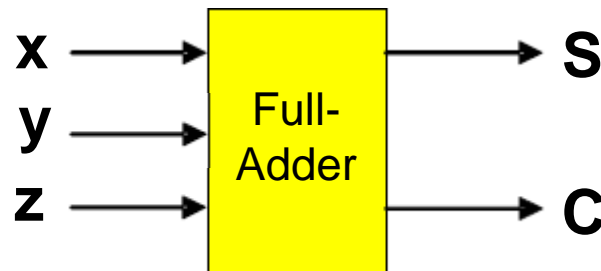
Full Adder

❖ Full adder adds 3 bits: **x**, **y**, and **z**

❖ Two output bits:

1. Carry bit: **C**

2. Sum bit: **S**



❖ Sum bit is 1 if the number of 1's in the input is odd (odd function)

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

❖ Carry bit is 1 if the number of 1's in the input is 2 or 3

$$C = xy + xz + yz$$

Truth Table

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

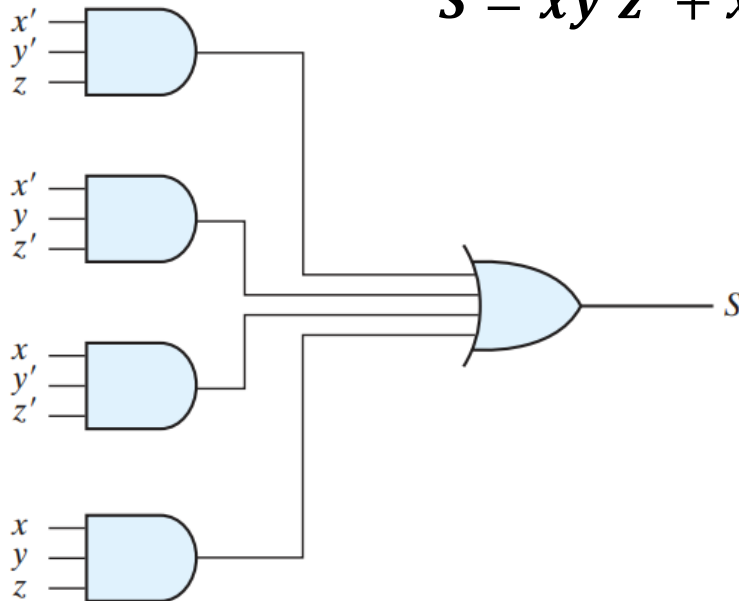
Full Adder

- ❖ The logic diagram for the full adder implemented in sum-of-products form:

K-Map of S

	yz	00	01	11	10
x	0	0	1	0	1
	1	1	0	1	0

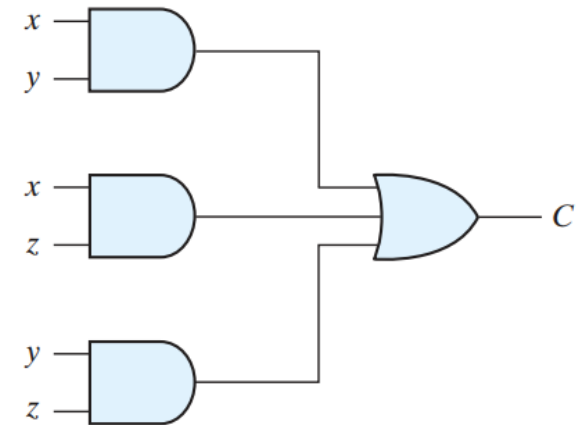
$$S = xy'z' + x'yz' + x'y'z + xyz$$



K-Map of C

	yz	00	01	11	10
x	0	0	0	1	0
	1	0	1	1	1

$$C = xy + xz + yz$$

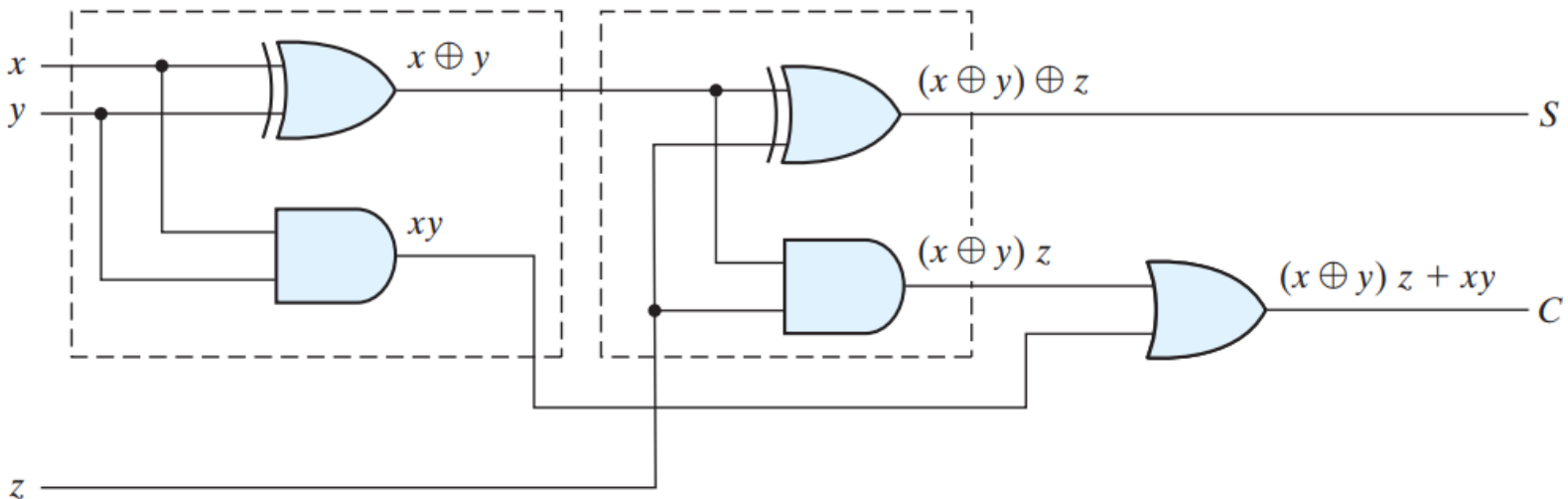


Full Adder

- ❖ Full adder can also be implemented with **two half adders** and **one OR gate**:

$$\begin{aligned} S &= xy'z' + x'yz' + x'y'z + xyz \\ &= z'(xy' + x'y) + z(x'y' + xy) \\ &= z'(x \oplus y) + z(x \oplus y)' \\ &= x \oplus y \oplus z = (x \oplus y) \oplus z \end{aligned}$$

$$\begin{aligned} C &= xy + xz + yz \\ &= xy + (x \oplus y)z \end{aligned}$$



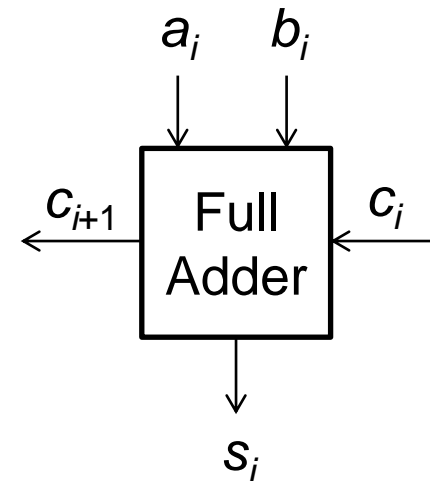
Binary Adder (Ripple Carry Adder)

- ❖ Start with the least significant bit (rightmost bit)
- ❖ Add each pair of bits
- ❖ Include the carry in the addition

carry		1	1	1	1				
	0	0	1	1	0	1	1	0	(54)
+	0	0	0	1	1	1	0	1	(29)
<hr/>									
	0	1	0	1	0	0	1	1	(83)
bit position:	7	6	5	4	3	2	1	0	

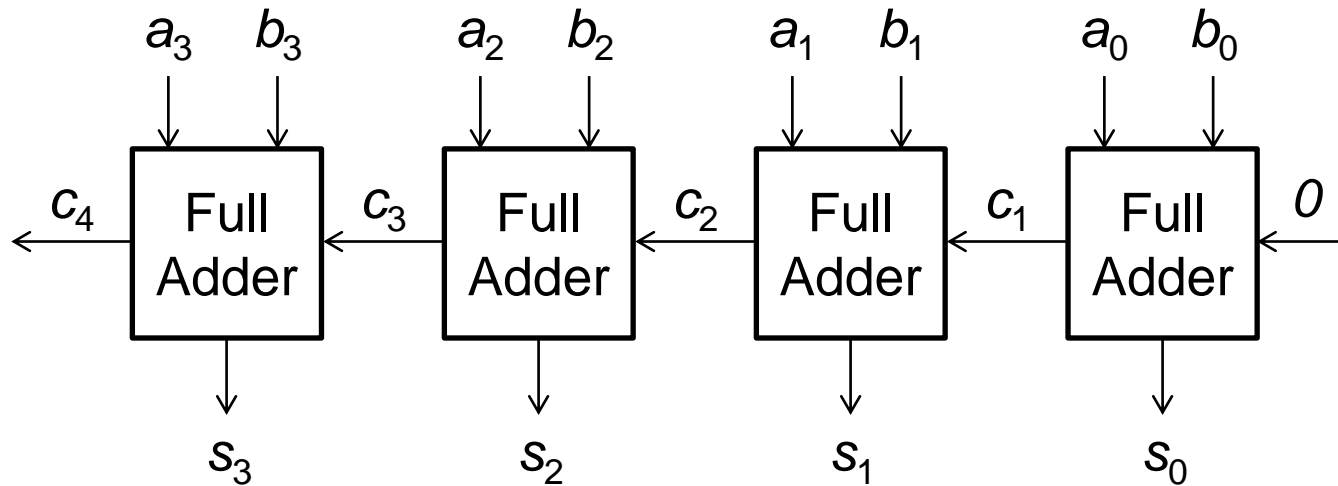
Iterative Design: Ripple Carry Adder

- ❖ Using **identical copies** of a smaller circuit to build a large circuit
- ❖ Addition of n -bit numbers requires:
 - ✧ A chain of n full adders, or
 - ✧ A chain of one-half adder and $(n - 1)$ full adders
- ❖ Example: Building a 4-bit adder using 4 copies of a full adder
 - ✧ The **cell** (iterative block) is a **full adder**
 - Adds 3 bits: a_i , b_i , c_i
 - Computes: Sum s_i and Carry-out c_{i+1}

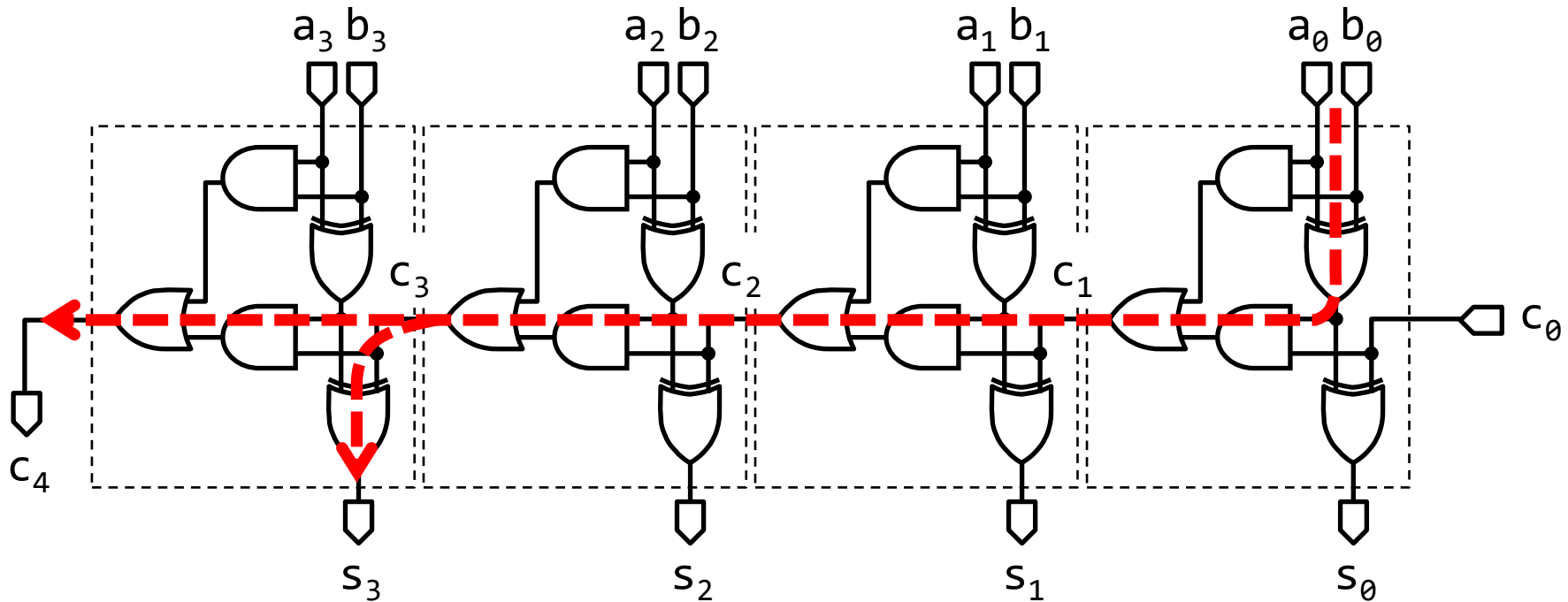


Iterative Design: Ripple Carry Adder

- ❖ The Figure below shows the interconnection of four full-adder (FA) circuits to provide a four-bit binary ripple carry adder
 - ✧ Carry-out of cell i becomes carry-in to cell $(i+1)$
 - ✧ The input carry to the least significant position is fixed at 0

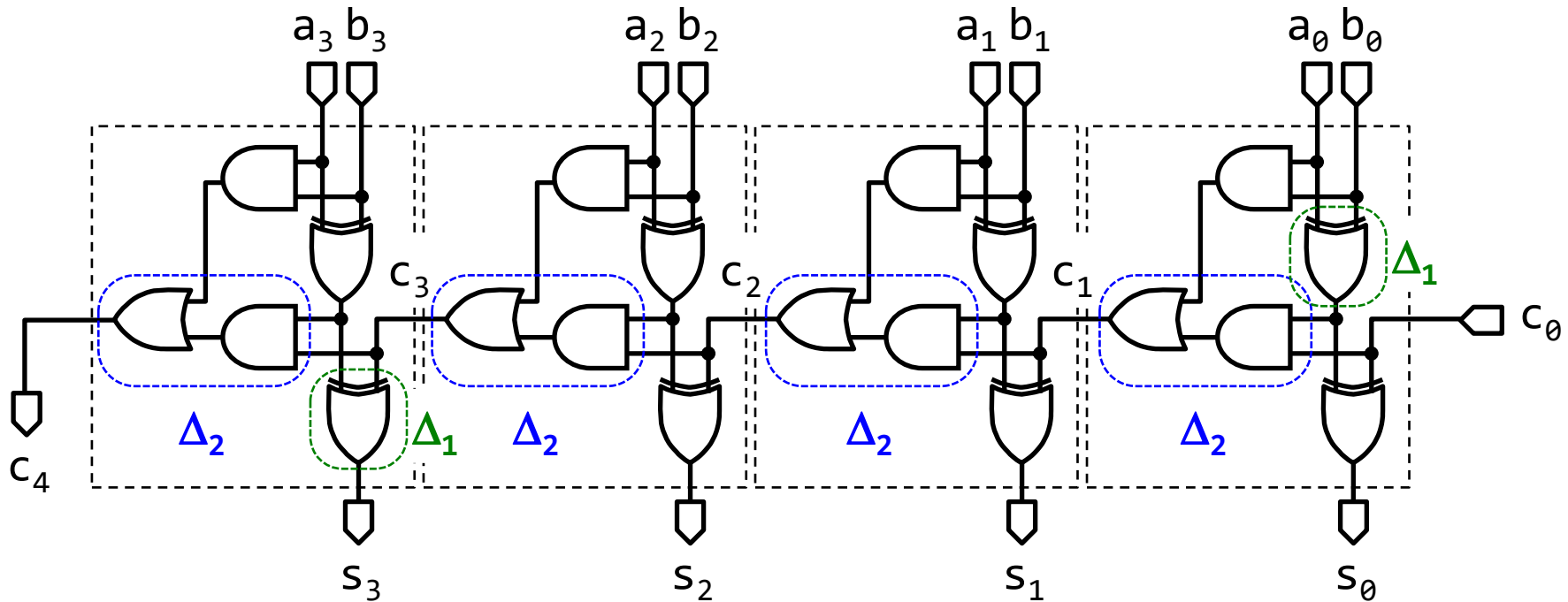


Carry Propagation



- ❖ Major drawback of ripple-carry adder is the **carry propagation**
- ❖ The carries are connected in a chain through the full adders
- ❖ The **carry ripples** (propagates) through all the full adders
- ❖ This is why it is called a **ripple-carry adder**

Longest Delay Analysis



- ❖ Suppose the **XOR** delay is Δ_1 (Delay of XOR > Delay of AND) and **AND-OR** delay is Δ_2
- ❖ For an N -bit ripple-carry adder, if all inputs are present at once:
 1. Most-significant sum-bit delay = $2\Delta_1 + (N - 1) \Delta_2$
 2. Final Carry-out delay = $\Delta_1 + N \Delta_2$

Carry Lookahead Adder

❖ Is it possible to eliminate carry propagation?

❖ Observation: $c_{i+1} = a_i b_i + (a_i \oplus b_i) c_i$

❖ If both inputs a_i and b_i are 1s then c_{i+1} will be 1 regardless of input c_i

❖ Therefore, define $g_i = a_i b_i$

✧ g_i is called **carry generate**: generates c_{i+1} regardless of c_i

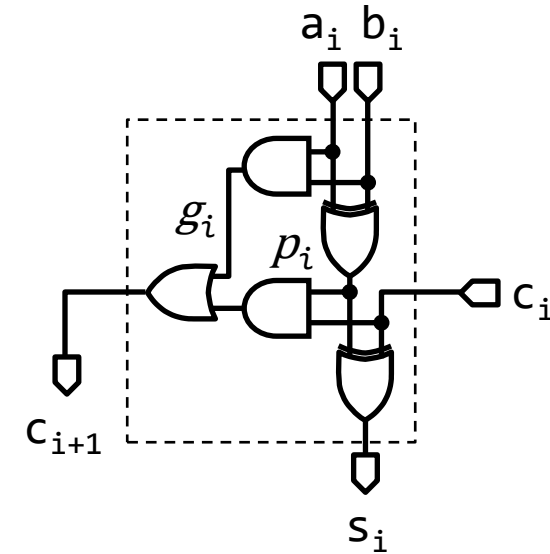
❖ In addition, define $p_i = (a_i \oplus b_i)$ a_i or b_i is 1, not both

✧ p_i is called **carry propagate**: propagates value of c_i to c_{i+1}

❖ Equation of output sum carry becomes:

$$s_i = p_i \oplus c_i \quad \text{and} \quad c_{i+1} = g_i + p_i c_i$$

✧ If both inputs a_i and b_i are 0s then $g_i = p_i = 0$ and $c_{i+1} = 0$



Carry Bits

Carry bits are generated by a **Lookahead Carry Unit** as follows:

c_0 = input carry

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2 c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

$$c_4 = g_3 + p_3 c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

Define **Group Generate**: $GG = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$

Define **Group Propagate**: $GP = p_3 p_2 p_1 p_0$

$$c_4 = GG + GP c_0$$

Carry does not ripple anymore

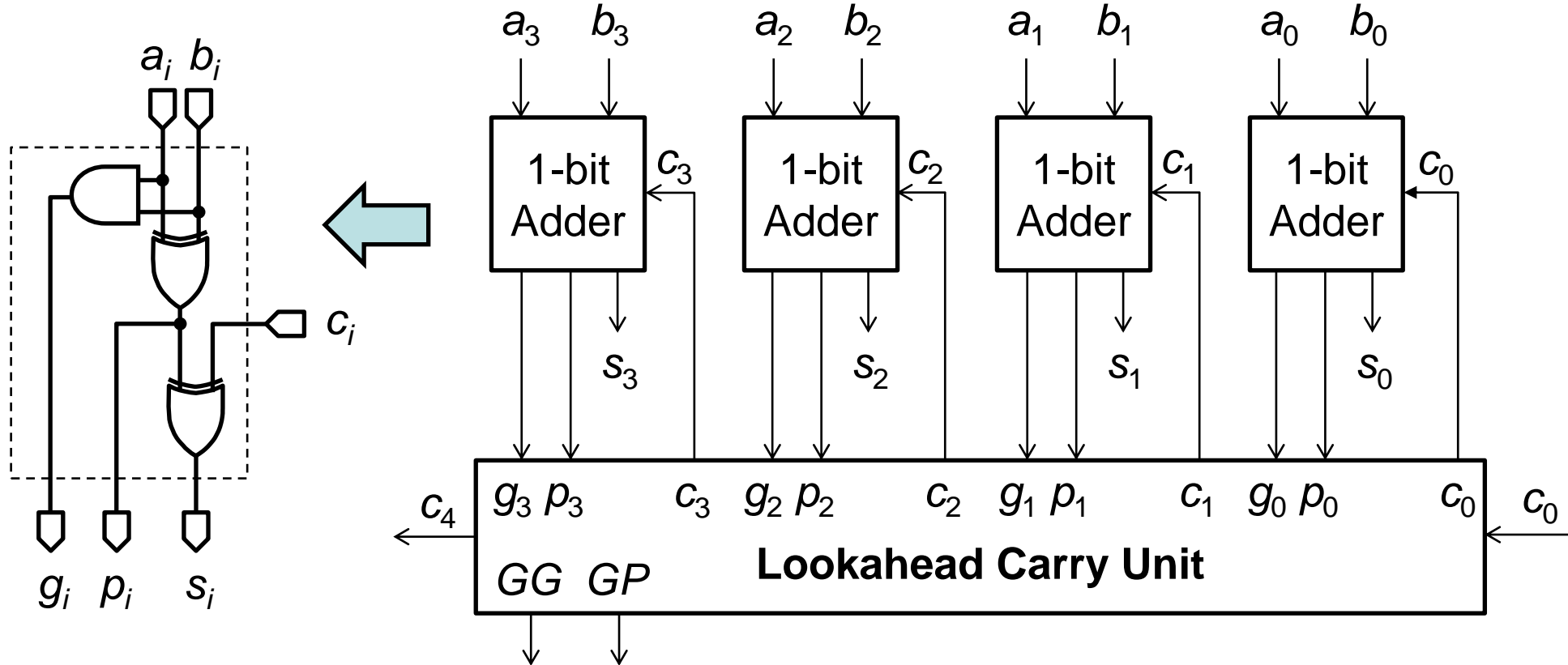
Reduced delay when generating c_1 to c_4 in parallel

4-Bit Carry Lookahead Adder

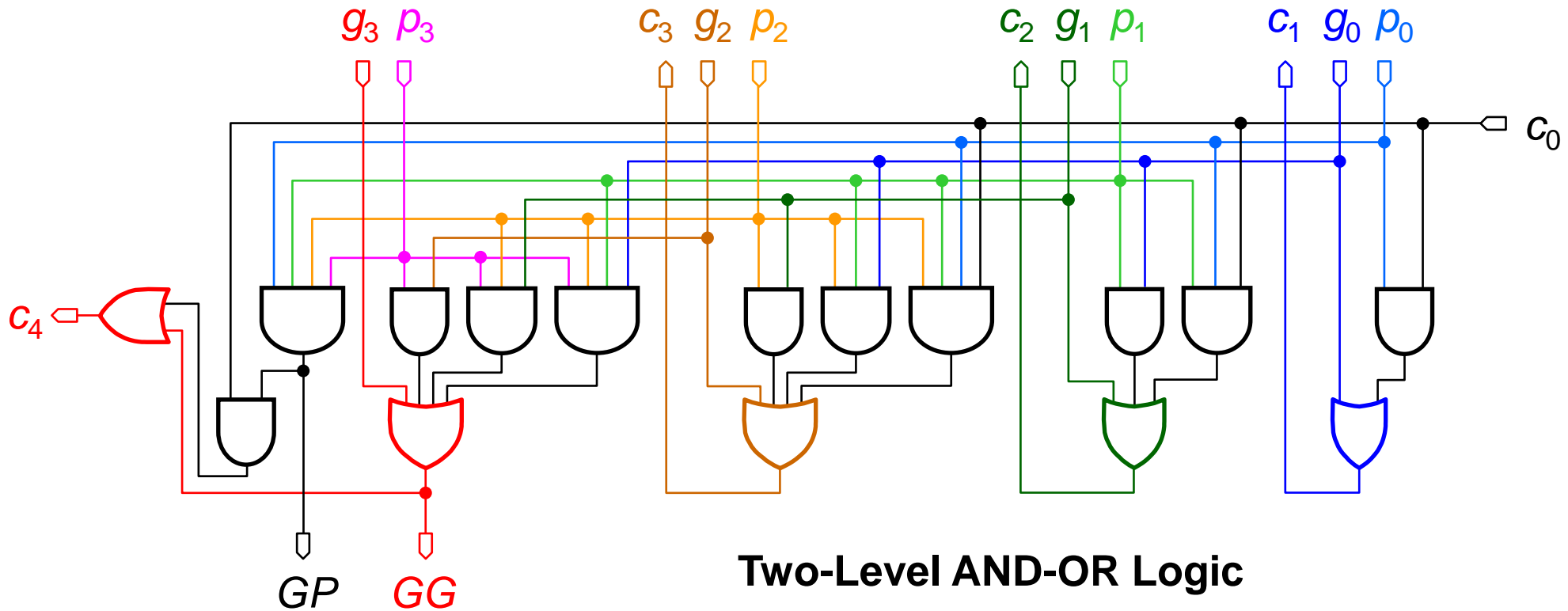
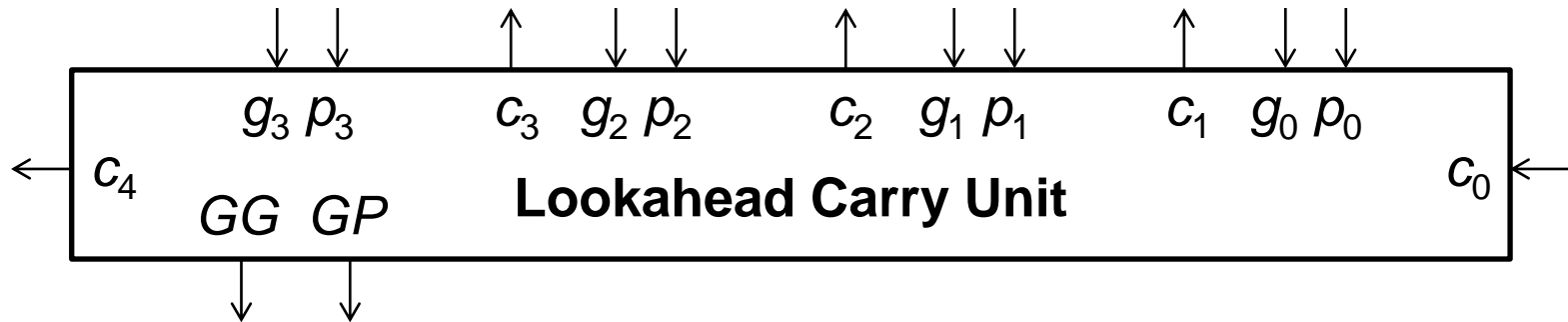
All **generate** and **propagate** signals (g_i, p_i) are generated in parallel

All carry bits (c_1 to c_4) are generated in parallel

The sum bits are generated faster than ripple-carry adder

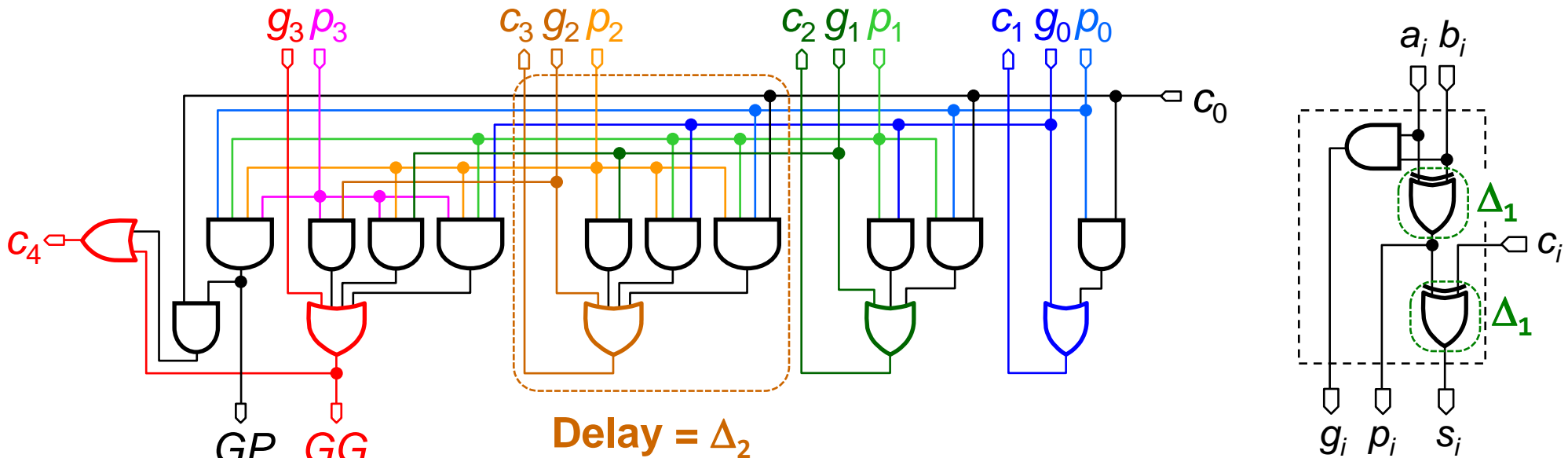


Lookahead Carry Unit



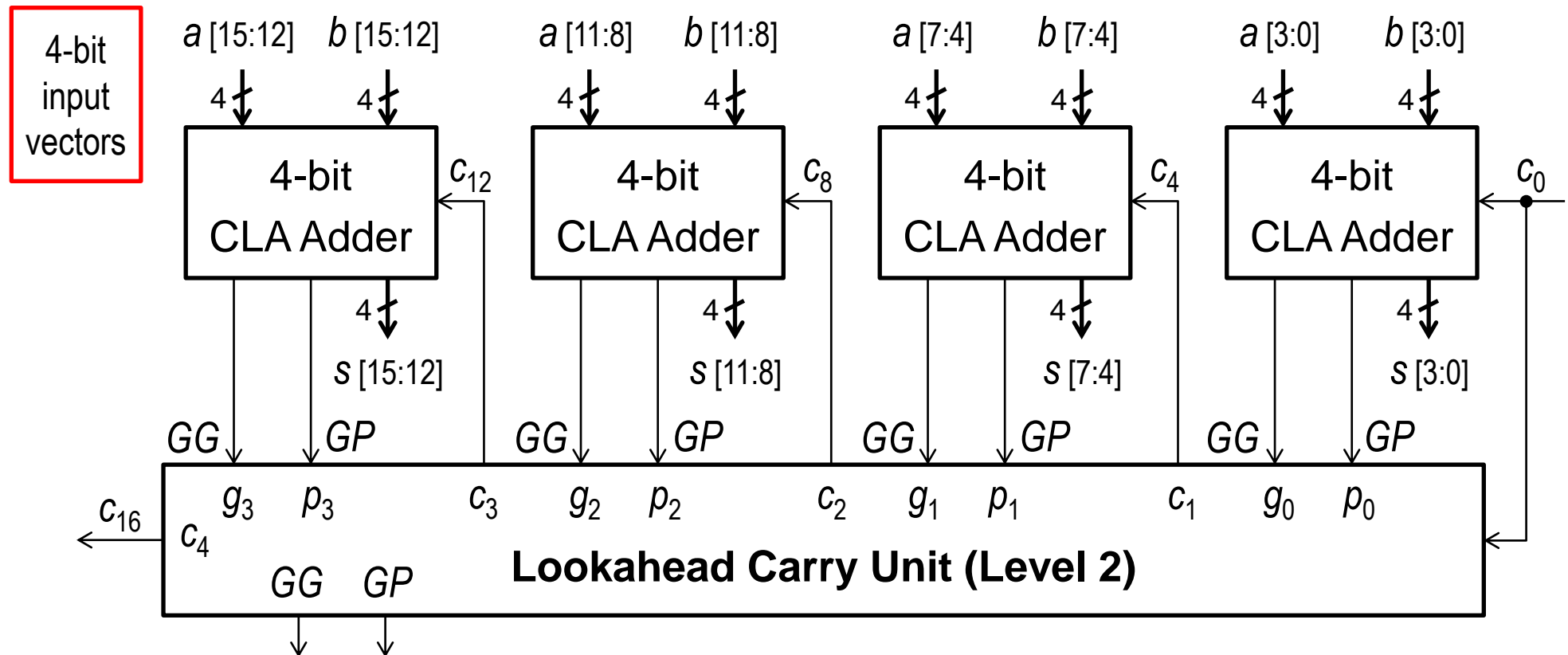
Longest Delay of the 4-bit CLA

- ❖ All generate and propagate signals are produced in parallel
- ❖ Delay of all g_i and $p_i = \Delta_1$ (Delay of XOR > Delay of AND)
- ❖ Carry bits c_1, c_2 , and c_3 are generated in parallel (Delay = Δ_2)
 - ✧ Carry-out bit c_4 is not needed to compute the sum bits
- ❖ Longest Delay of the 4-bit CLA = $\Delta_1 + \Delta_2 + \Delta_1 = 2\Delta_1 + \Delta_2$



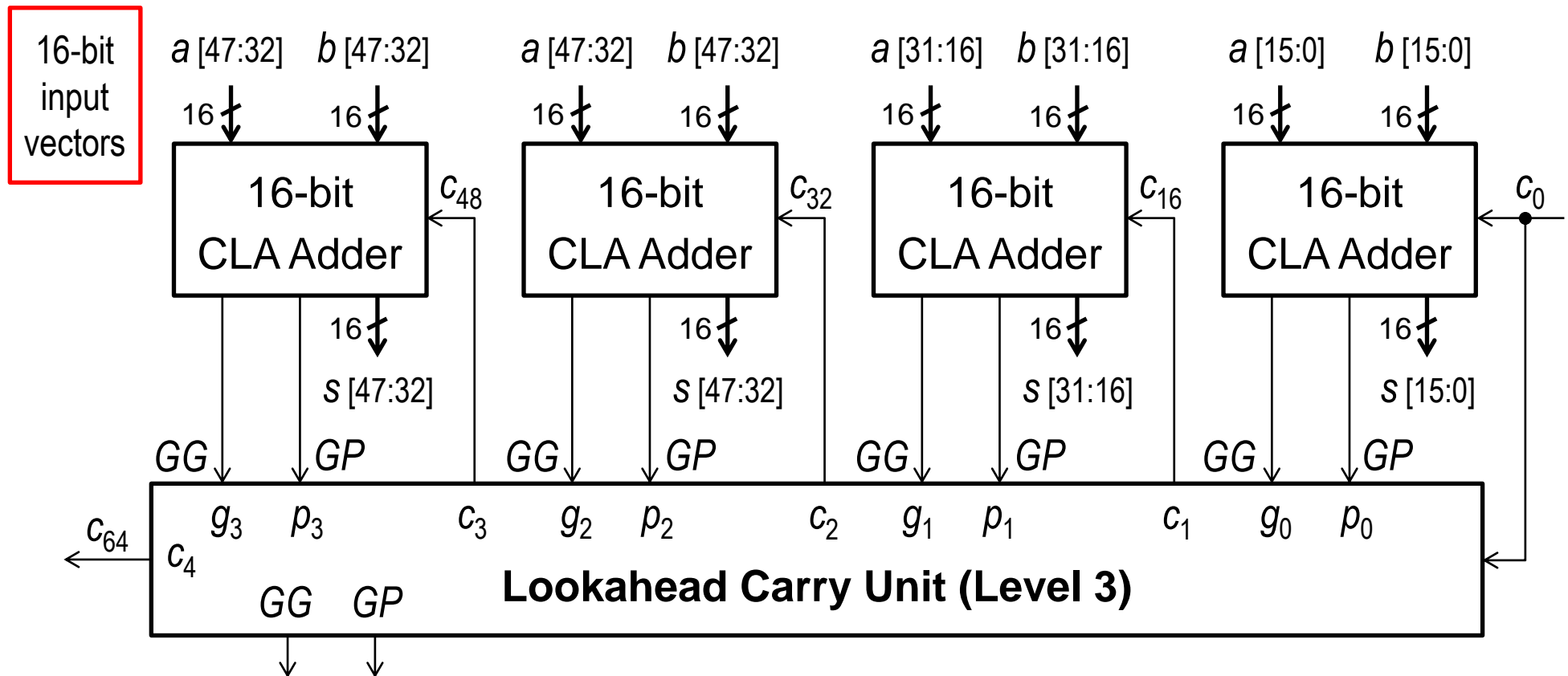
Hierarchical 16-Bit Carry Lookahead Adder

- ❖ Designed with Four 4-bit Carry Lookahead Adders (CLA)
- ❖ A **Second-Level Lookahead Carry Unit** is required
- ❖ Uses **Group Generate** (GG) and **Group Propagate** (GP) signals



Hierarchical 64-Bit Carry Lookahead Adder

- ❖ Designed with Four 16-bit Carry Lookahead Adders (CLA)
- ❖ A **Third-Level Lookahead Carry Unit** is required
- ❖ Uses **Group Generate** (GG) and **Group Propagate** (GP) signals

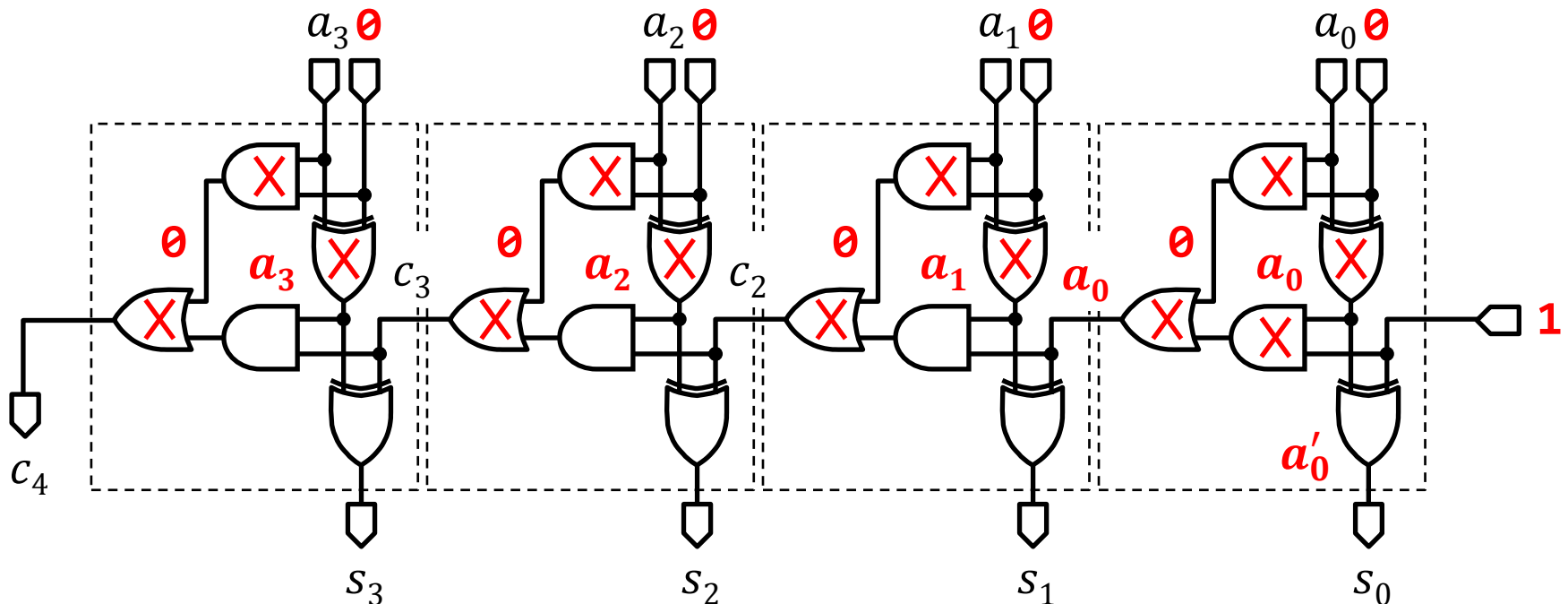


Incrementor Circuit

❖ An incrementer is a special case of an adder

$$Sum = A + 1 \quad (B = \mathbf{0}, C_0 = \mathbf{1})$$

❖ An n -bit Adder can be simplified into an n -bit Incrementer

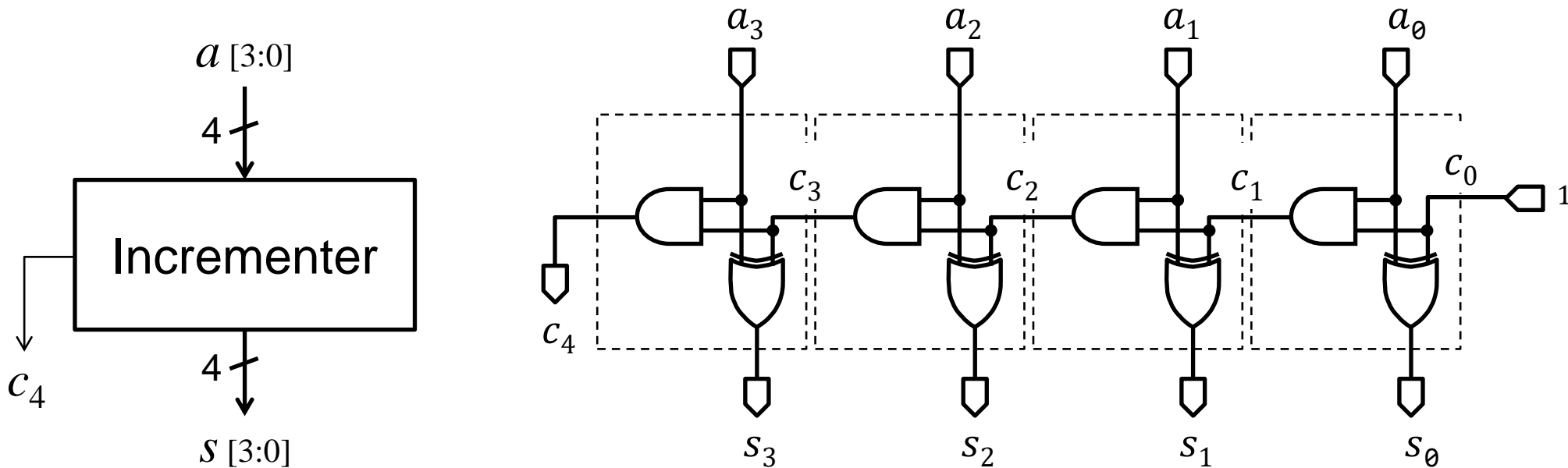


Design by Contraction

- ❖ Contraction is a technique for simplifying the logic
- ❖ Applying 0s and 1s to some inputs
- ❖ Equations are simplified after applying fixed 0 and 1 inputs
- ❖ Converting a function block to a more simplified function
- ❖ Examples of Design by Contraction
 - ✧ Incrementing a number by a fixed constant
 - ✧ Comparing a number to a fixed constant

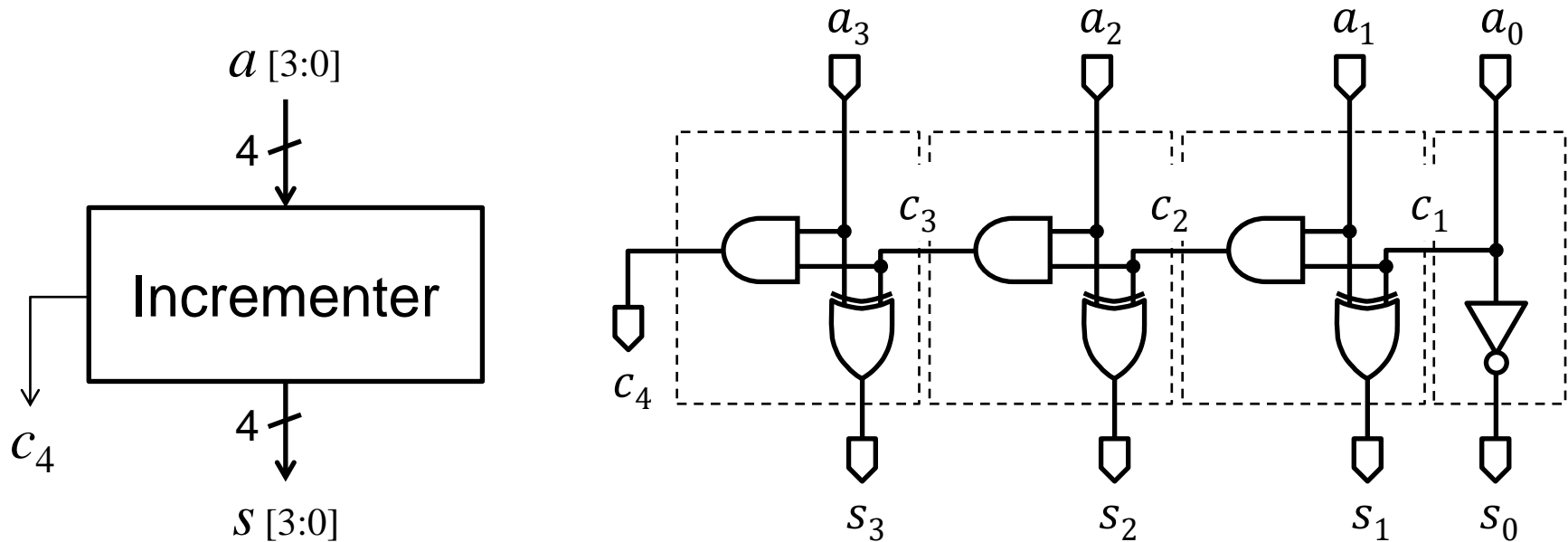
Simplifying the Incrementer Circuit

- ❖ Many gates were eliminated
- ❖ No longer needed when an input is a constant
- ❖ Last cell can be replicated to implement an n -bit incrementer



Simplifying the Incrementer Circuit

- ❖ First half adder can be simplified and replaced with an inverter



Binary Subtractor

❖ When computing $A - B$, convert B to its 2's complement

$$A - B = A + (\text{2's complement of } B)$$

❖ **Same adder** is used for **both addition and subtraction**

This is the biggest advantage of 2's complement

borrow:	-1	-1		-1				carry:	1	1		1	1					
	0	1	0	0	1	1	0	1		0	1	0	0	1	1	0	1	
-	0	0	1	1	1	0	1	0	→	+	1	1	0	0	0	1	1	0
	<hr/>									<hr/>								(2's complement)
	0	0	0	1	0	0	1	1		0	0	0	1	0	0	1	1	
																		(same result)

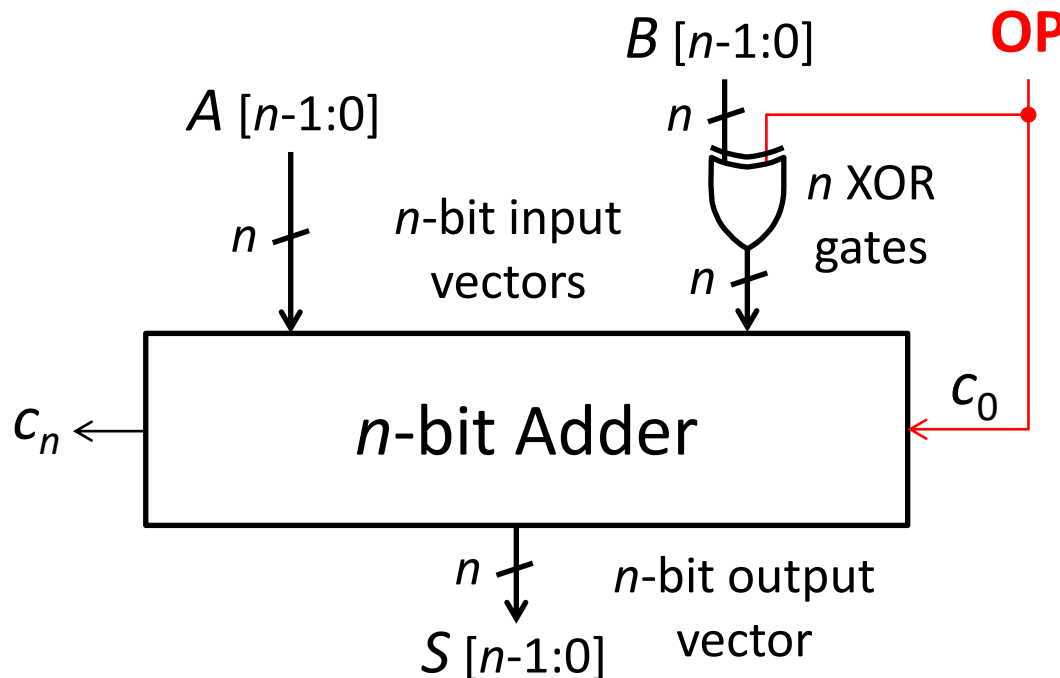
❖ Final carry is **ignored**, because

$$A + (\text{2's complement of } B) = A + (2^n - B) = (A - B) + 2^n$$

Final carry = 2^n , for n -bit numbers

Adder/Subtractor for 2's Complement

- ❖ Same adder is used to compute: $(A + B)$ or $(A - B)$
- ❖ Subtraction $(A - B)$ is computed as: $A + (2\text{'s complement of } B)$
 $2\text{'s complement of } B = (1\text{'s complement of } B) + 1$
- ❖ Two operations: **OP = 0 (ADD)**, **OP = 1 (SUBTRACT)**



OP = 0 (ADD)

$$B \text{ XOR } 0 = B$$

$$S = A + B + 0 = A + B$$

OP = 1 (SUBTRACT)

$$B \text{ XOR } 1 = 1\text{'s complement of } B$$

$$S = A + (1\text{'s complement of } B) + 1$$

$$S = A + (2\text{'s complement of } B)$$

$$S = A - B$$

Carry versus Overflow

❖ Carry is important when ...

- ✧ Adding **unsigned integers**
- ✧ Indicates that the **unsigned sum** is out of range
- ✧ $\text{Sum} > \text{maximum unsigned } n\text{-bit value}$

❖ Overflow is important when ...

- ✧ Adding or subtracting **signed integers**
- ✧ Indicates that the **signed sum** is out of range

❖ Overflow occurs when ...

- ✧ Adding two positive numbers and the sum is negative
- ✧ Adding two negative numbers and the sum is positive

❖ Simplest way to detect Overflow: $V = C_{n-1} \oplus C_n$

- ✧ C_{n-1} and C_n are the carry-in and carry-out of the most-significant bit

Carry and Overflow Examples

- ❖ We can have carry without overflow and vice-versa
- ❖ Four cases are possible (Examples on 8-bit numbers)

				1				
	0	0	0	0	1	1	1	1
15								
+	0	0	0	0	1	0	0	0
8								
<hr/>								
	0	0	0	1	0	1	1	1
23								
Carry = 0 Overflow = 0								

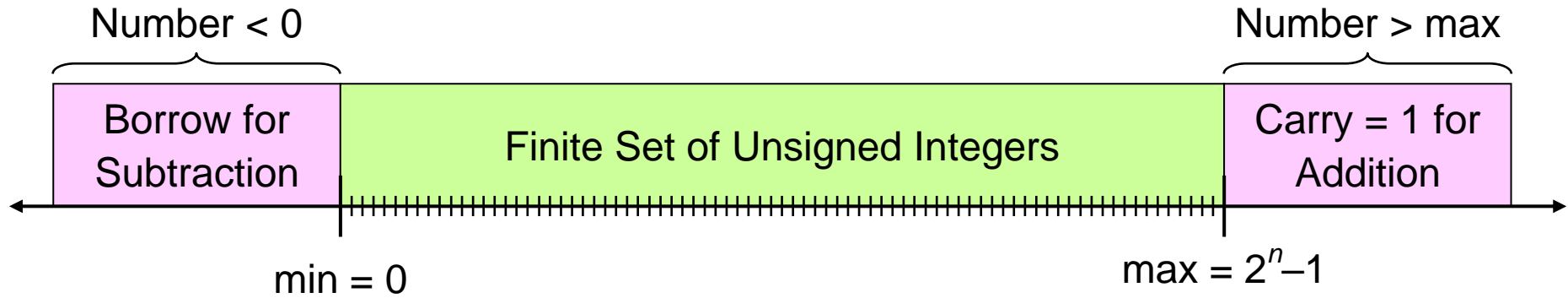
1	1	1	1	1				
	0	0	0	0	1	1	1	1
								15
+	1	1	1	1	1	0	0	0
								248 (-8)
<hr/>								
	0	0	0	0	0	1	1	1
								7
Carry = 1 Overflow = 0								

				1				
	0	1	0	0	1	1	1	1
								79
+	0	1	0	0	0	0	0	0
								64
<hr/>								
	1	0	0	0	1	1	1	1
								143 (-113)
Carry = 0 Overflow = 1								

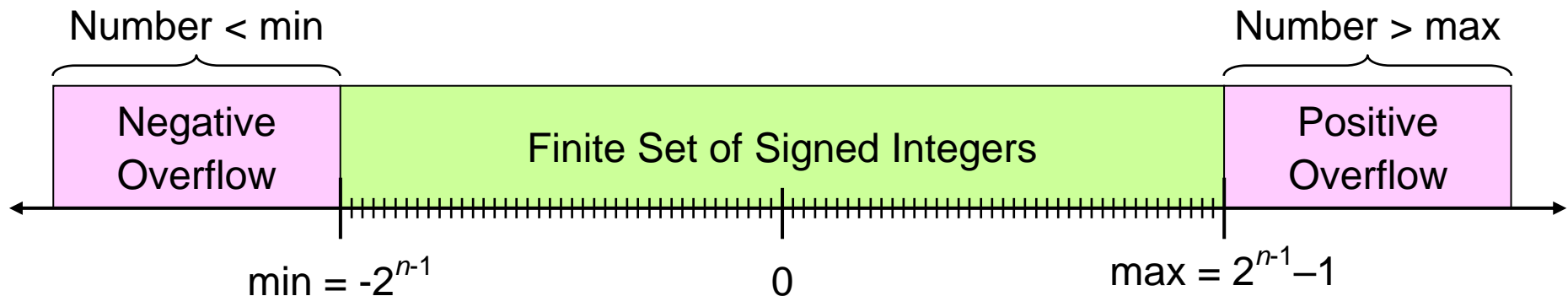
1				1	1			
	1	1	0	1	1	0	1	0
								218 (-38)
+	1	0	0	1	1	1	0	1
								157 (-99)
<hr/>								
	0	1	1	1	0	1	1	1
								119
Carry = 1 Overflow = 1								

Range, Carry, Borrow, and Overflow

❖ Unsigned Integers: n -bit representation

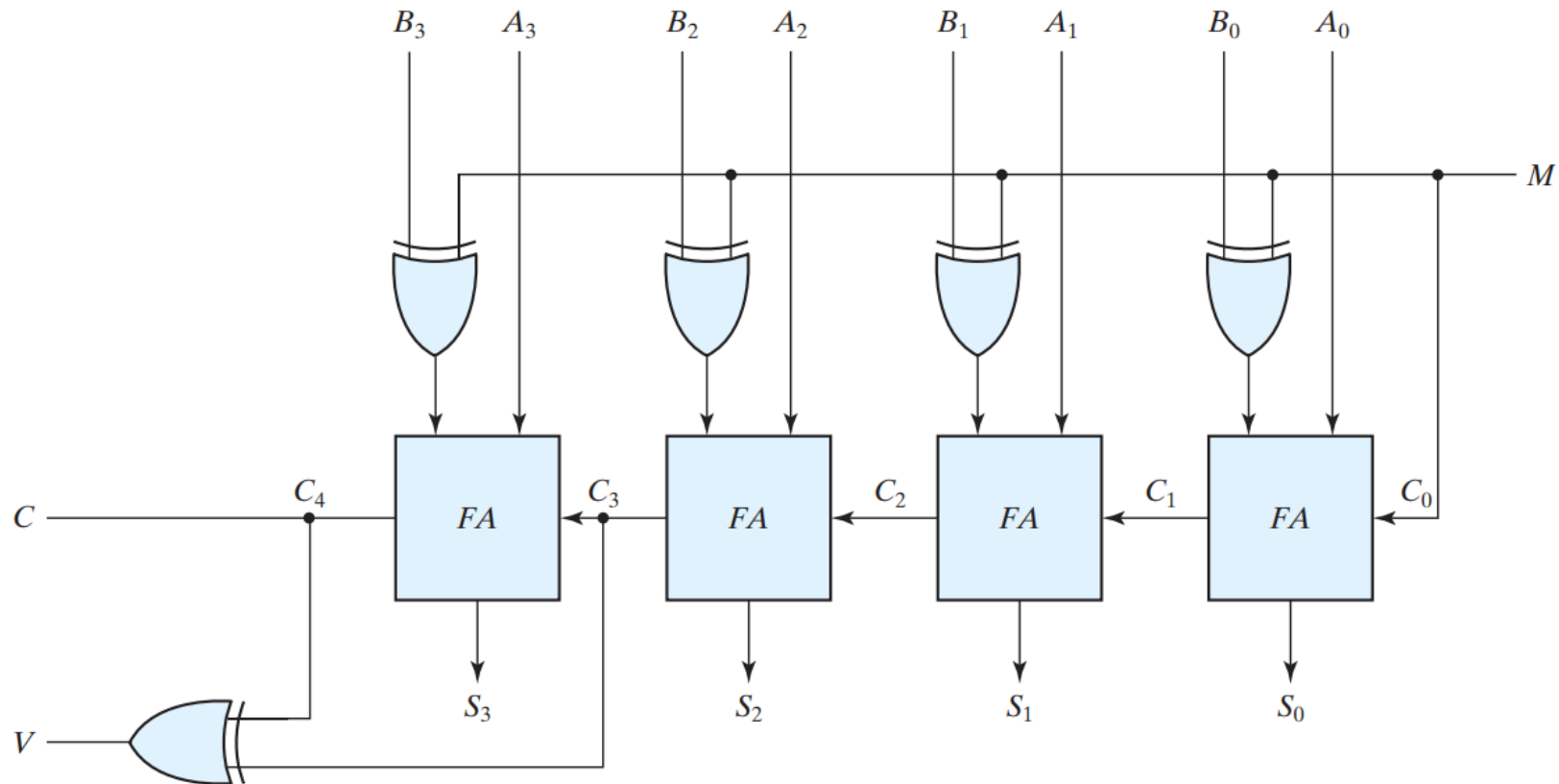


❖ Signed Integers: 2's complement representation



Binary Adder/Subtractor

- ❖ Example: A 4-bit adder/subtractor with carry/overflow detection
 - ✧ Two operations: **M = 0** (**S = A + B**), **M = 1** (**S = A - B**)
 - ✧ The **C** bit detects a carry after addition or a borrow after subtraction
 - ✧ The **V** bit detects an overflow



Zero versus Sign Extension

- ❖ **Unsigned** Integers are **Zero-Extended**
- ❖ **Signed** Integers are **Sign-Extended**
- ❖ Given that X is a 4-bit **unsigned** integer → Range = 0 to 15
- ❖ Given that Y is a 4-bit **signed** integer → Range = -8 to +7
- ❖ If **unsigned** $X = 4'b1101$ (binary), then $X = 13$ (decimal)
- ❖ If **signed** $Y = 4'b1101$ (binary), then $Y = -3$ (decimal)
- ❖ If X is **zero-extended** from 4 to 6 bits then $X = 6'b001101 = 13$
- ❖ If Y is **sign-extended** from 4 to 6 bits then $Y = 6'b111101 = -3$

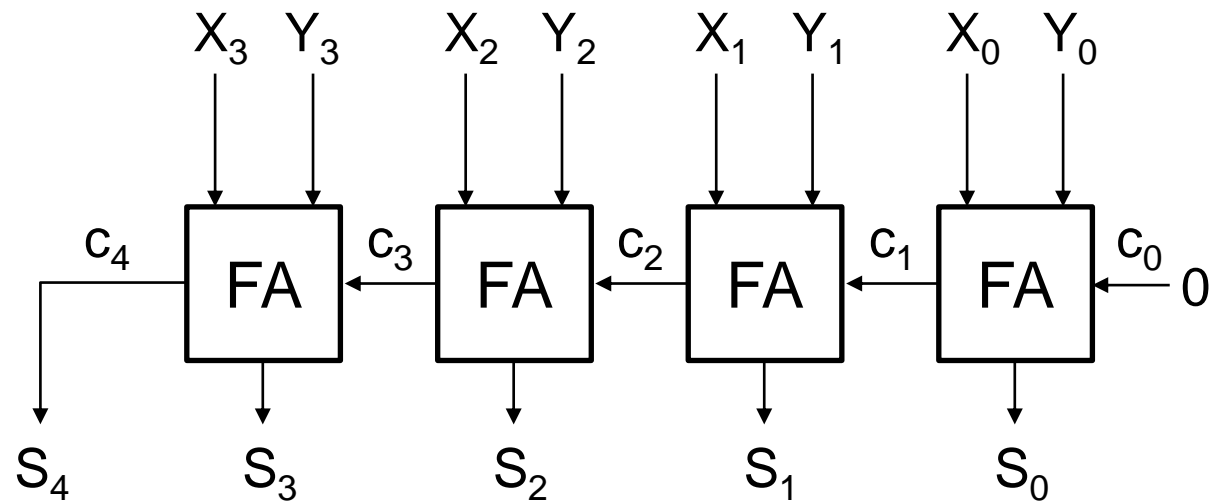
Unsigned Addition $S = X + Y$

- ❖ Design a circuit that computes: $S = X + Y$ (**unsigned X and Y**)
- ❖ $X[3:0]$ and $Y[3:0]$ are 4-bit **unsigned** integers → Range = 0 to 15

Solution:

- ❖ Maximum $S = 15 + 15 = 30$ → unsigned S must be **5 bits**

Most-significant
sum bit S_4 is
the carry bit c_4



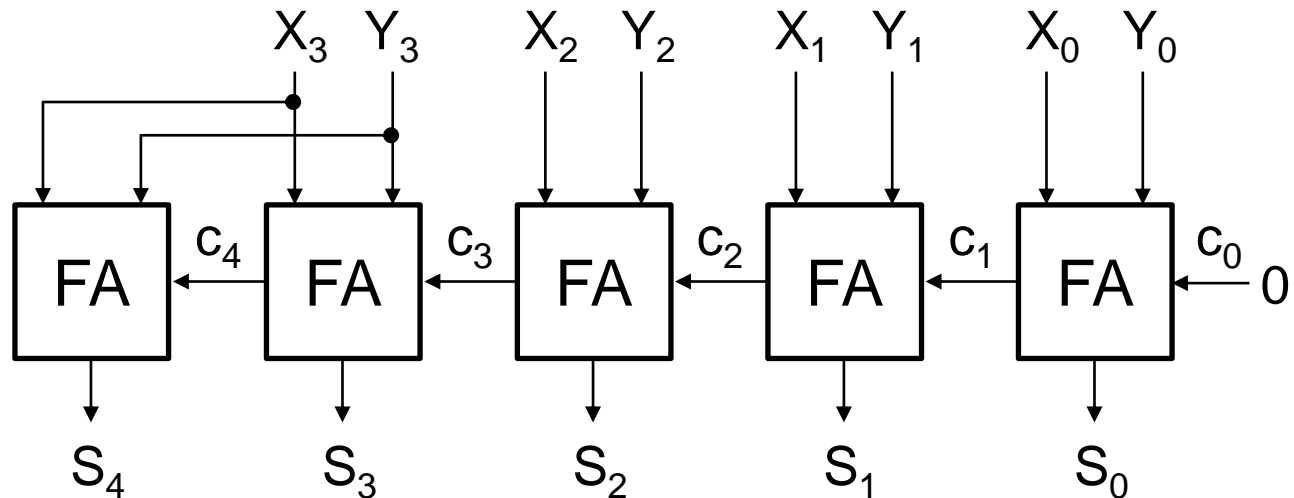
Signed Addition $S = X + Y$

- ❖ Design a circuit that computes: $S = X + Y$ (**signed X and Y**)
- ❖ $X[3:0]$ and $Y[3:0]$ are 4-bit **signed** integers \rightarrow Range = -8 to +7

Solution:

- ❖ Minimum $S = (-8) + (-8) = -16$, Maximum $S = (+7) + (+7) = +14$
- ❖ Therefore, signed range of $S = -16$ to $+14 \rightarrow S$ must be **5 bits**

X and Y are **sign-extended**
 X_3 and Y_3 are replicated to produce S_4



Unsigned Subtraction $S = X - Y$

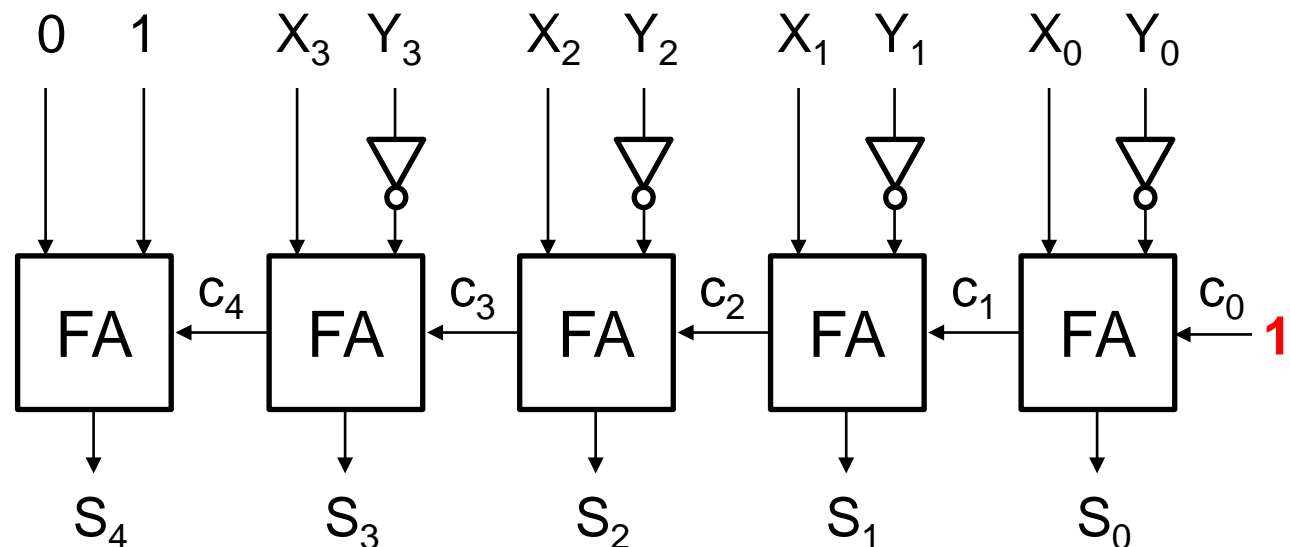
- ❖ Design a circuit that computes $S = X - Y$ (**unsigned X and Y**)
- ❖ $X[3:0]$ and $Y[3:0]$ are 4-bit **unsigned** integers → Range = 0 to 15

Solution: $S = X - Y = X + 2\text{'s complement of } Y = X + Y' + 1$

- ❖ Minimum $S = 0 - 15 = -15$, Maximum $S = 15 - 0 = +15$
- ❖ S is **signed**, even though X and Y are **unsigned** → S is **5 bits**

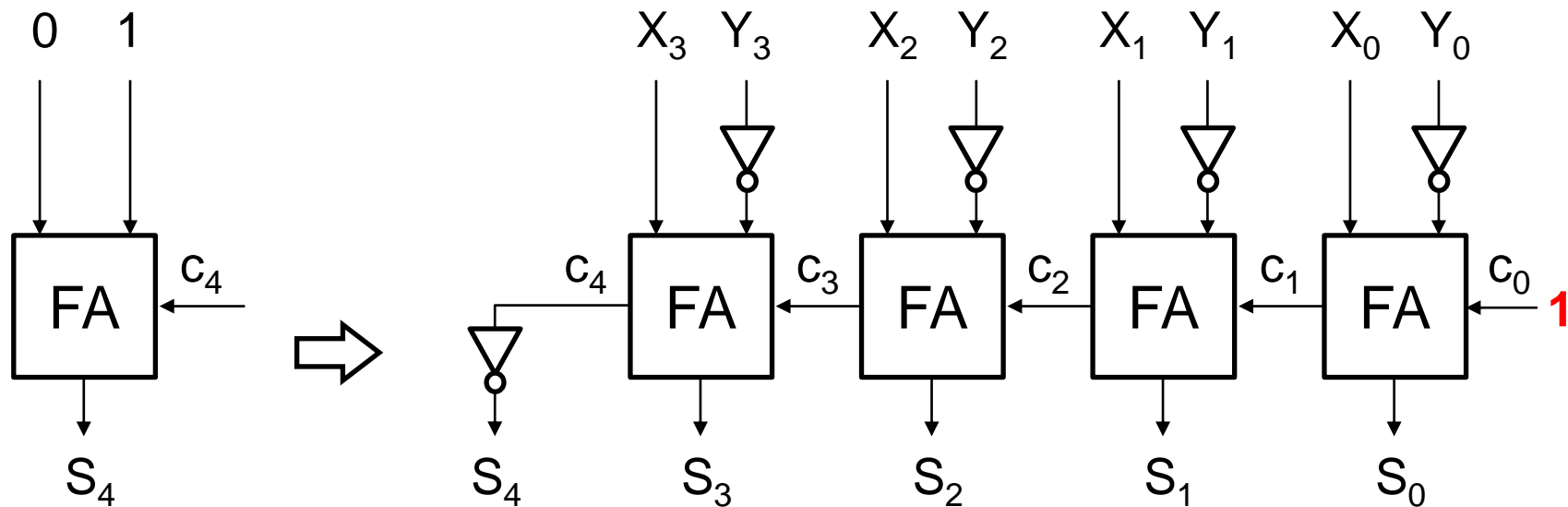
$$X - Y = X + Y' + 1$$

X and Y are
zero-extended.



Unsigned Subtraction $S = X - Y$

- ❖ Most-significant bit: $S_4 = 0 + 0' + c_4 = 1 + c_4 = c_4'$
- ❖ Full Adder for S_4 can be replaced by an **inverter**



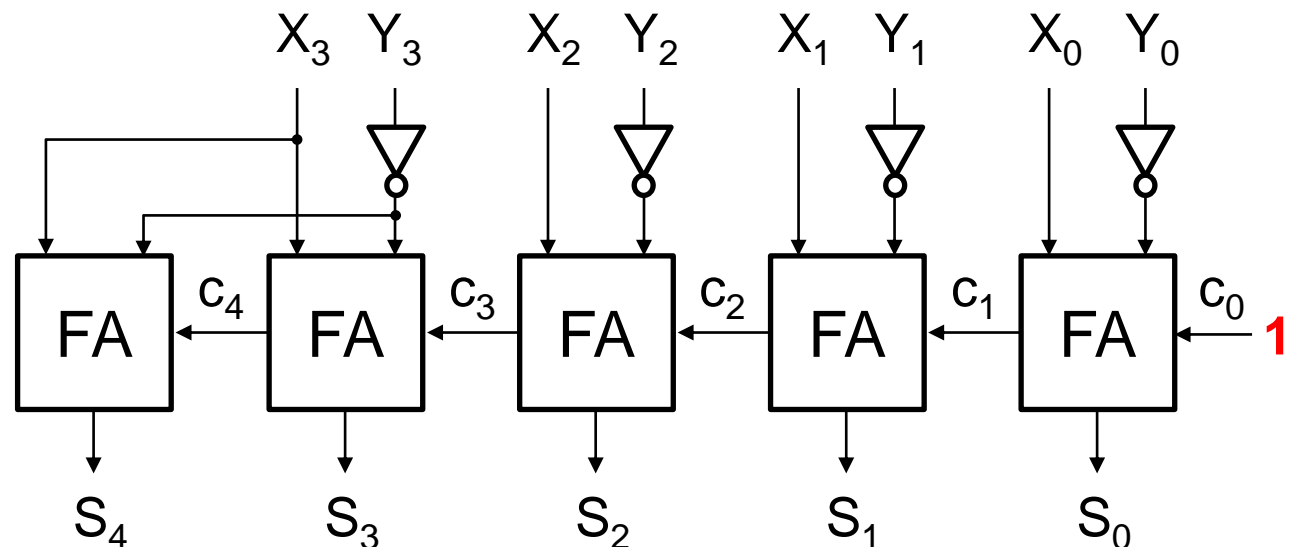
Signed Subtraction $S = X - Y$

- ❖ Design a circuit that computes $S = X - Y$ (**signed X and Y**)
- ❖ $X[3:0]$ and $Y[3:0]$ are 4-bit **signed** integers \rightarrow Range = -8 to +7

Solution: $S = X - Y = X + Y' + 1$

- ❖ Minimum $S = -8 - (+7) = -15$, Maximum $S = +7 - (-8) = +15$
- ❖ Signed range for S is -15 to +15 \rightarrow S is **5 bits**

$X - Y = X + Y' + 1$
X and Y are
sign-extended.

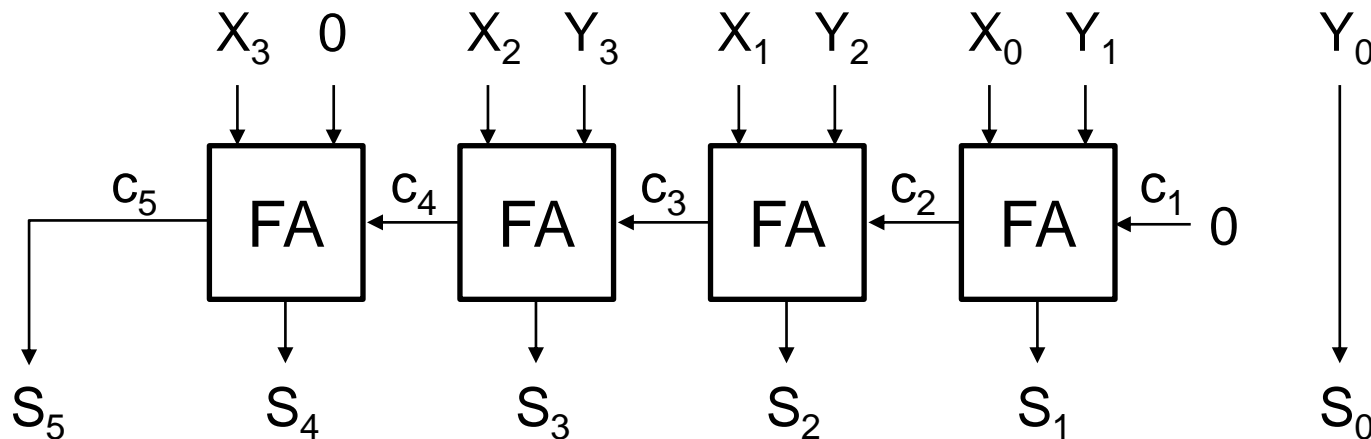


$S = 2 * X + Y$ (Unsigned X and Y)

- ❖ Design a circuit that computes $S = 2 * X + Y$ (**unsigned X and Y**)
- ❖ $X[3:0]$ and $Y[3:0]$ are 4-bit **unsigned** integers \rightarrow range = 0 to 15

Solution:

- ❖ $2 * X + Y = (X \ll 1) + Y$ (Shift-Left X by 1 bit)
- ❖ Maximum value of $S = 2 * 15 + 15 = 45 \rightarrow S$ is **6 bits** = $S[5:0]$

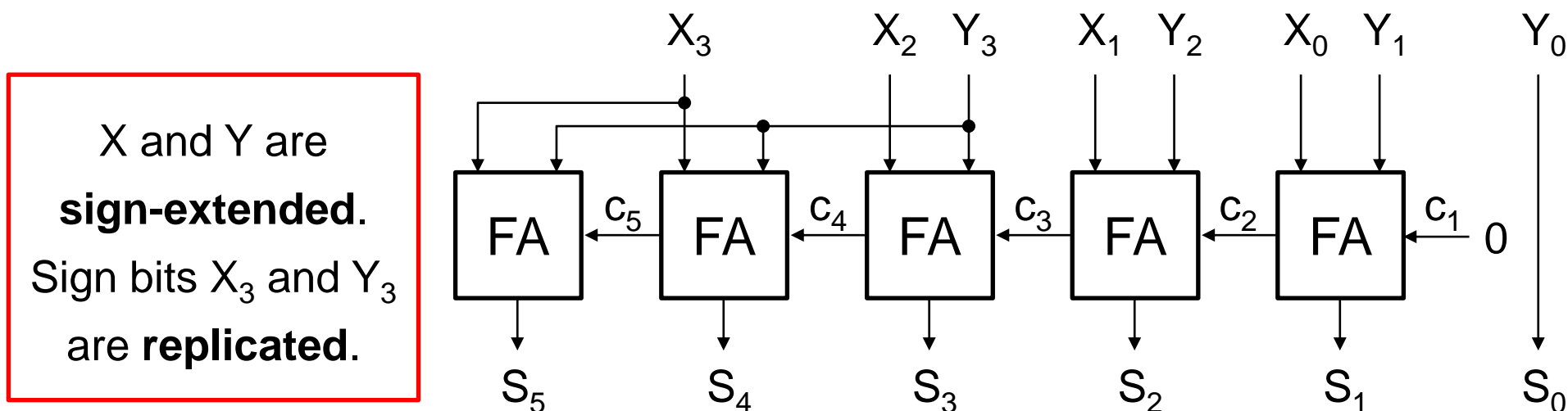


$S = 2 * X + Y$ (Signed X and Y)

- ❖ Design a circuit that computes $S = 2 * X + Y$ using Full Adders
- ❖ $X[3:0]$ and $Y[3:0]$ are 4-bit **signed** integers \rightarrow range = -8 to +7

Solution:

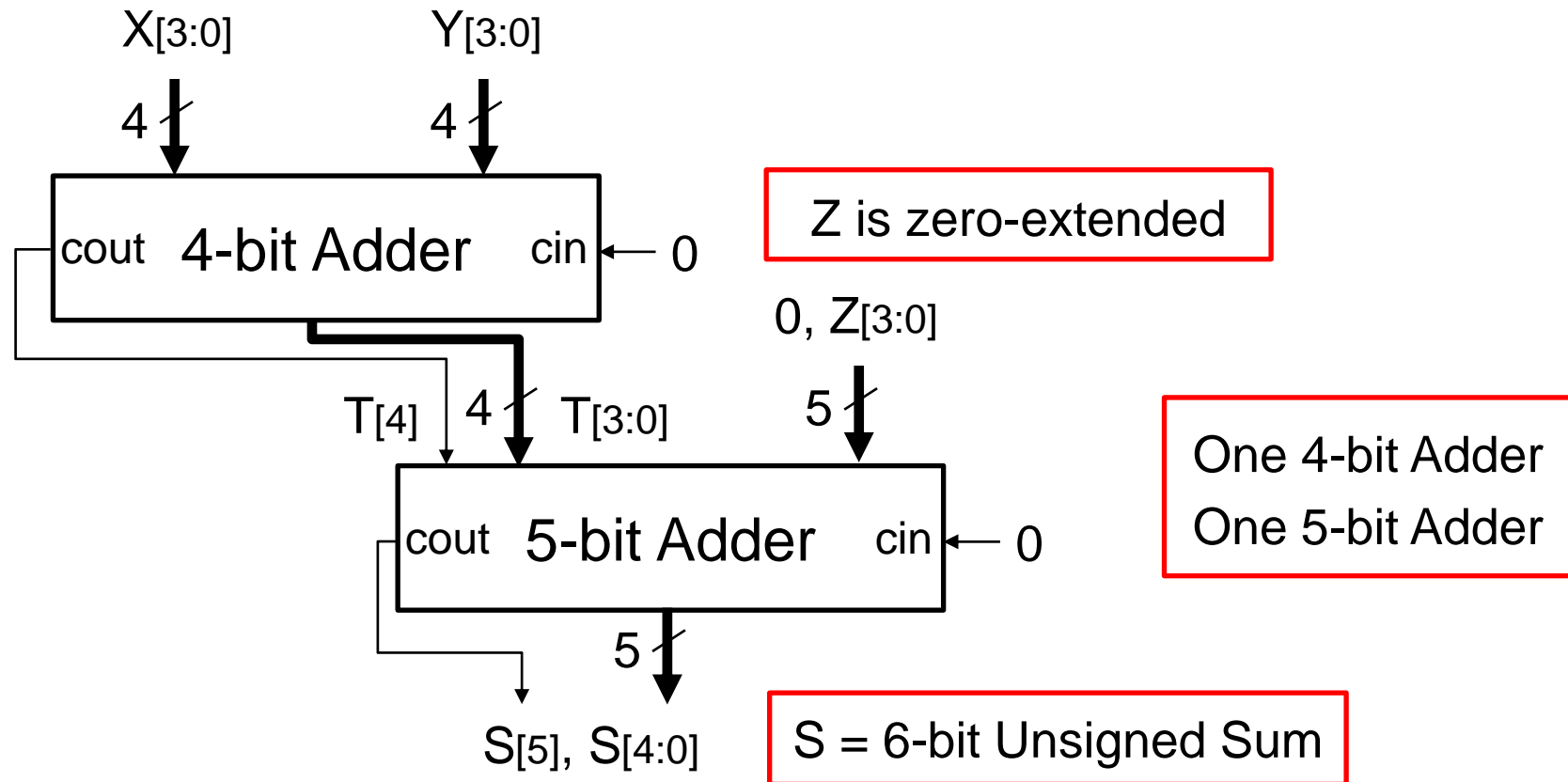
- ❖ Range of X and Y is -8 to +7 \rightarrow Minimum $S = 2 * (-8) + (-8) = -24$
- ❖ Maximum $S = 2 * (+7) + 7 = +21 \rightarrow S$ is **6 bits** = $S[5:0]$



Design a Circuit for Unsigned $S = X + Y + Z$

❖ X, Y, and Z are 4-bit **unsigned** integers → Range = 0 to 15

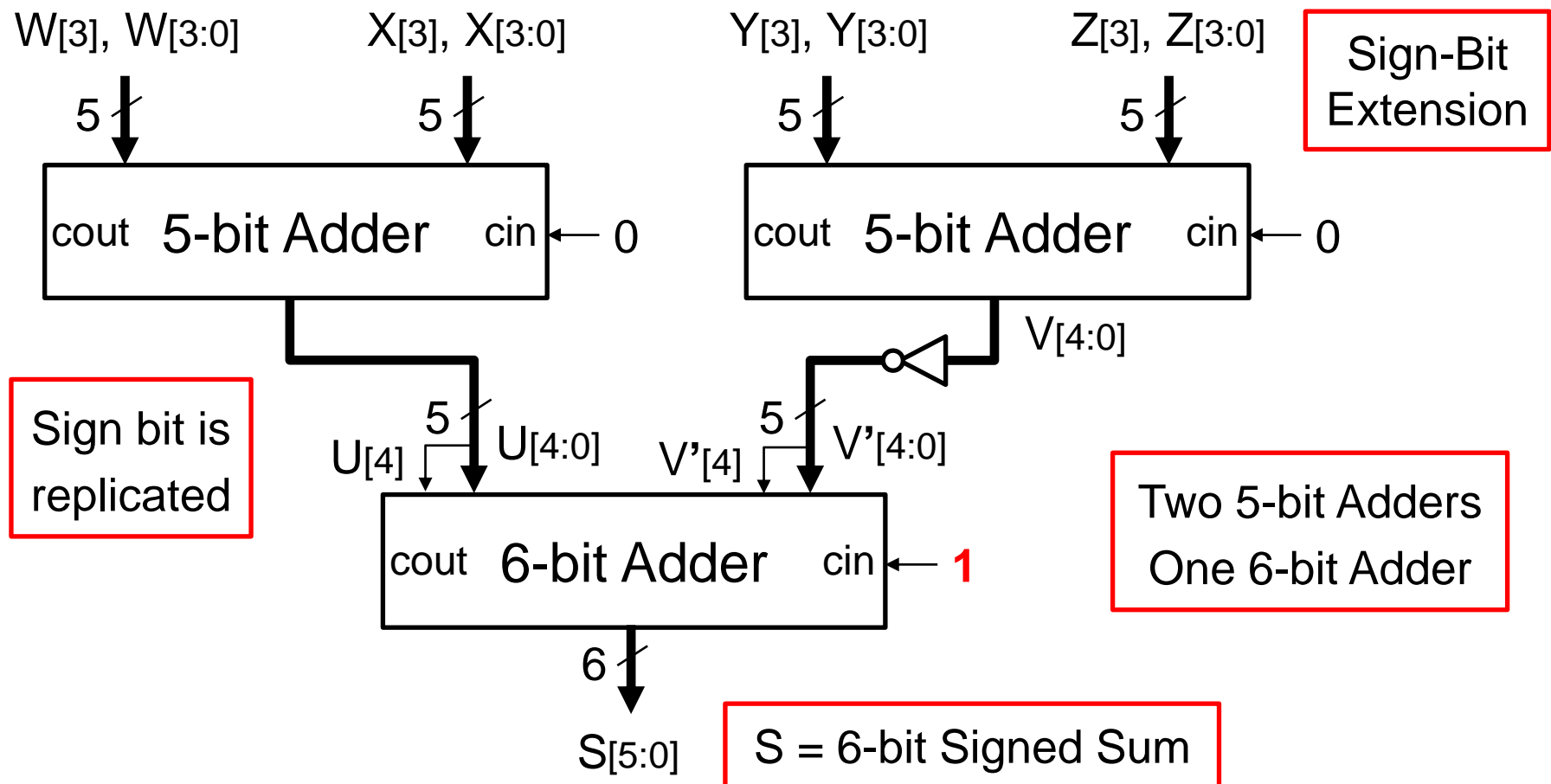
Solution: Maximum $S = 15 + 15 + 15 = 45$ → S must be **6 bits**



Design a Circuit for Signed $S = W + X - Y - Z$

❖ W, X, Y, and Z are 4-bit **signed** integers → Range = -8 to +7

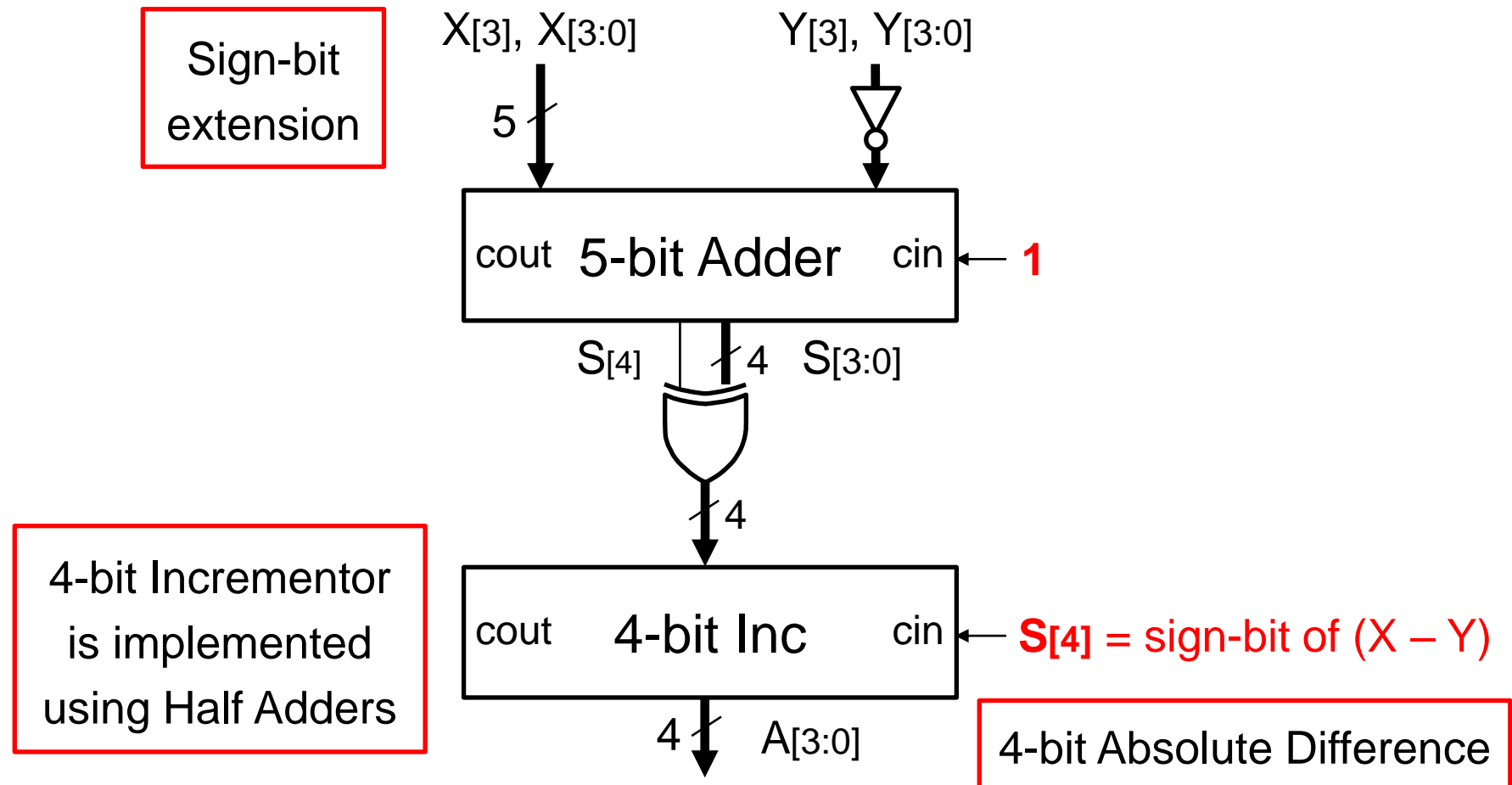
Solution: $S = W + X - Y - Z = (W+X) - (Y+Z) \rightarrow$ 6 bits are used



Absolute Difference $|X - Y|$ of Signed X, Y

❖ Design a circuit that computes $A = |X - Y|$ (absolute difference)

Solution: Maximum $A = |X - Y| = |-8 - +7| = 15 \rightarrow 4$ bits are used



Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ **Decimal Adder**
 - ✧ **BCD Adder**
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Design Examples

BCD Addition

- ❖ Consider adding two decimal digits in BCD
- ❖ Operands and Result: **0 to 9**
- ❖ Output sum cannot exceed **9 + 9 + 1 = 19**
 - ✧ The 1 in the sum is the input carry from previous digit
- ❖ We use a 4-bit binary adder to add the BCD digits
 - ✧ The adder will produce a result that ranges **from 0 through 19**
 - ✧ If the result is **more than 9**, it **must be corrected** to use 2 digits
 - ✧ To **correct** the digit, add 6 to the digit sum (a 4-bit binary adder)

$$\begin{array}{r} 1000 \\ + 0101 \\ \hline 1101 \\ + 0110 \\ \hline 1\ 0011 \end{array}$$

← Final answer
in BCD

$$\begin{array}{r} 8 \\ + 5 \\ \hline 13\ (>9) \\ + 6\ (\text{add } 6) \\ \hline 19\ (\text{carry} + 3) \end{array}$$

BCD Adder

Valid Codes

Invalid Codes
(need Correction)

Binary Sum					BCD Sum					Decimal
K	Z ₈	Z ₄	Z ₂	Z ₁	C	S ₈	S ₄	S ₂	S ₁	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

+0

+6

BCD Adder

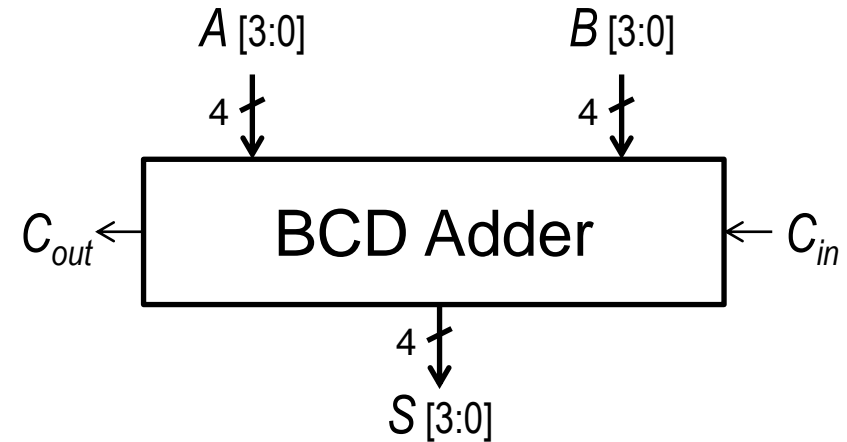
Correction is required if:

- 1) $Z > 9$ or
- 2) $K = 1$

$$C = K + Z_8 Z_4 + Z_8 Z_2$$

Correction circuit adds 0 or 6

$C = 0$	$C = 1$	➔	
$\begin{array}{cccc} Z_8 & Z_4 & Z_2 & Z_1 \\ + & 0 & 0 & 0 \\ \hline S_3 & S_2 & S_1 & S_0 \end{array}$	$\begin{array}{cccc} Z_8 & Z_4 & Z_2 & Z_1 \\ + & 0 & 1 & 1 \\ \hline S_3 & S_2 & S_1 & S_0 \end{array}$		$\begin{array}{cccc} Z_8 & Z_4 & Z_2 & Z_1 \\ + & 0 & C & C \\ \hline S_3 & S_2 & S_1 & S_0 \end{array}$

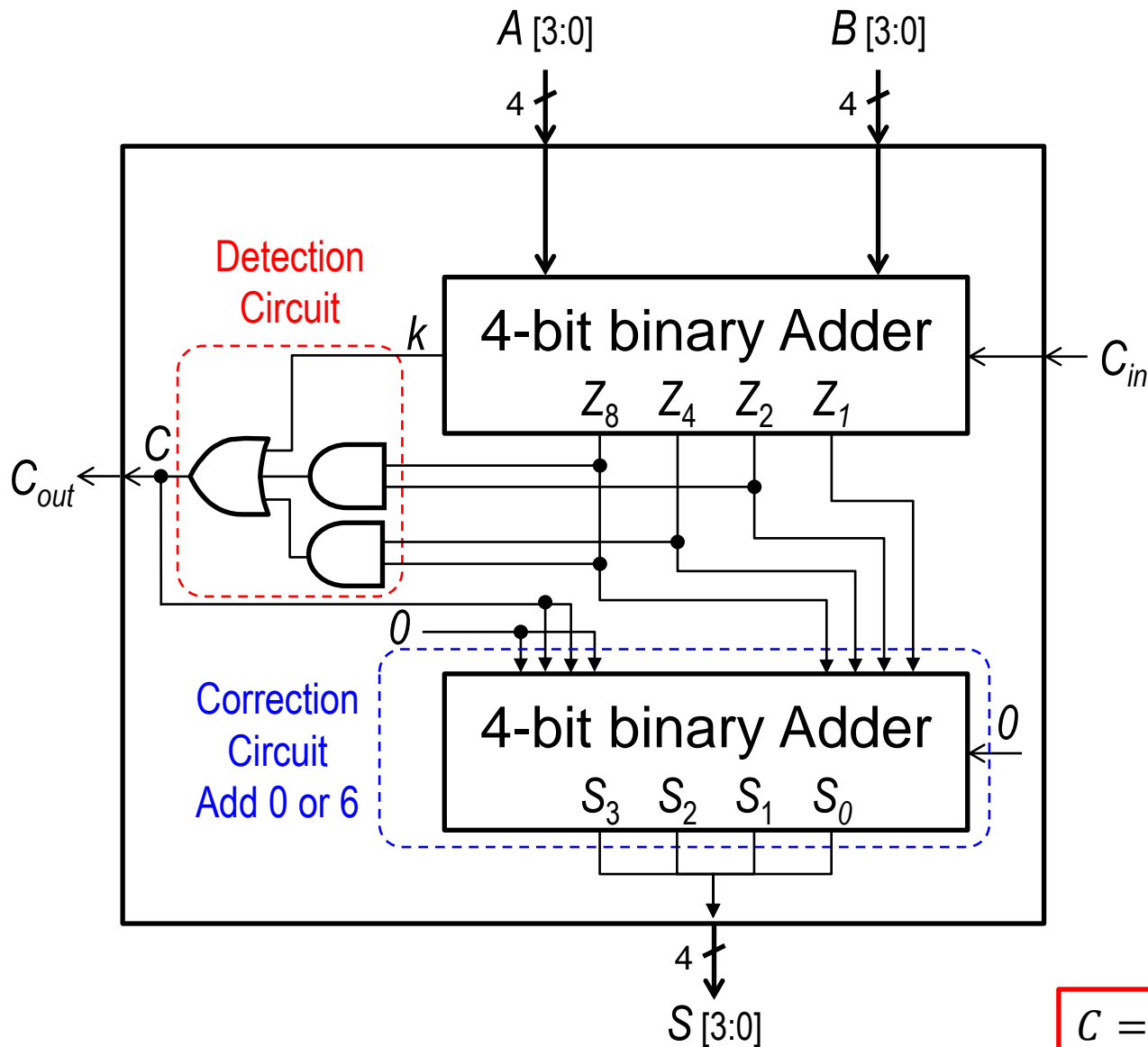


$Z > 9$

		$Z_2 Z_1$			
		00	01	11	10
$Z_8 Z_4$	00				
	01				
	11	1	1	1	1
	10			1	1

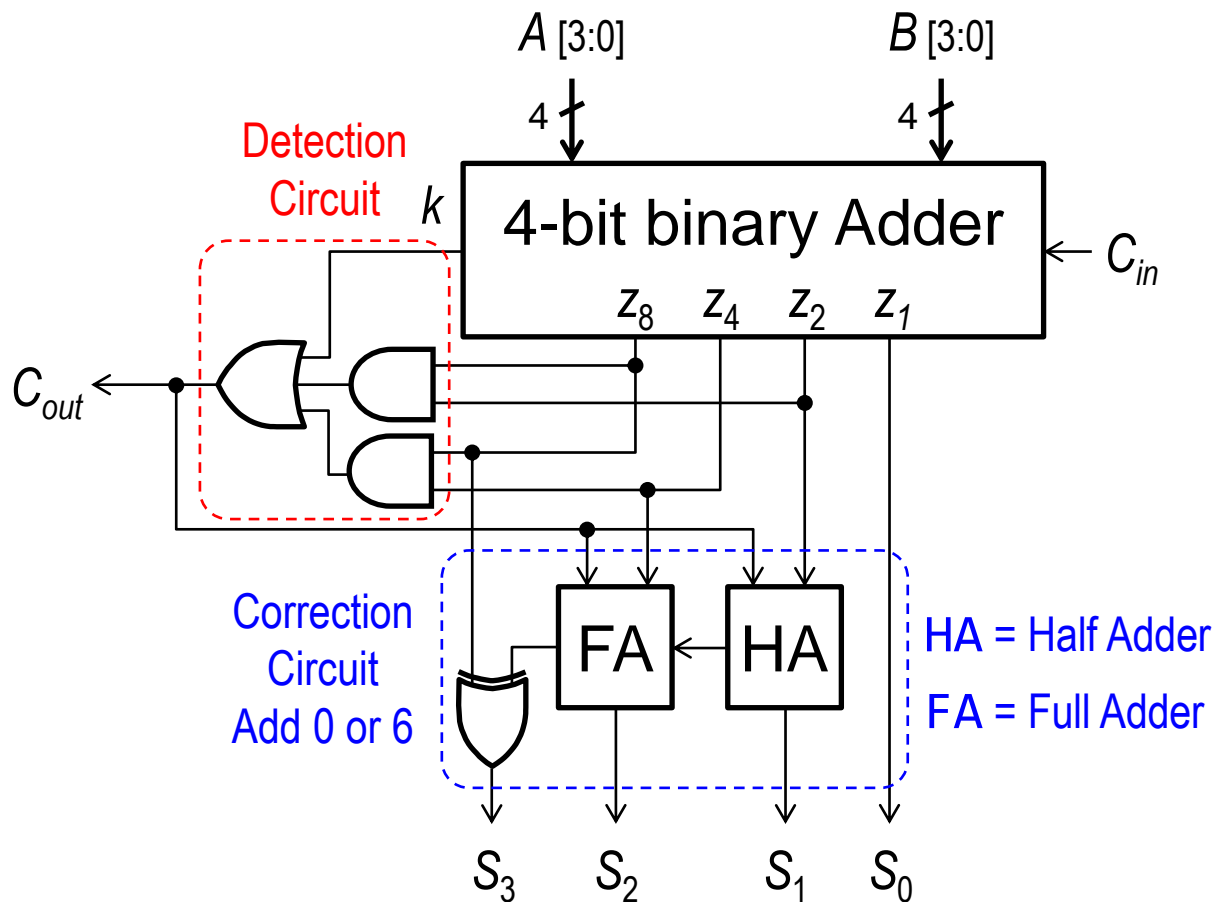
$$Z_8 Z_4 + Z_8 Z_2$$

BCD Adder



$$C = K + Z_8 Z_4 + Z_8 Z_2$$

BCD Adder



Multiple Digit BCD Addition

Add: 2905 + 1897 in BCD

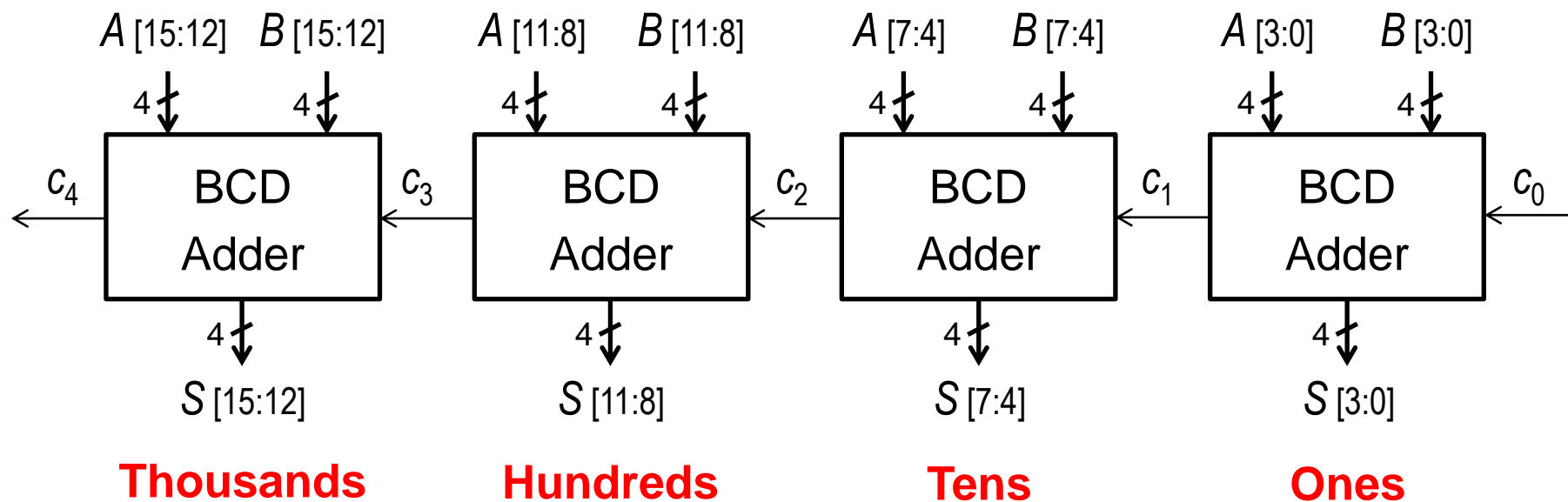
Showing carries and digit corrections

	carry	+1	+1	+1	
	+	0010	1001	0000	0101
		0001	1000	1001	0111
	<hr/>				
		0100	10010	1010	1100
digit correction			0110	0110	0110
	<hr/>				
		0100	1000	0000	0010

Final answer: $2905 + 1897 = 4802$

Ripple-Carry BCD Adder

- ❖ Inputs are BCD digits: 0 to 9
- ❖ Sum are BCD digits: **ones**, **tens**, **hundreds**, **thousands**, etc.
- ❖ Can be extended to any number of BCD digits
- ❖ BCD adders are larger in size than binary adders



Thousands

Hundreds

Tens

Ones

Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ **Binary Multiplier**
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Design Examples

Binary Multiplication

❖ Binary Multiplication is simple:

$$0 \times 0 = 0, \quad 0 \times 1 = 0, \quad 1 \times 0 = 0, \quad 1 \times 1 = 1$$

Multiplicand $1100_2 = 12$

Multiplier $\times 1101_2 = 13$

$$\begin{array}{r} 1100 \\ 0000 \\ 1100 \\ 1100 \end{array}$$

Product $10011100_2 = 156$

Binary multiplication

$0 \times \text{multiplicand} = 0$

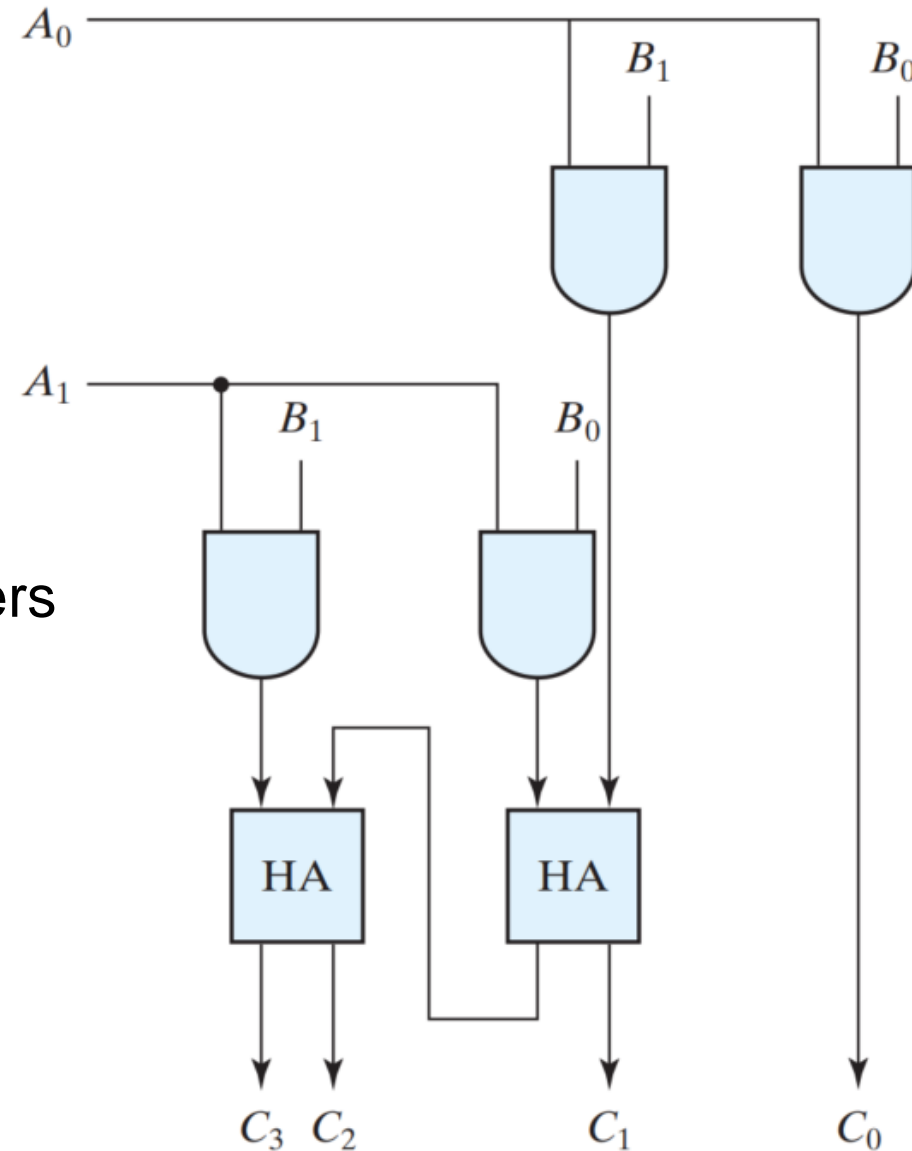
$1 \times \text{multiplicand} = \text{multiplicand}$

❖ n -bit multiplicand $\times m$ -bit multiplier = $(n + m)$ -bit product

2-bit × 2-bit Binary Multiplier

- ❖ Suppose we want to multiply two numbers $B = B_1B_0$ and $A = A_1A_0$
- ❖ Step 1: AND (multiply) each bit of A with each bit of B
 - ✧ Requires 2x2 AND gates and produces 2x2 product bits
- ❖ Step 2: Add the partial product
 - ✧ Requires (2 - 1) 2-bit binary adders

$$\begin{array}{r} B_1 B_0 \\ \underline{A_1 } A_0 \\ A_0 B_1 A_0 B_0 \\ \\ A_1 B_1 A_1 B_0 \\ \hline C_3 C_2 C_1 \end{array}$$

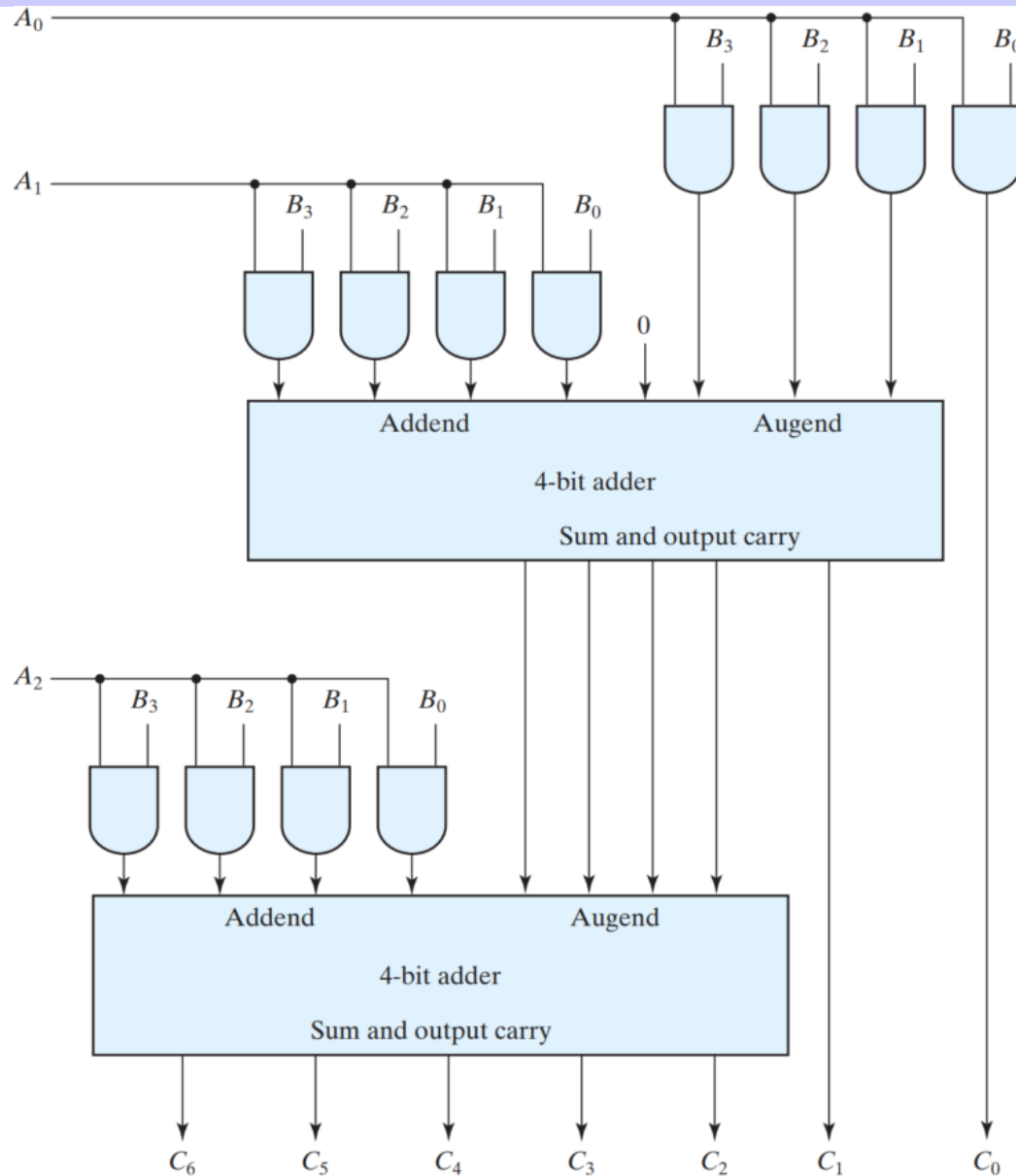


4-bit × 3-bit Binary Multiplier

- ❖ Suppose we want to multiply two numbers $B = B_3B_2B_1B_0$ and $A = A_2A_1A_0$
- ❖ Step 1: AND (multiply) each bit of A with each bit of B
 - ✧ Requires 4x3 AND gates and produces 4x3 product bits
- ❖ Step 2: Add the partial product
 - ✧ Requires (3 - 1) 4-bit binary adders

$$\begin{array}{rccccccc} & & & B_3 & B_2 & B_1 & B_0 & \\ & & & X & A_2 & A_1 & A_0 & \\ \hline & & A_0B_3 & A_0B_2 & A_0B_1 & A_0B_0 & & \\ & A_1B_3 & A_1B_2 & A_1B_1 & A_1B_0 & & & \\ & A_2B_3 & A_2B_2 & A_2B_1 & A_2B_0 & & & \\ \hline C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0 & \end{array}$$

4-bit × 3-bit Binary Multiplier



$$\begin{array}{r}
 \begin{array}{cccc}
 B_3 & B_2 & B_1 & B_0 \\
 \times & A_2 & A_1 & A_0 \\
 \hline
 A_0 B_3 & A_0 B_2 & A_0 B_1 & A_0 B_0 \\
 A_1 B_3 & A_1 B_2 & A_1 B_1 & A_1 B_0 \\
 A_2 B_3 & A_2 B_2 & A_2 B_1 & A_2 B_0 \\
 \hline
 C_6 & C_5 & C_4 & C_3 & C_2 & C_1 & C_0
 \end{array}
 \end{array}$$

Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ **Magnitude Comparator**
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Design Examples

Magnitude Comparator

❖ A combinational circuit that compares two unsigned integers

❖ Two Inputs:

✧ Unsigned integer A (m -bit number)

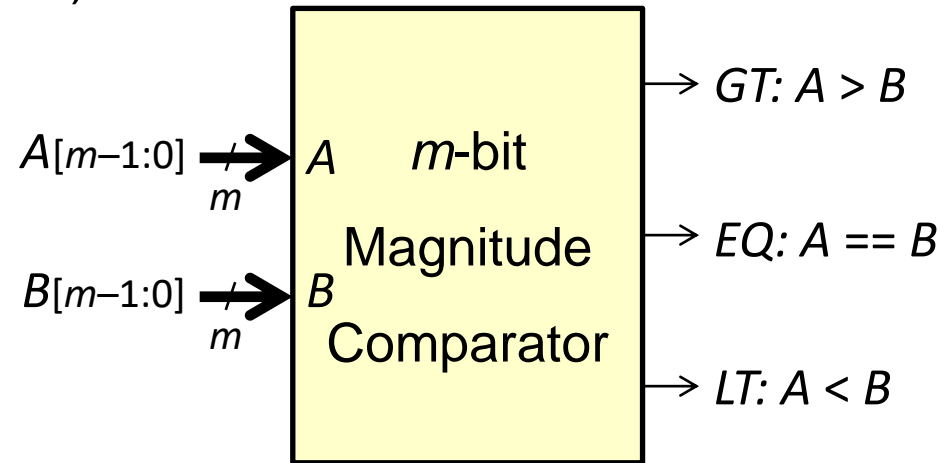
✧ Unsigned integer B (m -bit number)

❖ Three outputs:

✧ $A > B$ (GT output)

✧ $A == B$ (EQ output)

✧ $A < B$ (LT output)



❖ Exactly one of the three outputs must be equal to 1

❖ While the remaining two outputs must be equal to 0

Example: 4-bit Magnitude Comparator

❖ Inputs:

✧ $A = A_3A_2A_1A_0$

✧ $B = B_3B_2B_1B_0$

✧ 8 bits in total → 256 possible combinations

✧ Not simple to design using conventional K-map techniques

❖ The magnitude comparator can be designed at a higher level

❖ Let us implement first the EQ output (A is equal to B)

✧ $EQ = 1 \leftrightarrow A_3 == B_3, A_2 == B_2, A_1 == B_1, \text{ and } A_0 == B_0$

✧ Define: $E_i = (A_i == B_i) = (A_i \oplus B_i)' = A_iB_i + A_i'B_i'$

✧ Therefore, $EQ = (A == B) = E_3E_2E_1E_0$

The Greater Than Output

Given the 4-bit input numbers: A and B

1. If $A_3 > B_3$ then $GT = 1$, irrespective of the lower bits of A and B

Define: $G_3 = A_3 B'_3$ ($A_3 == 1$ and $B_3 == 0$)

2. If $A_3 == B_3$ ($E_3 == 1$), we compare A_2 with B_2

Define: $G_2 = A_2 B'_2$ ($A_2 == 1$ and $B_2 == 0$)

3. If $A_3 == B_3$ and $A_2 == B_2$, we compare A_1 with B_1

Define: $G_1 = A_1 B'_1$ ($A_1 == 1$ and $B_1 == 0$)

4. If $A_3 == B_3$ and $A_2 == B_2$ and $A_1 == B_1$, we compare A_0 with B_0

Define: $G_0 = A_0 B'_0$ ($A_0 == 1$ and $B_0 == 0$)

Therefore, $GT = G_3 + E_3 G_2 + E_3 E_2 G_1 + E_3 E_2 E_1 G_0$

The Less Than Output

We can derive the expression for the LT output, similar to GT

Given the 4-bit input numbers: A and B

1. If $A_3 < B_3$ then $LT = 1$, irrespective of the lower bits of A and B

Define: $L_3 = A'_3 B_3$ ($A_3 == 0$ and $B_3 == 1$)

2. If $A_3 = B_3$ ($E_3 == 1$), we compare A_2 with B_2

Define: $L_2 = A'_2 B_2$ ($A_2 == 0$ and $B_2 == 1$)

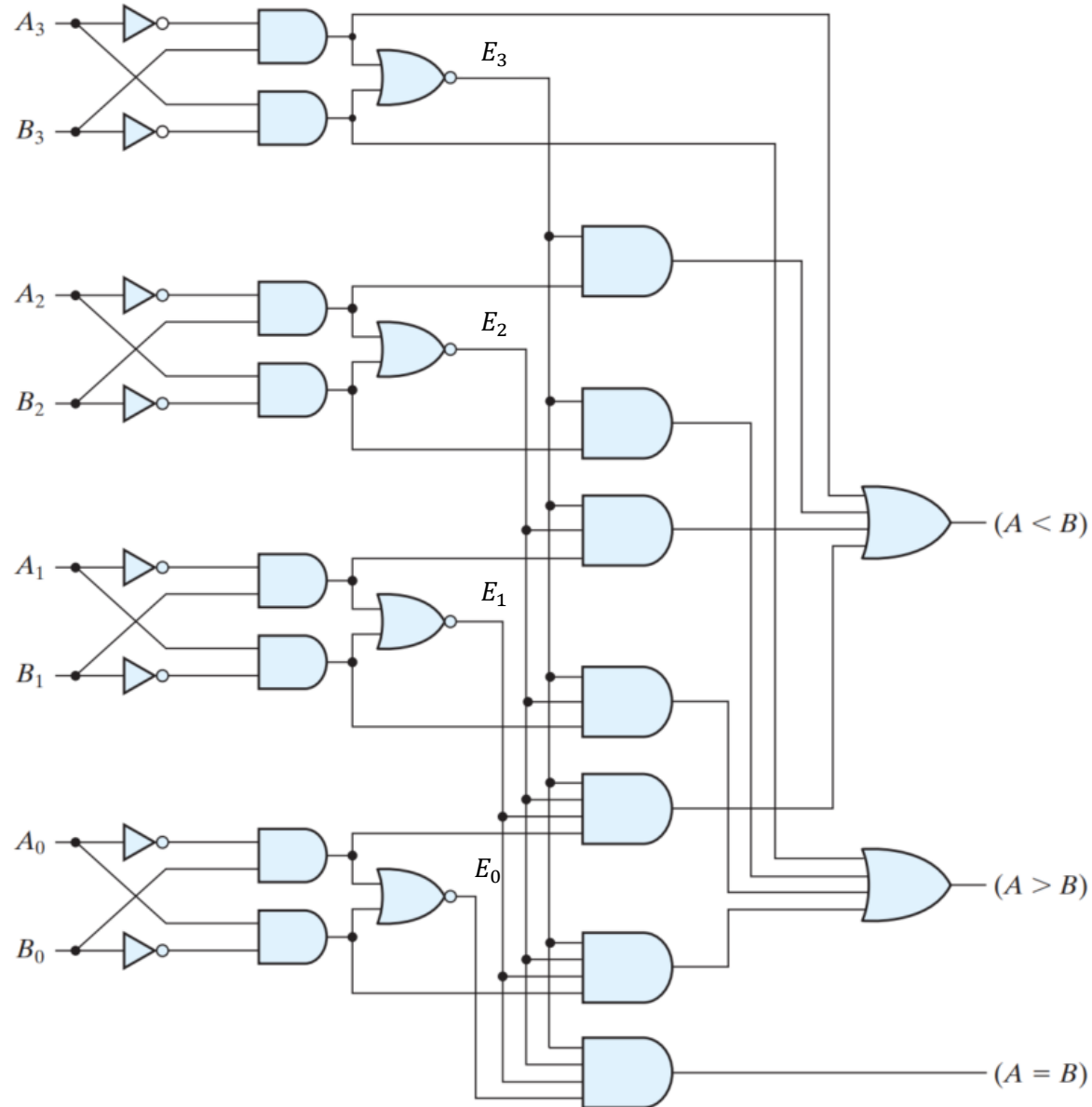
3. Define: $L_1 = A'_1 B_1$ ($A_1 == 0$ and $B_1 == 1$)

4. Define: $L_0 = A'_0 B_0$ ($A_0 == 0$ and $B_0 == 1$)

Therefore, $LT = L_3 + E_3 L_2 + E_3 E_2 L_1 + E_3 E_2 E_1 L_0$

Knowing GT and EQ , we can also derive $LT = (GT + EQ)'$

Example: 4-bit Magnitude Comparator



Iterative Magnitude Comparator Design

❖ The Magnitude comparator can also be designed iteratively

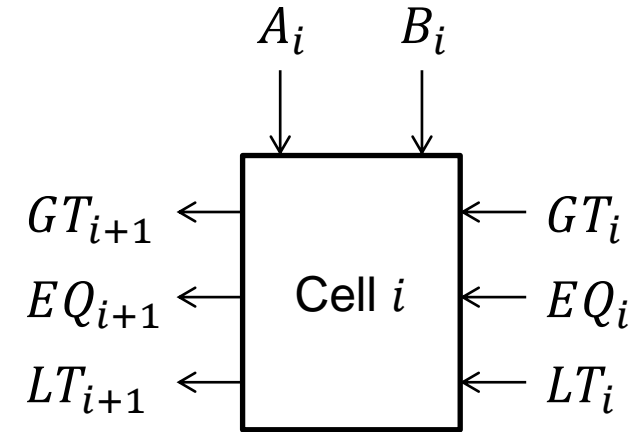
❖ Each Cell i receives as inputs:

Bit i of inputs A and B : A_i and B_i
 GT_i , EQ_i , and LT_i from cell $(i - 1)$

❖ Each Cell i produces three outputs:

GT_{i+1} , EQ_{i+1} , and LT_{i+1}

Outputs of cell i are inputs to cell $(i + 1)$



❖ **Output Expressions of Cell i**

$$EQ_{i+1} = E_i \cdot EQ_i$$

$$E_i = A_i' B_i' + A_i B_i \quad (A_i \text{ equals } B_i)$$

$$GT_{i+1} = A_i B_i' + E_i \cdot GT_i$$

$$A_i B_i' \quad (A_i > B_i)$$

$$LT_{i+1} = A_i' B_i + E_i \cdot LT_i$$

$$A_i' B_i \quad (A_i < B_i)$$

Third output can be produced for first two: $LT_{i+1} = (EQ_{i+1} + GT_{i+1})'$

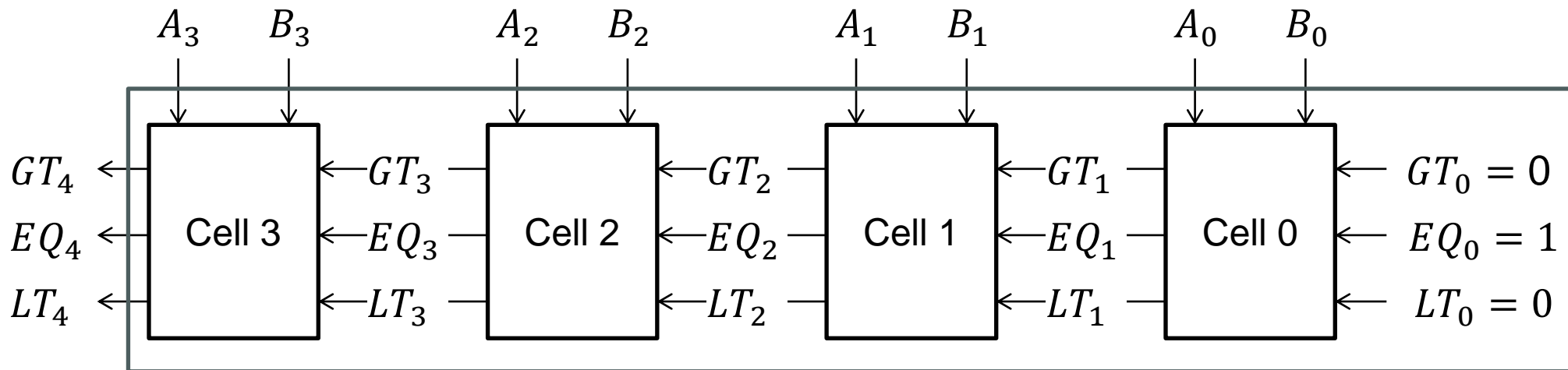
Iterative Magnitude Comparator Design

❖ 4-bit magnitude comparator is implemented using 4 identical cells

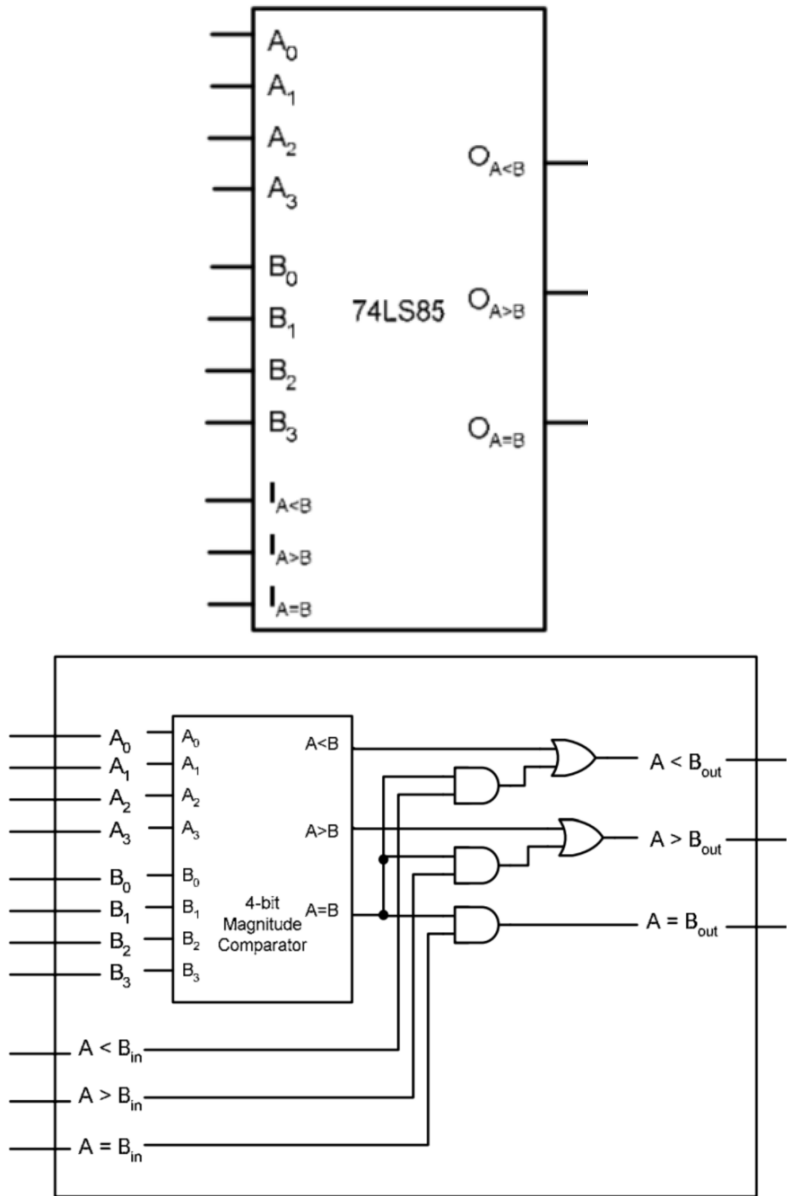
Design can be extended to any number of cells

❖ Comparison starts at least-significant bit

❖ Final comparator output: $GT = GT_4$, $EQ = EQ_4$, $LT = LT_4$



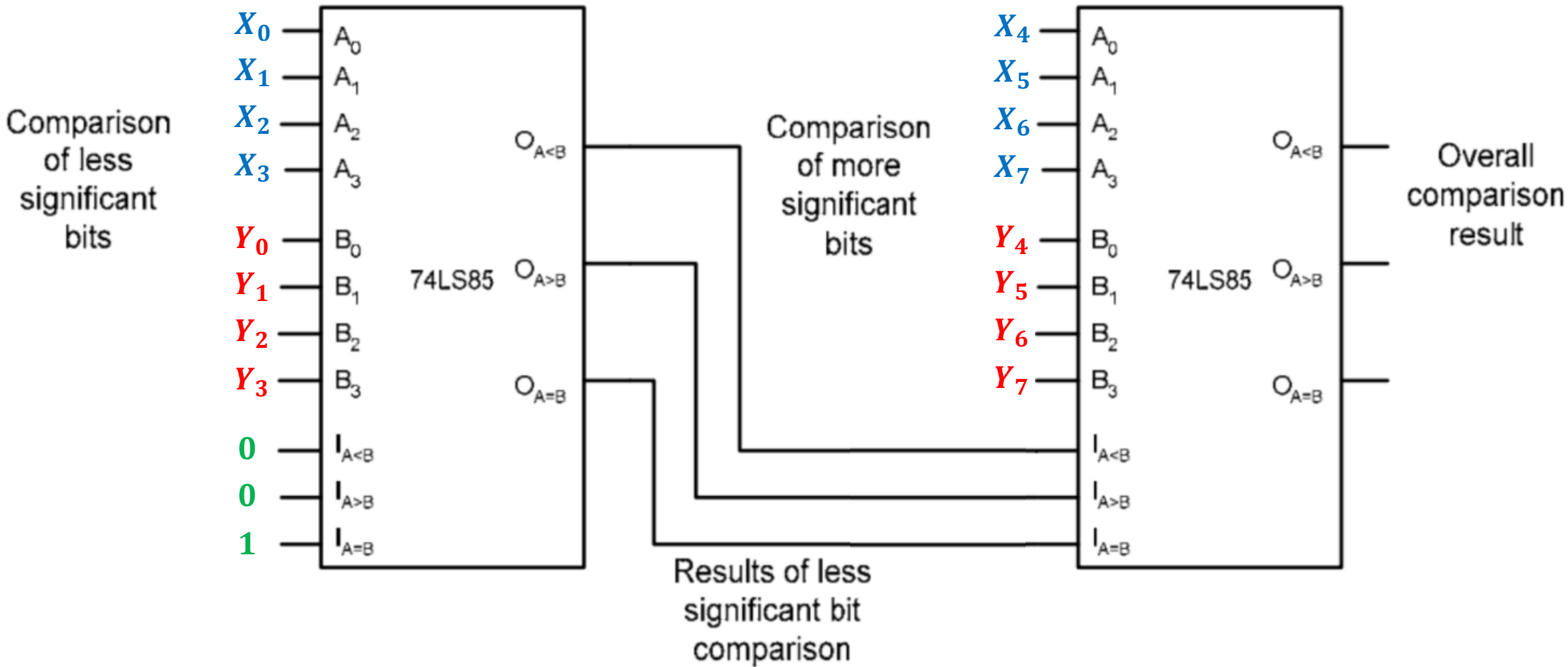
DM74LS85: A 4-Bit Magnitude Comparator



Comparing Inputs				Cascading Inputs			Outputs		
A3, B3	A2, B2	A1, B1	A0, B0	A > B	A < B	A = B	A > B	A < B	A = B
A3 > B3	X	X	X	X	X	X	H	L	L
A3 < B3	X	X	X	X	X	X	L	H	L
A3 = B3	A2 > B2	X	X	X	X	X	H	L	L
A3 = B3	A2 < B2	X	X	X	X	X	L	H	L
A3 = B3	A2 = B2	A1 > B1	X	X	X	X	H	L	L
A3 = B3	A2 = B2	A1 < B1	X	X	X	X	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 > B0	X	X	X	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 < B0	X	X	X	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	L	L	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	H	L	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	L	H	L	L	H
A3 = B3	A2 = B2	A1 = B1	A0 = B0	X	X	H	L	L	H
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	H	L	L	L	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	L	L	H	H	L

H = HIGH Level, L = LOW Level, X = Don't Care

Cascading Two Comparators



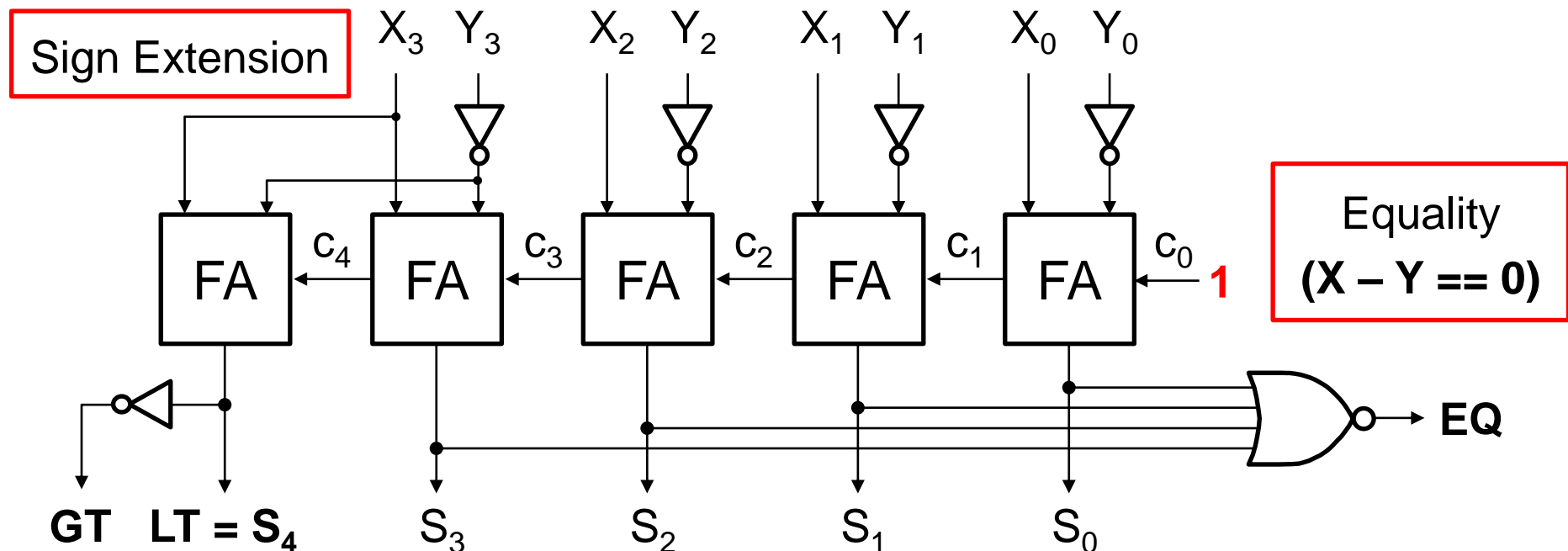
Signed Less Than: $LT = X < Y$

❖ Design a circuit that computes **signed LT** (Signed X and Y)

Solution:

❖ If $(X < Y)$ then $(X - Y) < 0$, If $(X == Y)$ then $(X - Y == 0)$

❖ Do **signed subtraction**, $LT = S_4 = \text{sign-bit}$ of the result

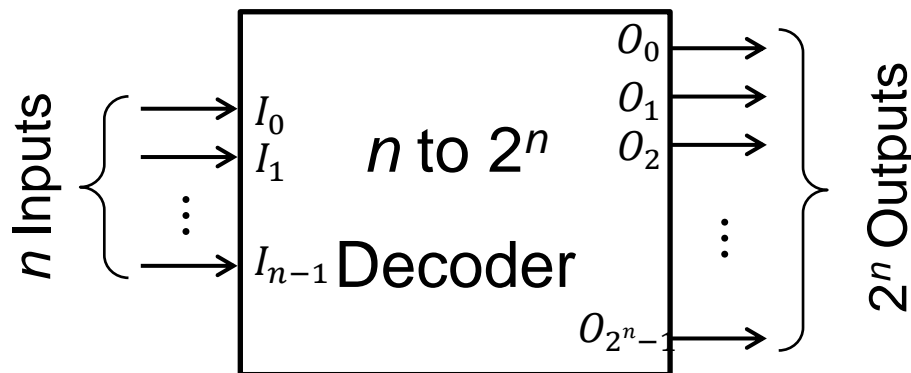


Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ **Decoders**
- ❖ Encoders
- ❖ Multiplexers
- ❖ Design Examples

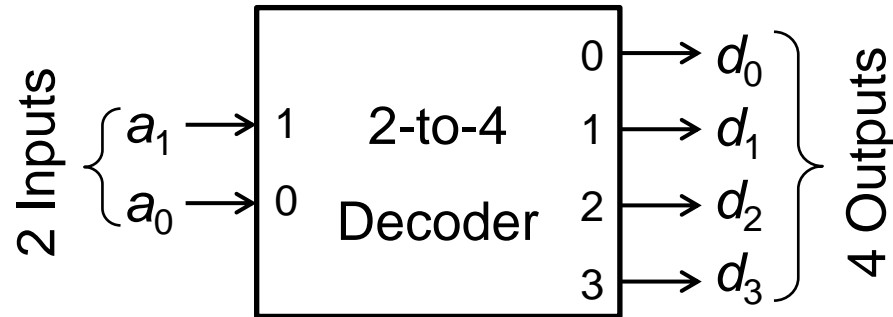
Binary Decoders

- ❖ Given a n -bit binary code, there are 2^n possible code values
- ❖ The decoder has an output for each possible code value
- ❖ The n -to- 2^n decoder has n inputs and 2^n outputs
- ❖ Depending on the input code, **only one output** is set to **logic 1**
- ❖ The conversion of input to output is called **decoding**



A decoder can have less than 2^n outputs if some input codes are unused

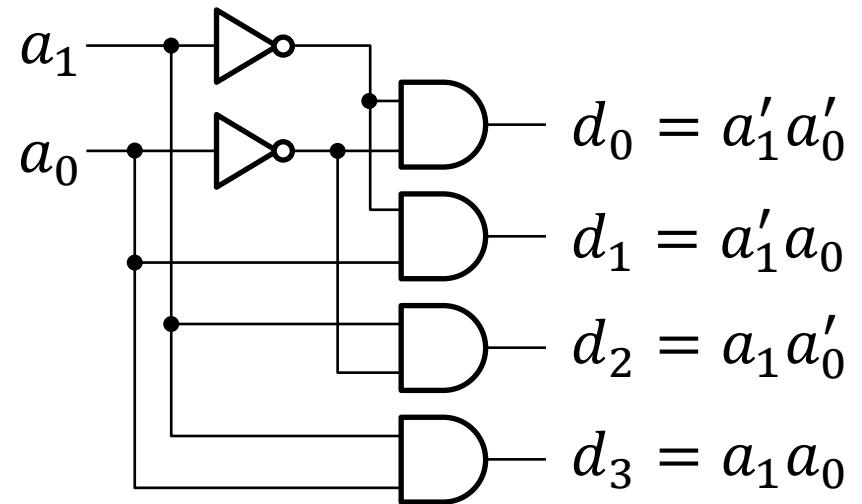
Examples of Binary Decoders



Truth Tables

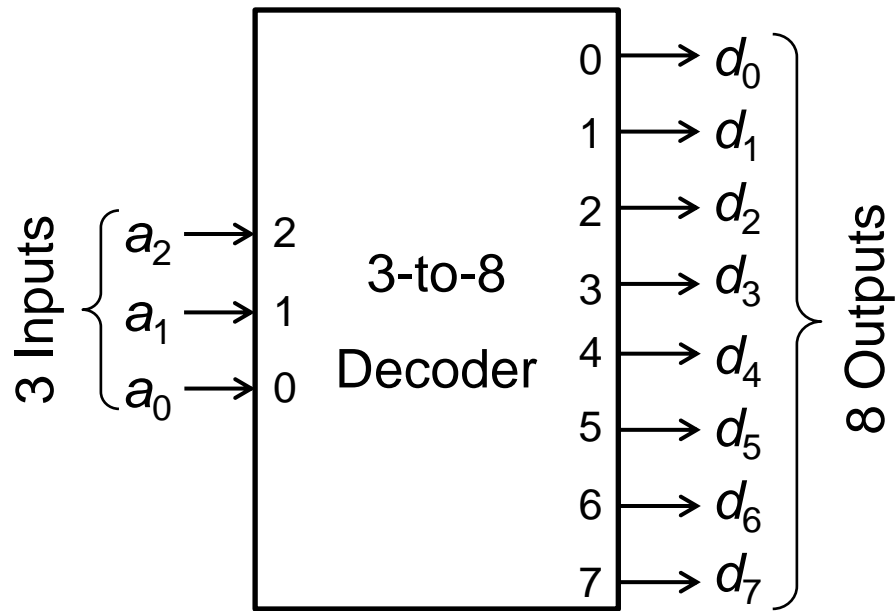
Inputs		Outputs			
a_1	a_0	d_0	d_1	d_2	d_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

2-to-4 Decoder Implementation



Each decoder output is a **minterm**

Examples of Binary Decoders

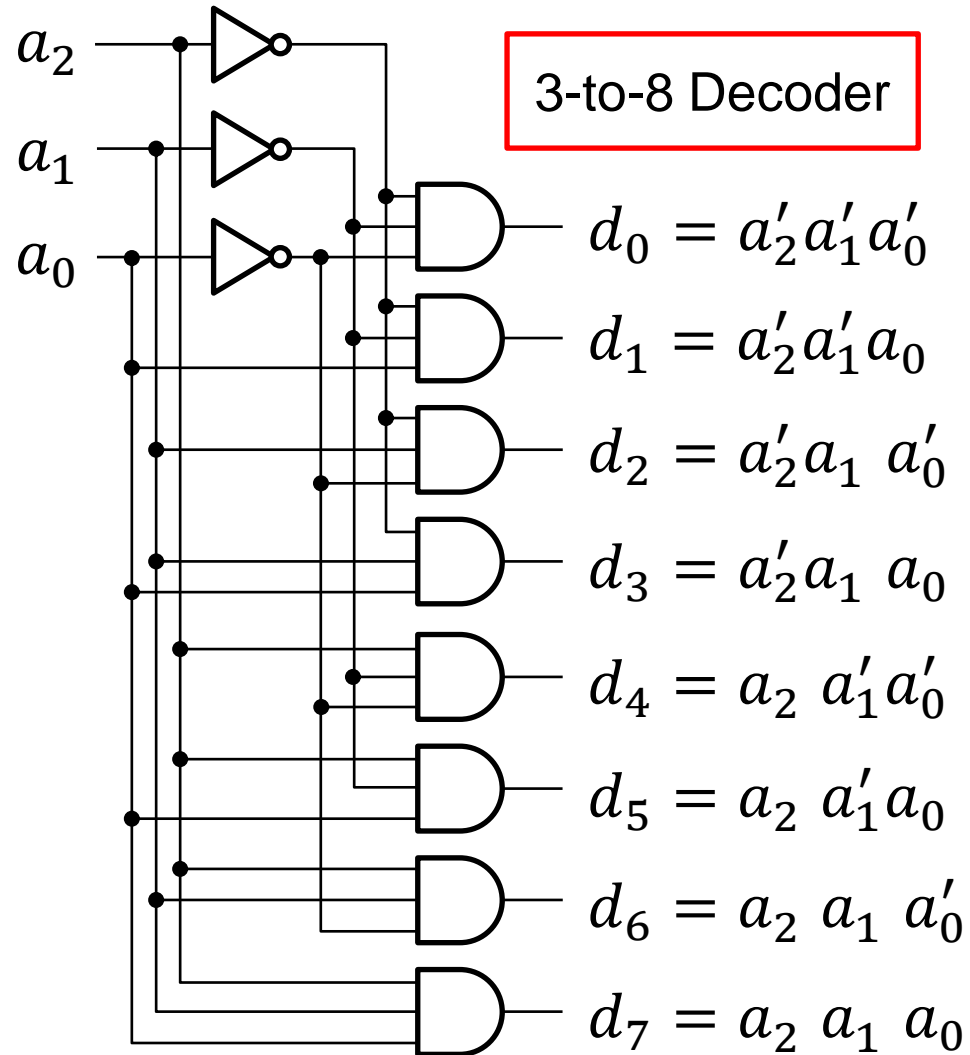


Truth Tables

Inputs			Outputs							
a ₂	a ₁	a ₀	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

3-to-8 Decoder Implementation

Each decoder output
is a **minterm**



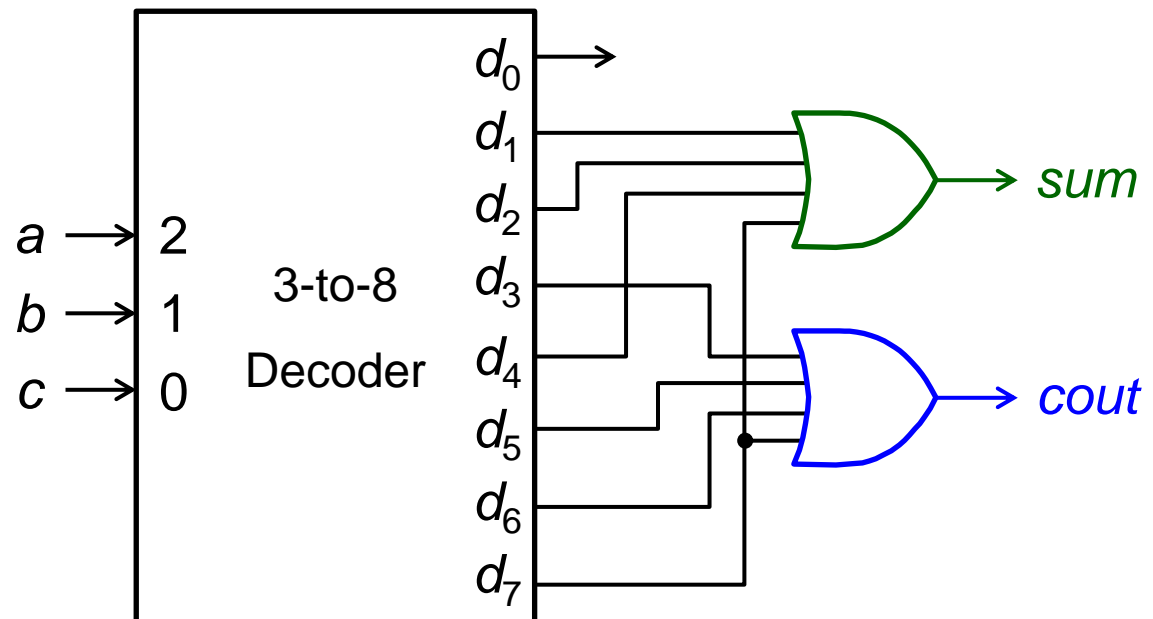
Using Decoders to Implement Functions

- ❖ A decoder generates all the minterms
- ❖ A Boolean function can be expressed as a sum of minterms
- ❖ Any function can be implemented using a decoder + OR gate

Note: the function **must not be minimized**

- ❖ **Example:** Full Adder $\text{sum} = \Sigma(1, 2, 4, 7)$, $\text{cout} = \Sigma(3, 5, 6, 7)$

Inputs			Outputs	
a	b	c	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

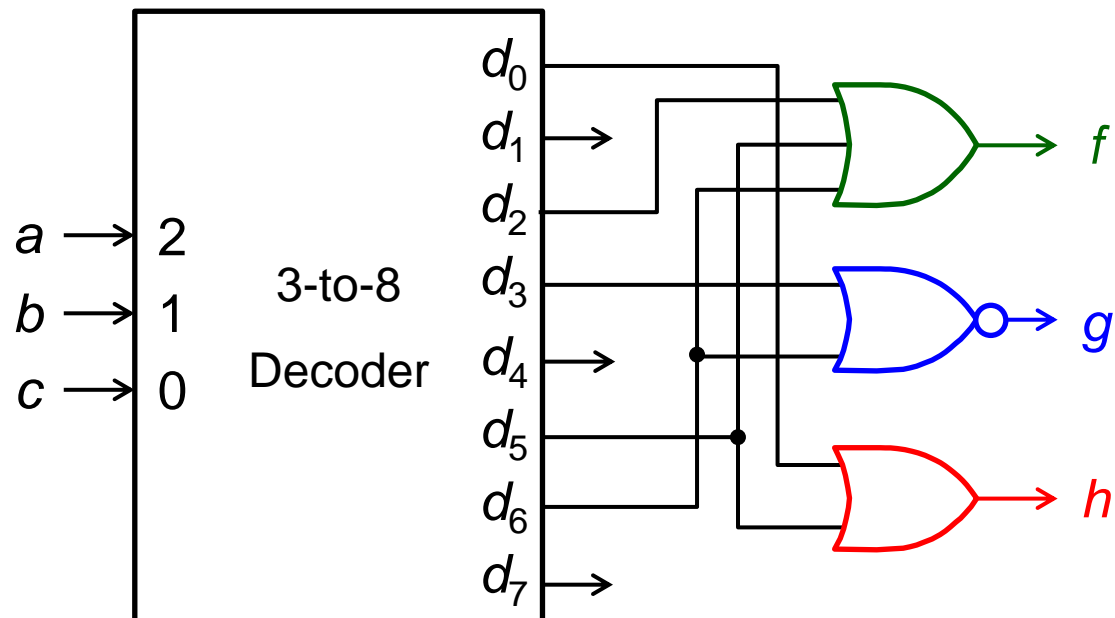


Using Decoders to Implement Functions

- ❖ Good if many output functions of the same input variables
- ❖ If number of minterms is large → Wider OR gate is needed
- ❖ Use NOR gate if number of maxterms is less than minterms
- ❖ **Example:** $f(a,b,c) = \Sigma(2, 5, 6)$, $g(a,b,c) = \Pi(3, 6)$, $h(a,b,c) = \Sigma(0, 5)$

→ $g' = \Sigma(3, 6)$

Inputs			Outputs		
a	b	c	f	g	h
0	0	0	0	1	1
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	0	0	0
1	0	0	0	1	0
1	0	1	1	1	1
1	1	0	1	0	0
1	1	1	0	1	0

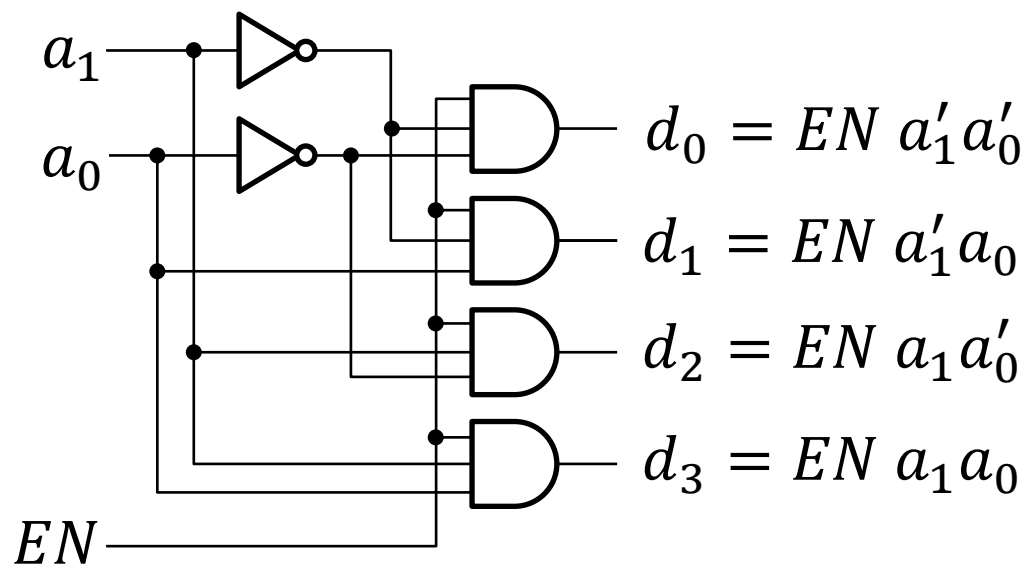
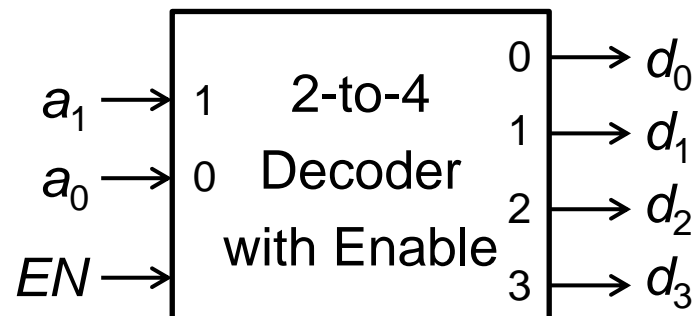


2-to-4 Decoder with Enable Input

Truth Table

Inputs		Outputs			
EN	$a_1 a_0$	d_0	d_1	d_2	d_3
0	X X	0	0	0	0
1	0 0	1	0	0	0
1	0 1	0	1	0	0
1	1 0	0	0	1	0
1	1 1	0	0	0	1

If EN input is zero then all outputs are zeros, regardless of a_1 and a_0

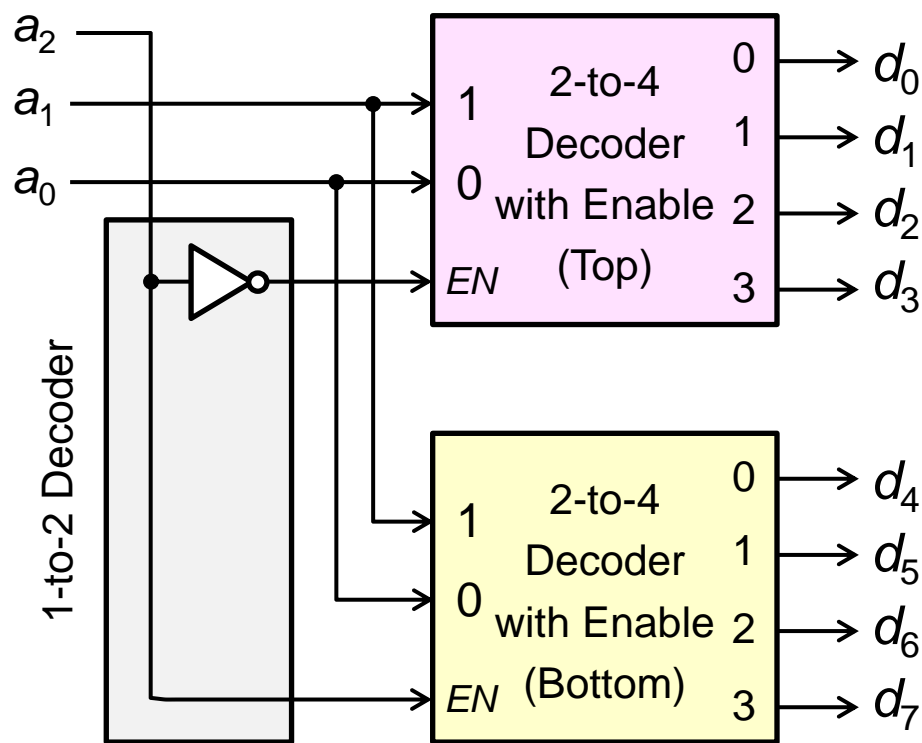


Building Larger Decoders

- ❖ Larger decoders can be build using smaller ones
- ❖ A 3-to-8 decoder can be built using:

Two 2-to-4 decoders with Enable and an inverter (1-to-2 decoder)

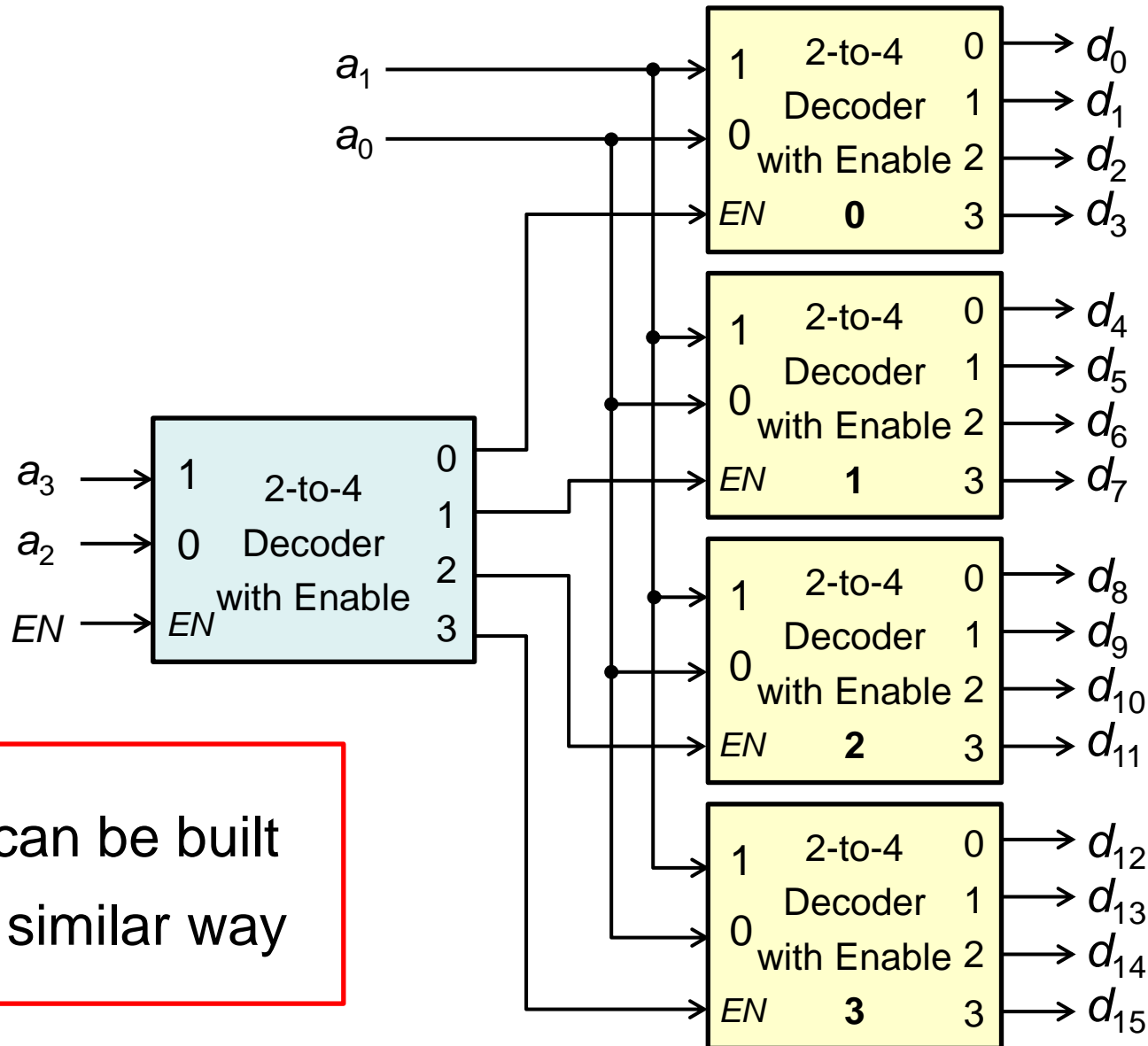
Inputs			Outputs							
a_2	a_1	a_0	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



Building Larger Decoders

A 4-to-16 decoder with enable can be built using **five** 2-to-4 decoders with enables

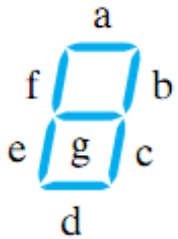
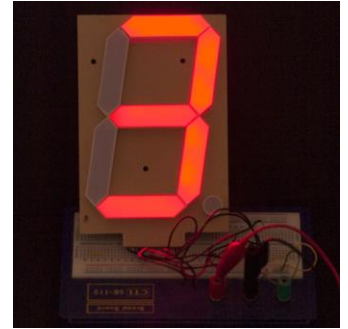
Larger decoders can be built hierarchically in a similar way



BCD to 7-Segment Decoder

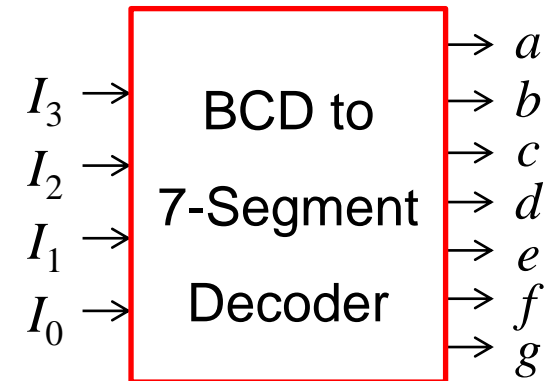
❖ Seven-Segment Display:

- ✧ Made of Seven segments: light-emitting diodes (LED)
- ✧ Found in electronic devices: such as clocks, calculators, etc.



❖ BCD to 7-Segment Decoder

- ✧ Called also a decoder, but not a binary decoder
- ✧ Accepts as input a BCD decimal digit (0 to 9)
- ✧ Generates output to the seven LED segments to display the BCD digit
- ✧ Each segment can be turned on or off separately



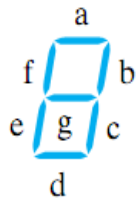
BCD to 7-Segment Decoder

Specification:

- ✧ Input: 4-bit BCD (I_3, I_2, I_1, I_0)
- ✧ Output: 7-bit (a, b, c, d, e, f, g)
- ✧ Display should be OFF for Non-BCD input codes.

Implementation can use:

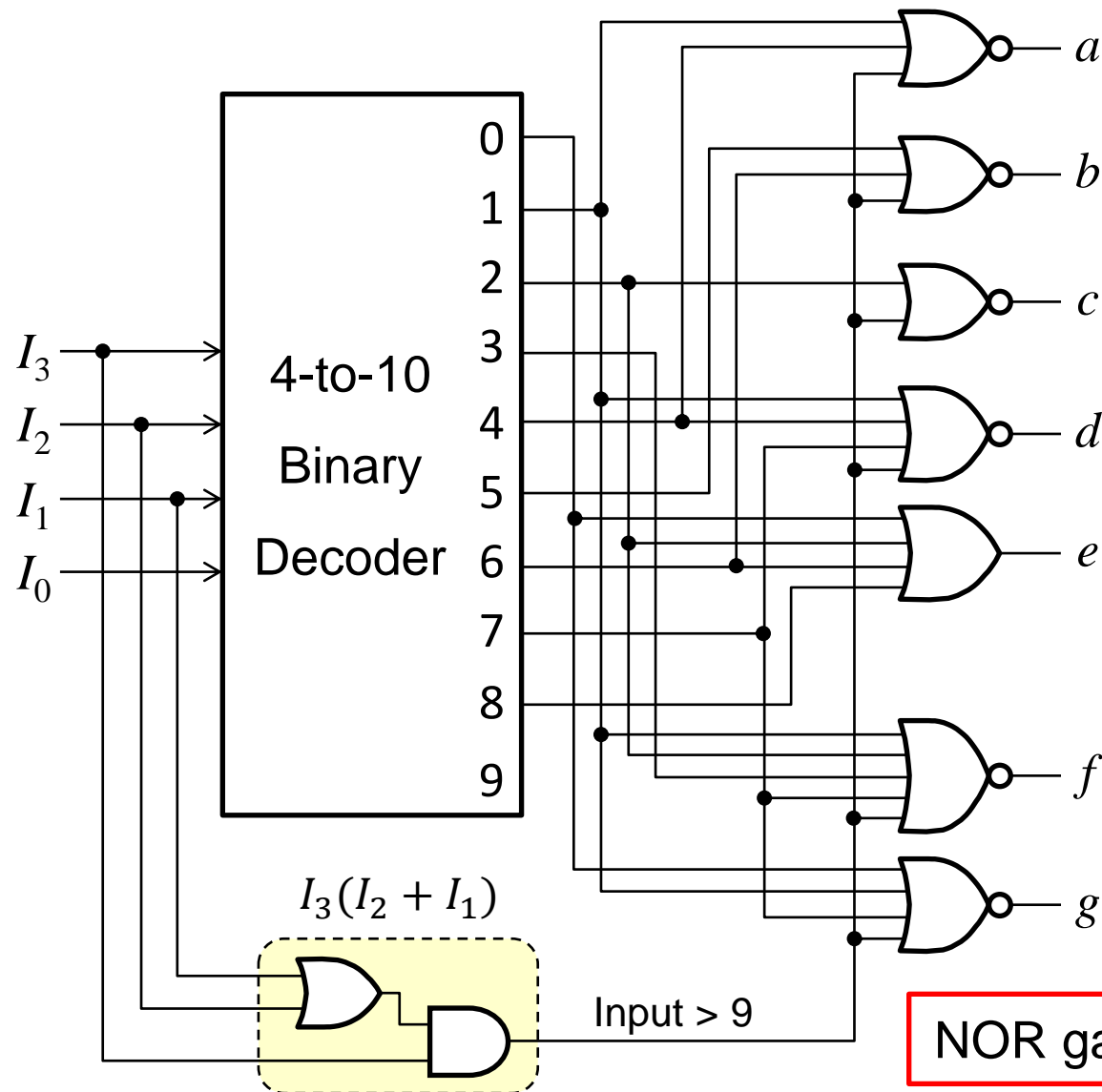
- ✧ A binary decoder
- ✧ Additional gates



Truth Table

BCD input				7-Segment Output						
I_3	I_2	I_1	I_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1010 to 1111				0	0	0	0	0	0	0

Implementing a BCD to 7-Segment Decoder



Truth Table

I_3	I_2	I_1	I_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1010	-	1111		0	0	0	0	0	0	0

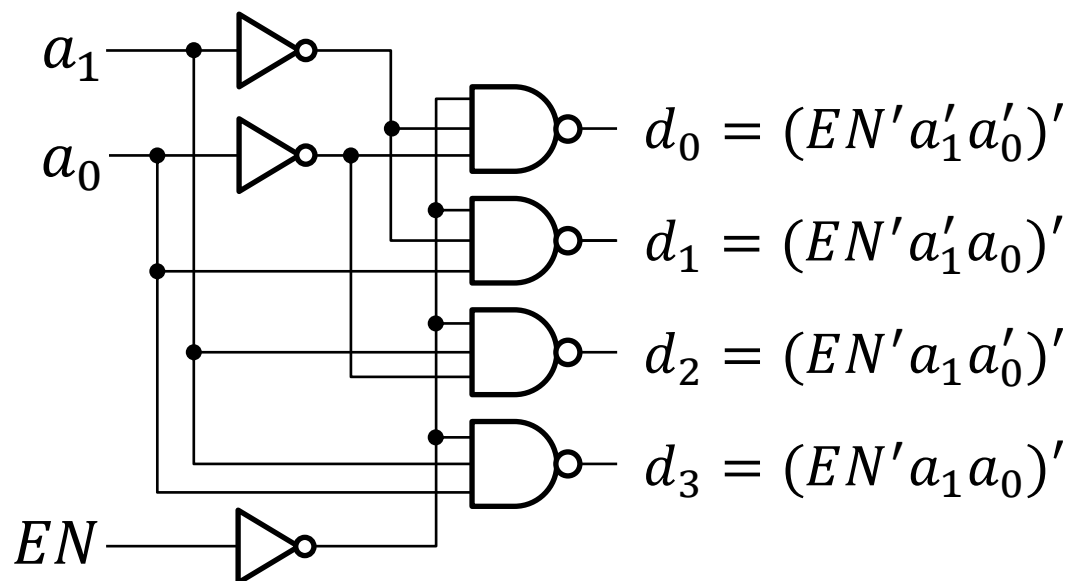
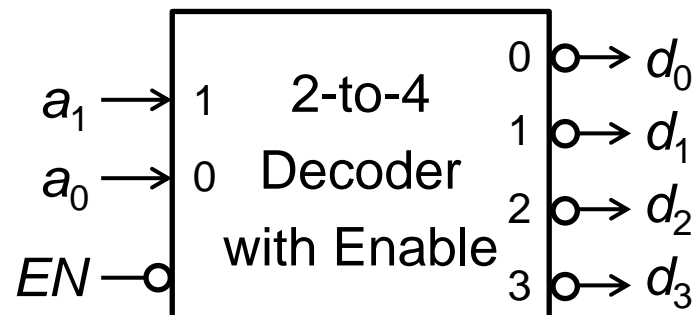
NOR gate is used for 0's

NAND Decoders with Inverted Outputs

Truth Table

Inputs		Outputs			
EN	$a_1 a_0$	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

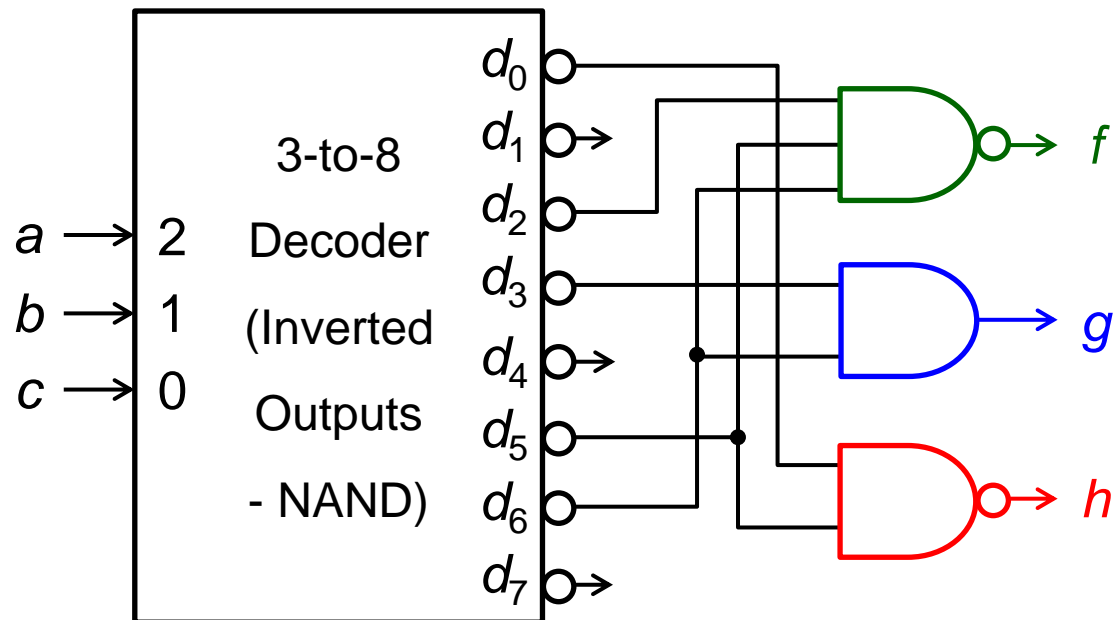
Some decoders are constructed with NAND gates. Their outputs are inverted. The Enable input is also active low (Enable if zero)



Using NAND Decoders

- ❖ NAND decoders can be used to implement functions
- ❖ Use **NAND gates** to output the minterms (if fewer ones)
- ❖ Use **AND gates** to output the maxterms (if fewer zeros)
- ❖ **Example:** $f = \Sigma(2, 5, 6)$, $g = \Pi(3, 6)$, $h = \Sigma(0, 5)$

Inputs			Outputs		
a	b	c	f	g	h
0	0	0	0	1	1
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	0	0	0
1	0	0	0	1	0
1	0	1	1	1	1
1	1	0	1	0	0
1	1	1	0	1	0



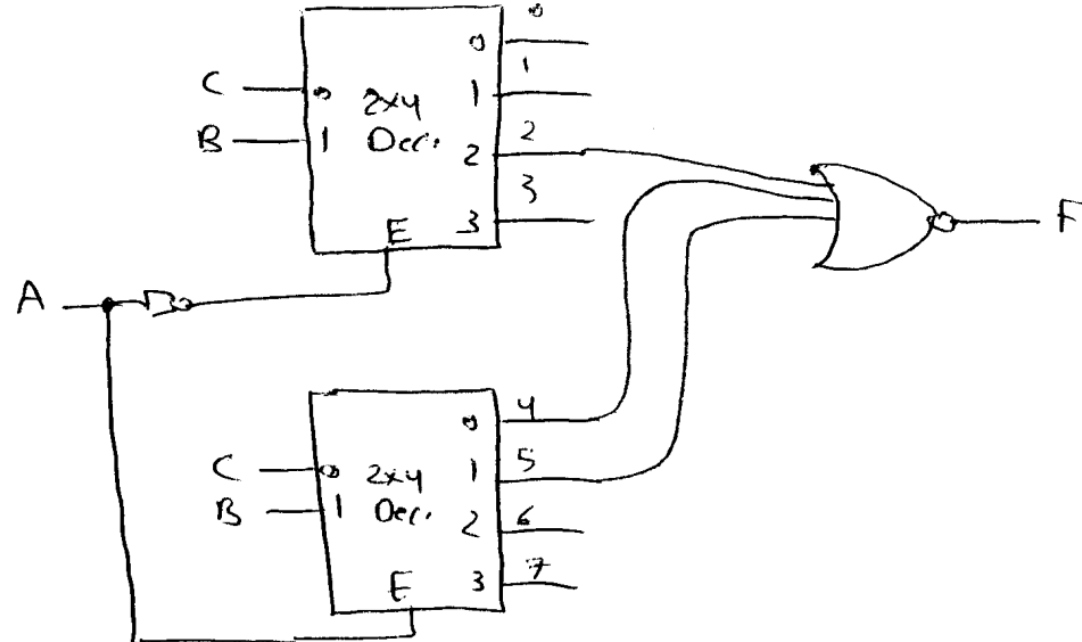
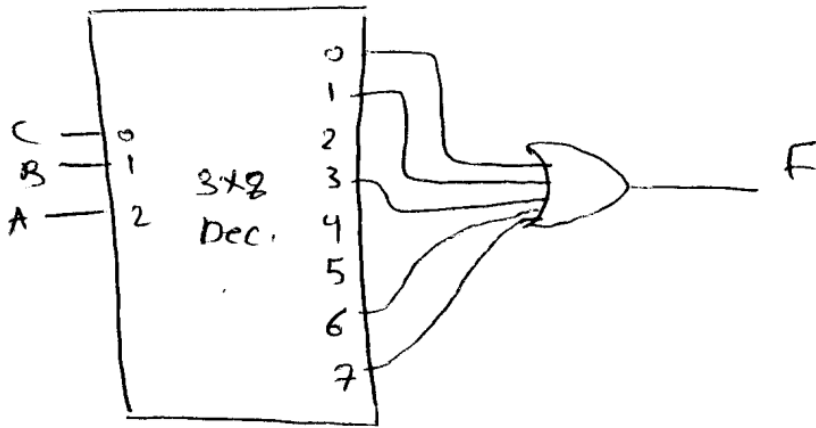
Example

❖ Implement the Boolean function: $F(A, B, C) = AB + A'C + A'B'$

a) Using a single 3x8 decoder and an OR gate.

b) Using a single NOR gate and the minimum number of 2x4 decoders with enable.

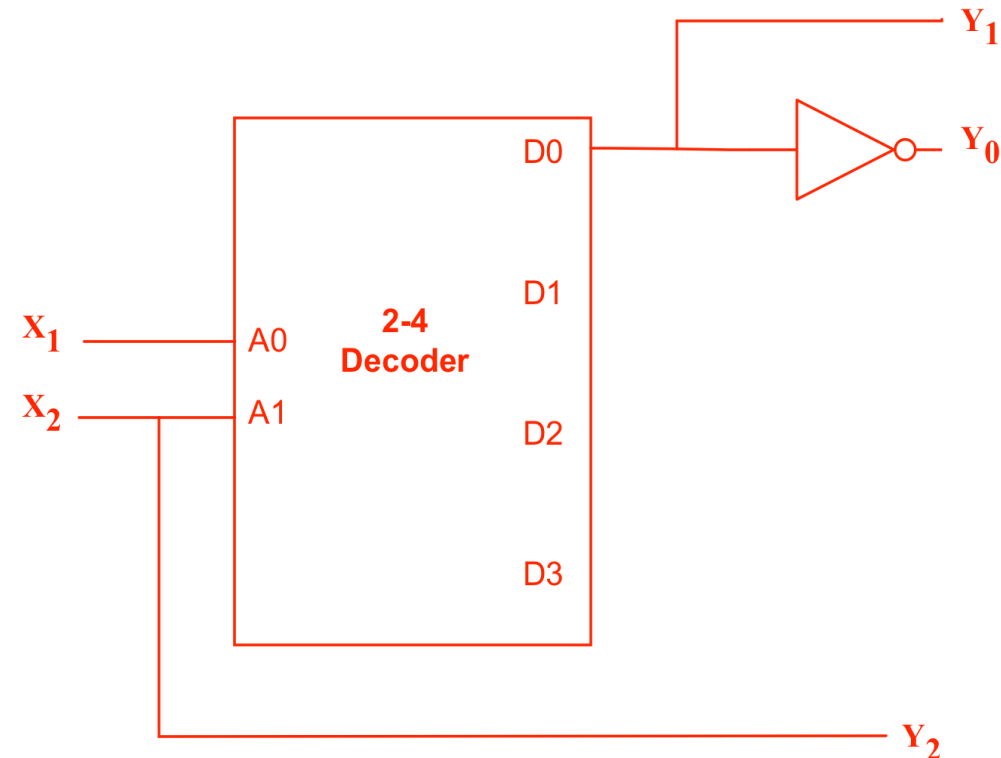
$$F = \sum m(0, 1, 3, 6, 7)$$



Example

- ❖ Consider the following truth table, in which X_2 , X_1 , and X_0 are the inputs and Y_2 , Y_1 , and Y_0 are the outputs. Using a **minimum-size decoder** and a **minimum number** of additional gates, show how to implement Y_2 , Y_1 , and Y_0 . Your additional logic gates must use the smallest possible number of inputs.

X_2	X_1	X_0	Y_2	Y_1	Y_0
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	0	0	1
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	1	0	1
1	1	1	1	0	1



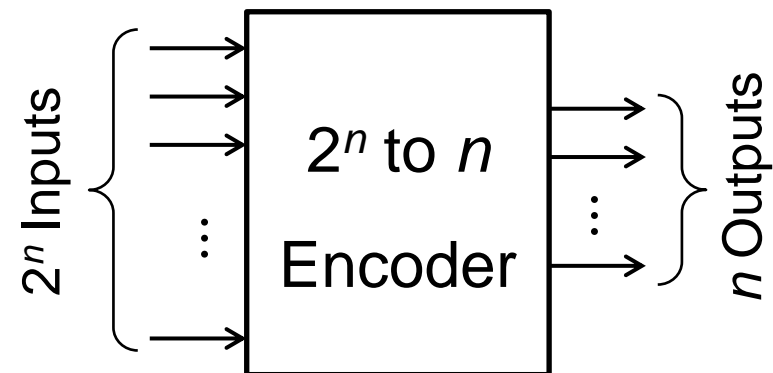
Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ **Encoders**
- ❖ Multiplexers
- ❖ Design Examples

Encoders

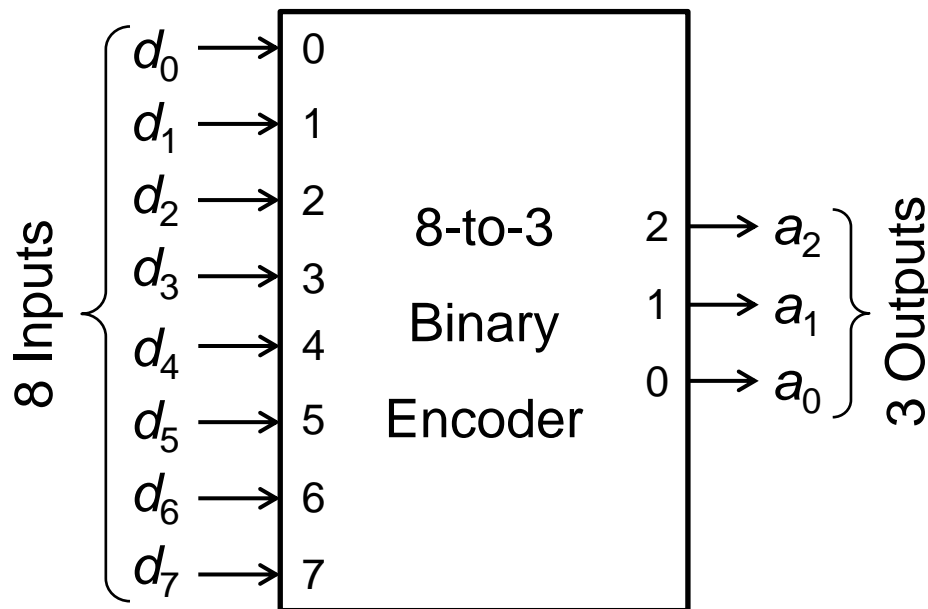
- ❖ An encoder performs the opposite operation of a decoder
- ❖ It converts a 2^n input to an n -bit output code
- ❖ The output indicates which input is active (logic **1**)
- ❖ Typically, **one** input should be **1** and all others must be **0**'s
- ❖ The conversion of input to output is called **encoding**

An encoder can have less than 2^n inputs if some input lines are unused



Example of an 8-to-3 Binary Encoder

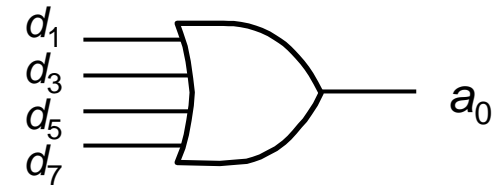
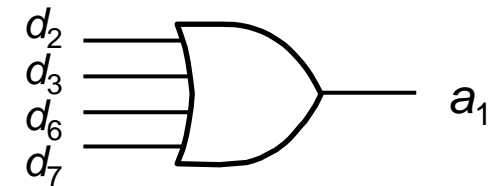
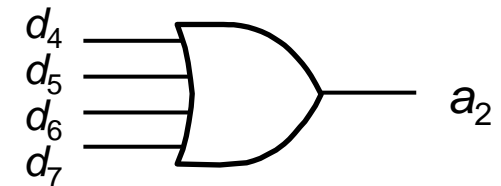
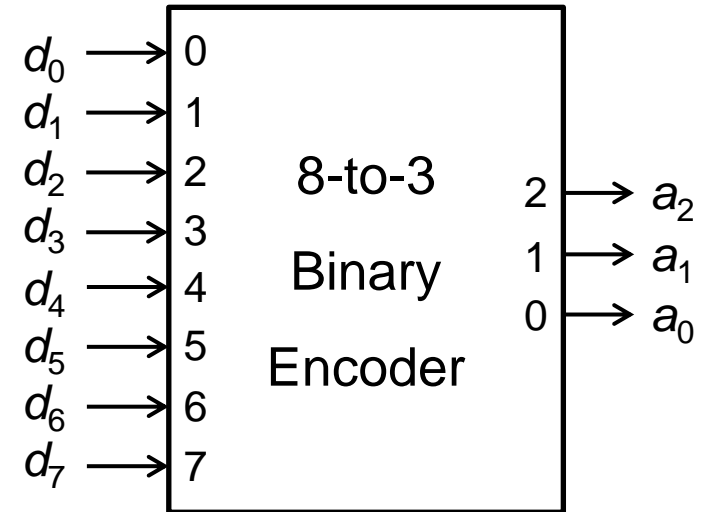
- ❖ 8 inputs, 3 outputs, only **one input** is **1**, all others are **0**'s
- ❖ Encoder generates the output binary code for the active input
- ❖ Output is **not specified** if more than one input is **1**



Inputs								Outputs		
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	a_2	a_1	a_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

8-to-3 Binary Encoder Implementation

Inputs								Outputs		
d_7	d_6	d_5	d_4	d_3	d_2	d_1	d_0	a_2	a_1	a_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1



$$a_2 = d_4 + d_5 + d_6 + d_7$$

$$a_1 = d_2 + d_3 + d_6 + d_7$$

$$a_0 = d_1 + d_3 + d_5 + d_7$$

8-to-3 binary
encoder
implemented
using three
4-input OR gates

Binary Encoder Limitations

- ❖ Exactly **one input** must be **1** at a time (all others must be **0**'s)
- ❖ If **more than one** input is **1** then the output will be **incorrect**

- ❖ For example, if $d_3 = d_6 = 1$

Then $a_2 a_1 a_0 = \mathbf{111}$ (**incorrect**)

- ❖ Two problems to resolve:

$$a_2 = d_4 + d_5 + d_6 + d_7$$

$$a_1 = d_2 + d_3 + d_6 + d_7$$

$$a_0 = d_1 + d_3 + d_5 + d_7$$

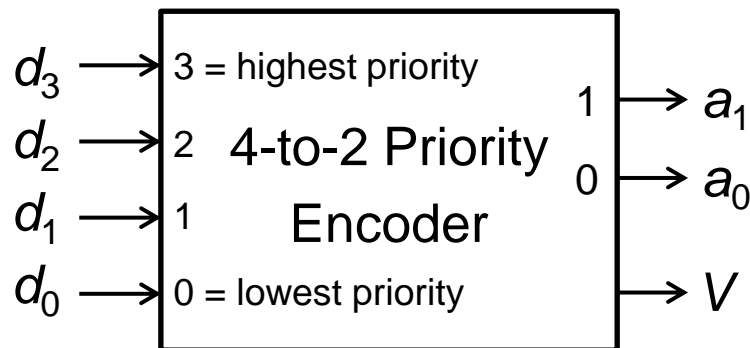
1. If **two** inputs are **1** at the same time, what should be the output?
 2. If **all** inputs are **0**'s, what should be the output?
- ❖ Output $a_2 a_1 a_0 = 000$ if $d_0 = 1$ or all inputs are 0's

How to resolve this ambiguity?

Priority Encoder

- ❖ Eliminates the two problems of the binary encoder
- ❖ Inputs are ranked from highest priority to lowest priority
- ❖ If **more than one** input is active (logic **1**) then priority is used
Output encodes the active input with higher priority
- ❖ If all inputs are zeros then the **V** (Valid) output is zero

Indicates that all inputs are zeros

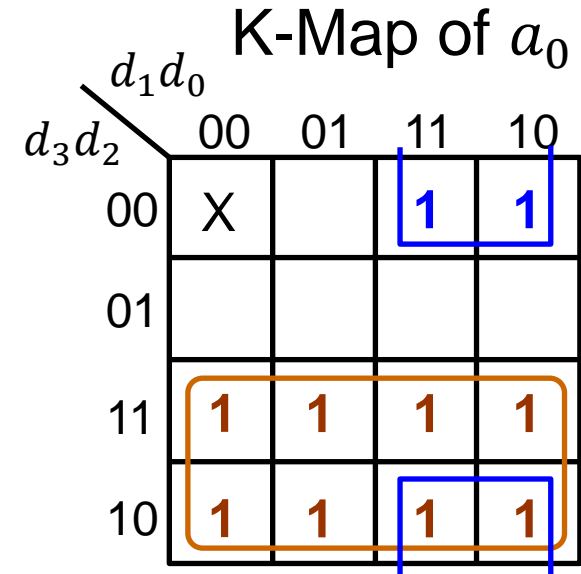
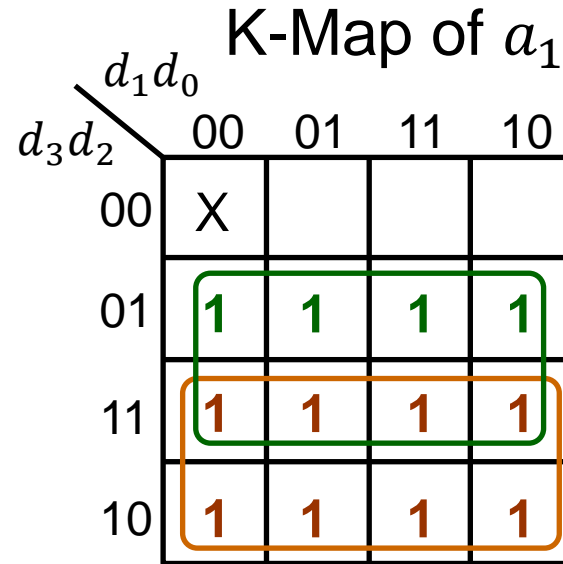


Condensed
Truth Table
All 16 cases
are listed

Inputs				Outputs		
d_3	d_2	d_1	d_0	a_1	a_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Implementing a 4-to-2 Priority Encoder

Inputs				Outputs		
d_3	d_2	d_1	d_0	a_1	a_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

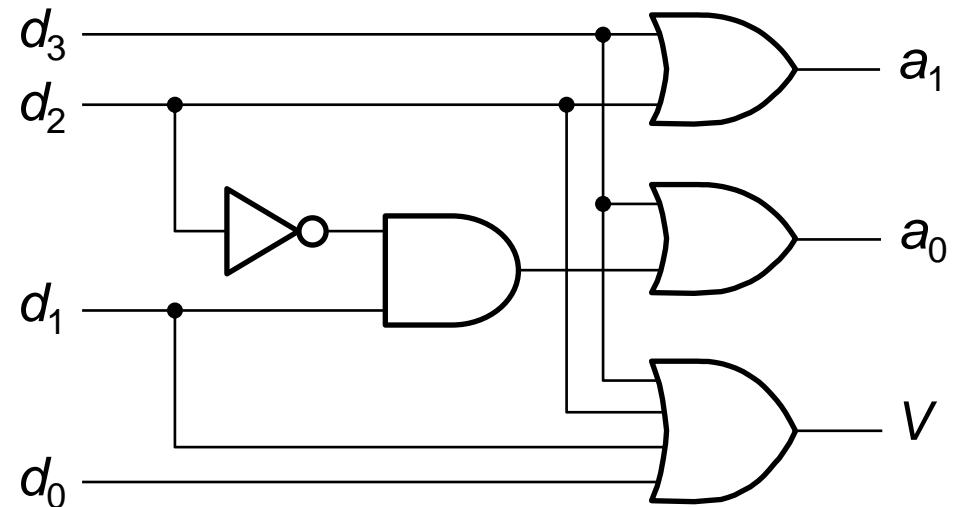


Output Expressions:

$$a_1 = d_3 + d_2$$

$$a_0 = d_3 + d_1 d_2'$$

$$V = d_3 + d_2 + d_1 + d_0$$



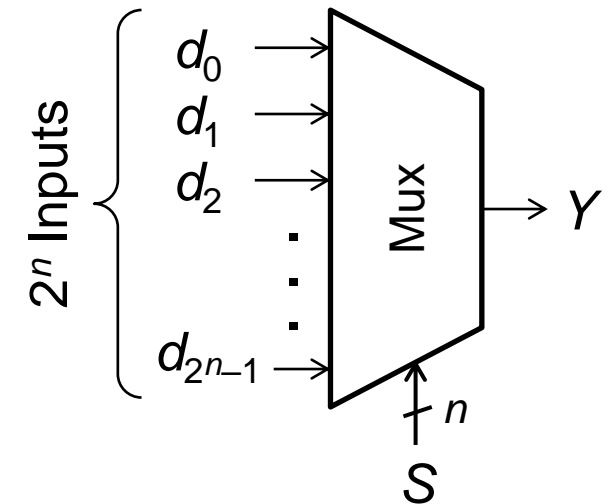
Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ Encoders
- ❖ **Multiplexers**
- ❖ Design Examples

Multiplexers

- ❖ Selecting data is an essential function in digital systems
- ❖ Functional blocks that perform selecting are called **multiplexers**
- ❖ A Multiplexer (or Mux) is a combinational circuit that has:

- ✧ Multiple data inputs (typically 2^n) to select from
- ✧ An n -bit select input S used for control
- ✧ One output Y



- ❖ The n -bit select input directs one of the data inputs to the output

Examples of Multiplexers

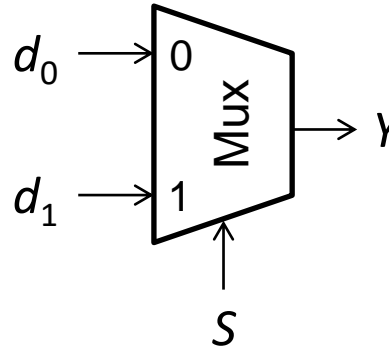
❖ 2-to-1 Multiplexer

if ($S == 0$) $Y = d_0$;

else $Y = d_1$;

Logic expression:

$$Y = d_0 S' + d_1 S$$



Inputs			Output
S	d ₀	d ₁	Y
0	0	X	0 = d ₀
0	1	X	1 = d ₀
1	X	0	0 = d ₁
1	X	1	1 = d ₁

❖ 4-to-1 Multiplexer

if ($S_1 S_0 == 00$) $Y = d_0$;

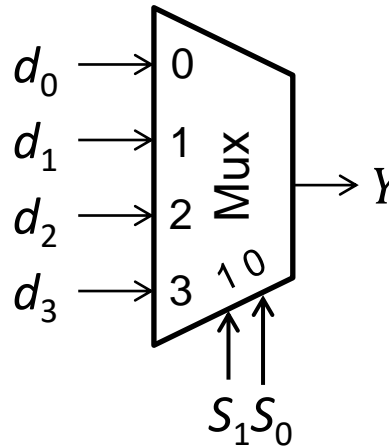
else if ($S_1 S_0 == 01$) $Y = d_1$;

else if ($S_1 S_0 == 10$) $Y = d_2$;

else $Y = d_3$;

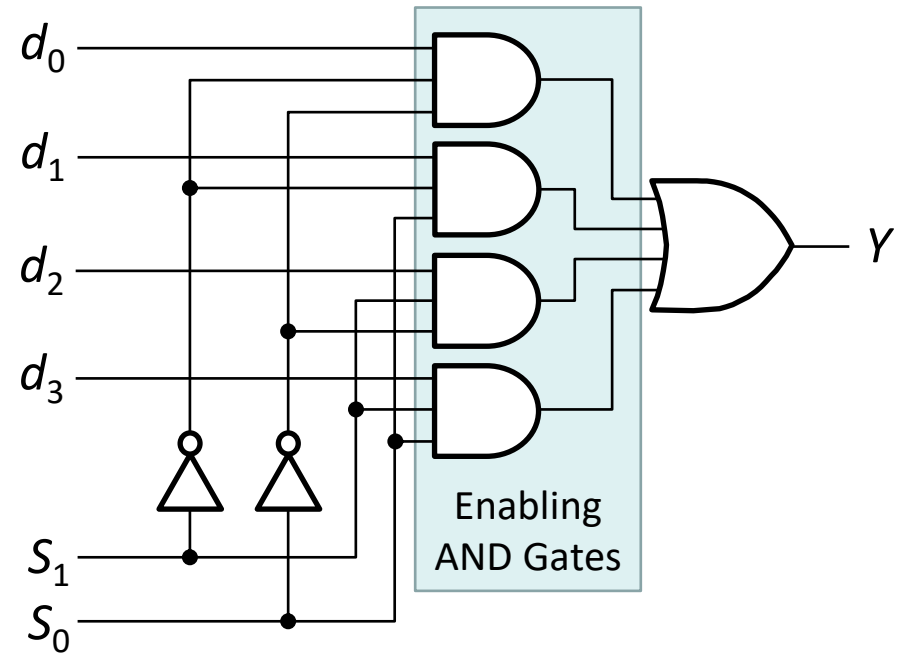
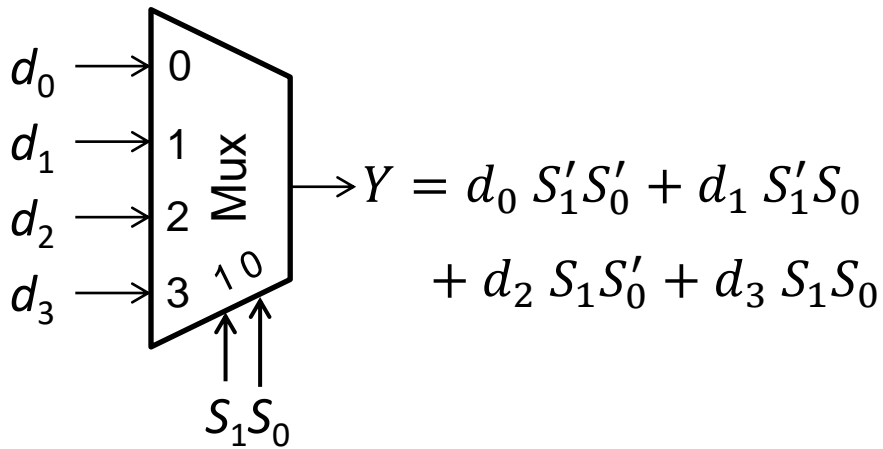
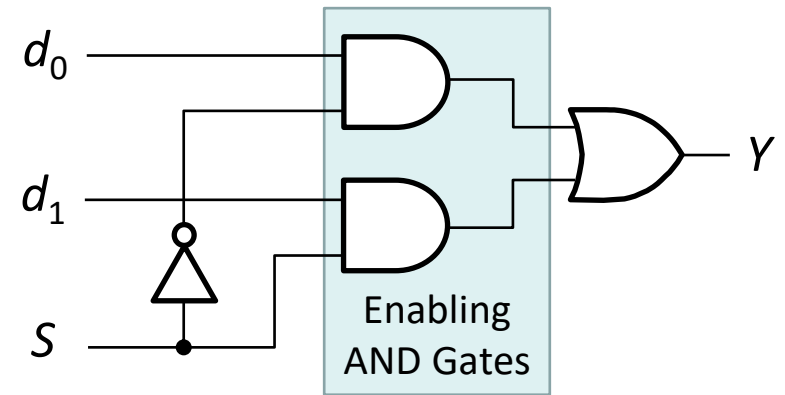
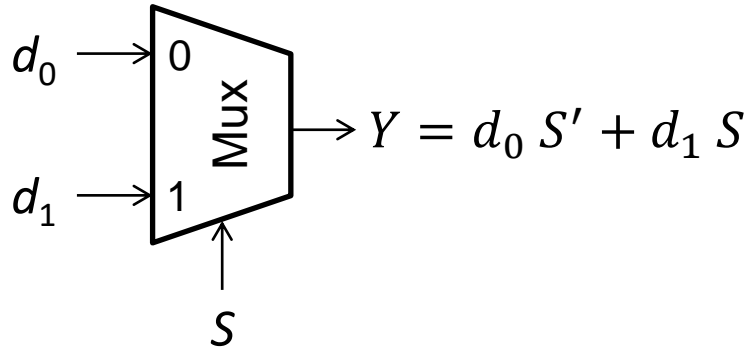
Logic expression:

$$Y = d_0 S'_1 S'_0 + d_1 S'_1 S_0 + d_2 S_1 S'_0 + d_3 S_1 S_0$$



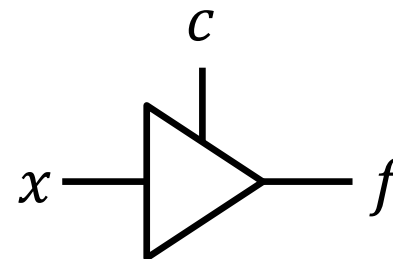
Inputs						Output
S ₁	S ₀	d ₀	d ₁	d ₂	d ₃	Y
0	0	0	X	X	X	0 = d ₀
0	0	1	X	X	X	1 = d ₀
0	1	X	0	X	X	0 = d ₁
0	1	X	1	X	X	1 = d ₁
1	0	X	X	0	X	0 = d ₂
1	0	X	X	1	X	1 = d ₂
1	1	X	X	X	0	0 = d ₃
1	1	X	X	X	1	1 = d ₃

Implementing Multiplexers



3-State Gate

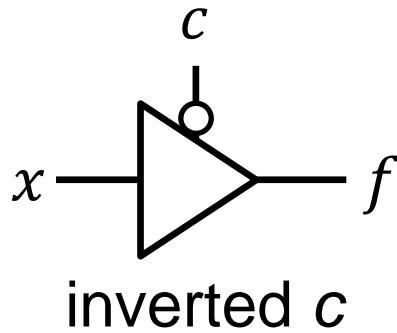
- ❖ Logic gates studied so far have two outputs: 0 and 1
- ❖ Three-State gate has three possible outputs: **0, 1, Z**
 - ✧ **Z** is the **Hi-Impedance** output
 - ✧ **Z** means that the output is **disconnected** from the input
 - ✧ Gate behaves as an **open switch** between input and output
- ❖ Input **c** connects input to output
 - ✧ **c** is the control (enable) input
 - ✧ If **c** is **0** then **f = Z**
 - ✧ If **c** is **1** then **f = input x**



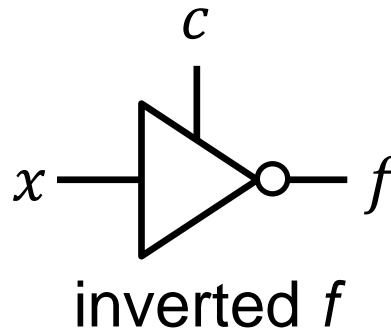
c	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

Variations of the 3-State Gate

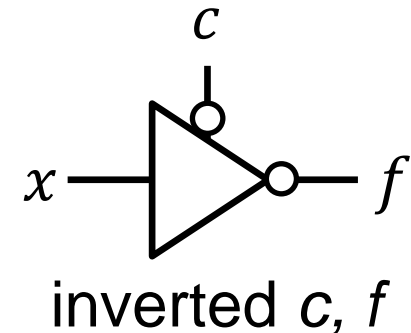
- ❖ Control input **c** and output **f** can be inverted
- ❖ A bubble is inserted at the input **c** or output **f**



c	x	f
0	0	0
0	1	1
1	0	Z
1	1	Z



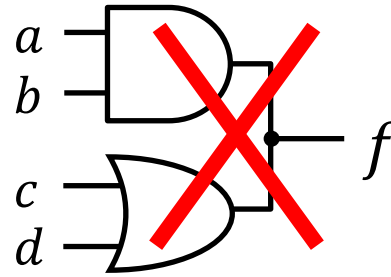
c	x	f
0	0	Z
0	1	Z
1	0	1
1	1	0



c	x	f
0	0	1
0	1	0
1	0	Z
1	1	Z

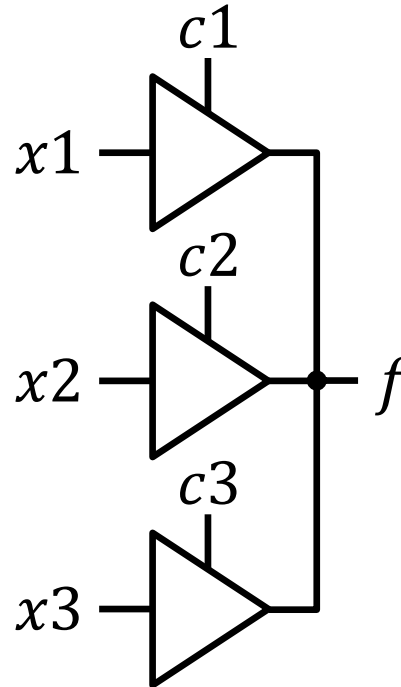
Wired Output

Logic gates with 0 and 1 outputs **cannot** have their outputs wired together



This will result in a **short circuit** that will burn the gates

3-state gates **can wire** their outputs together



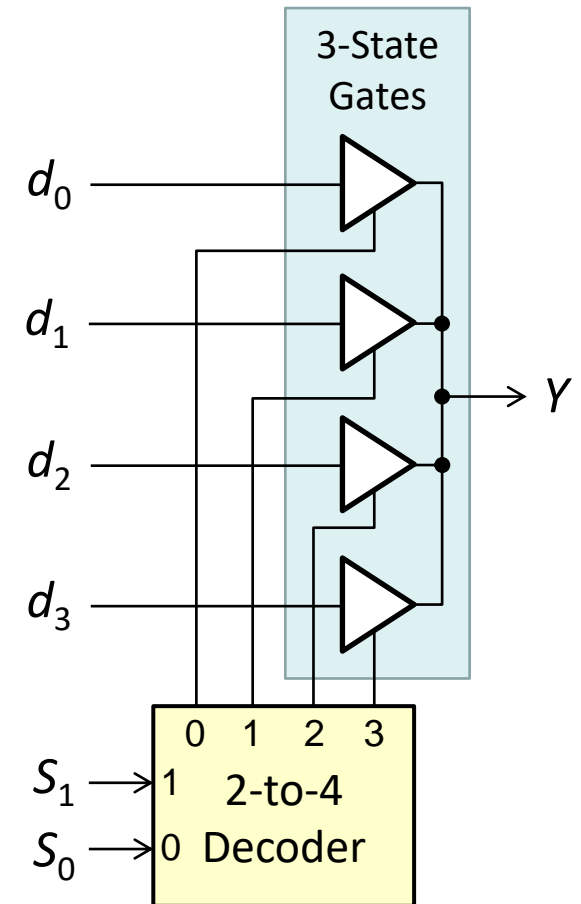
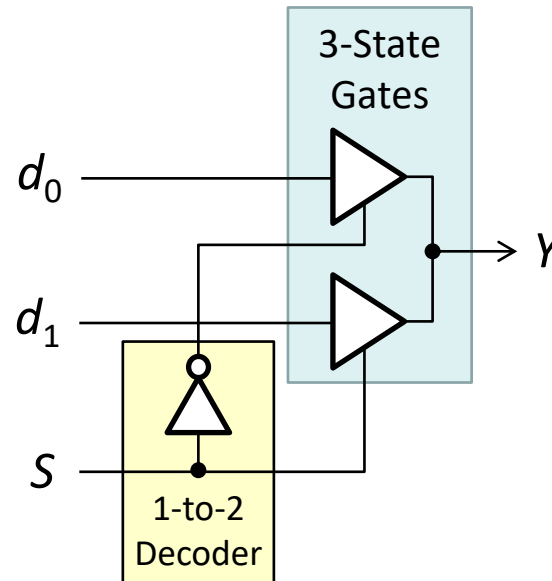
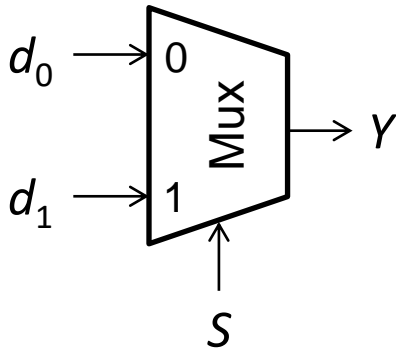
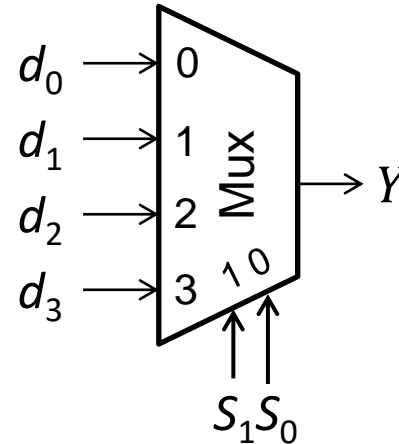
At most one 3-state gate can be enabled at a time
Otherwise, conflicting outputs will burn the circuit

c1	c2	c3	f
0	0	0	z
1	0	0	x1
0	1	0	x2
0	0	1	x3
0	1	1	Burn
1	0	1	Burn
1	1	0	Burn
1	1	1	Burn

Implementing Multiplexers with 3-State Gates

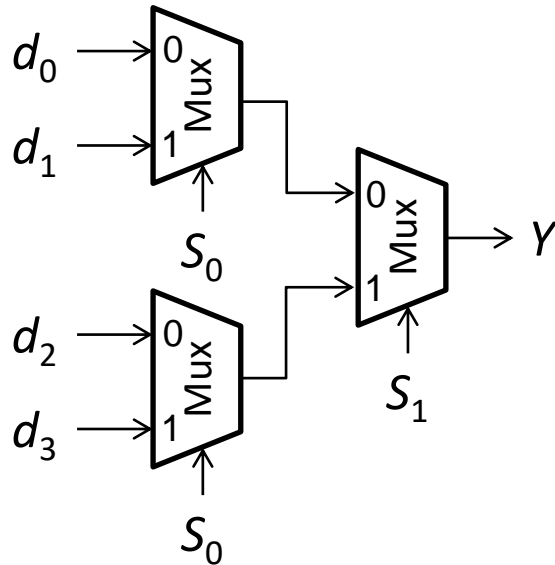
A Multiplexer can also be implemented using:

1. A decoder
2. Three-state gates

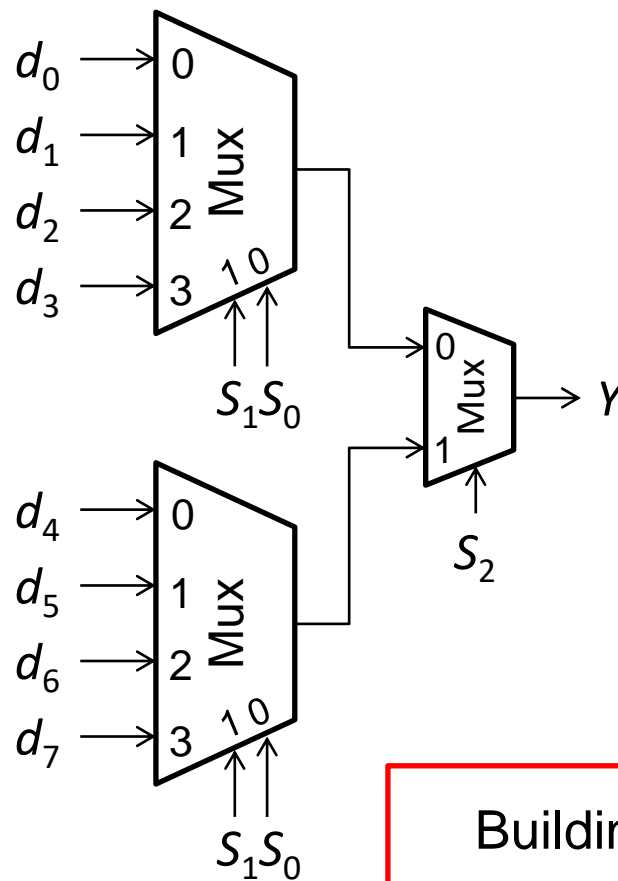


Building Larger Multiplexers

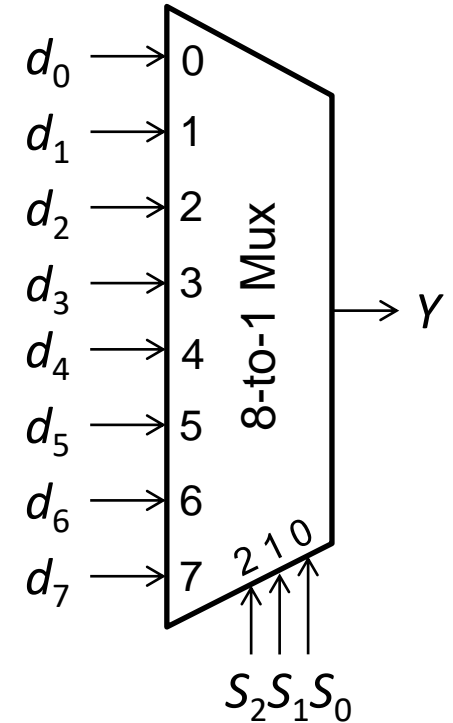
Larger multiplexers can be built hierarchically using smaller ones



Building 4-to-1
Mux using three
2-to-1 Muxes

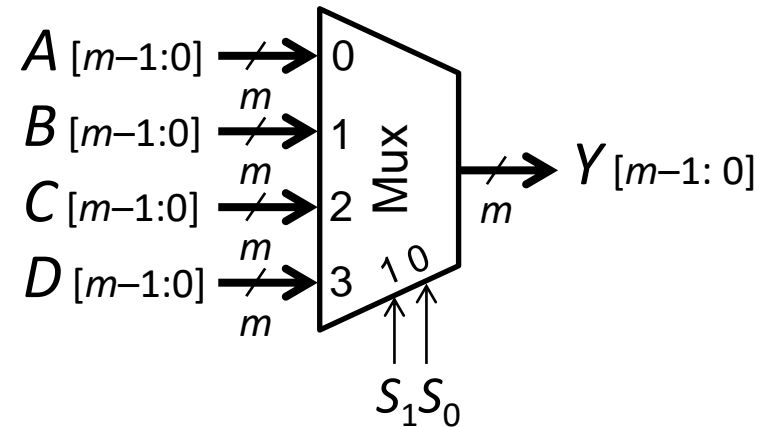
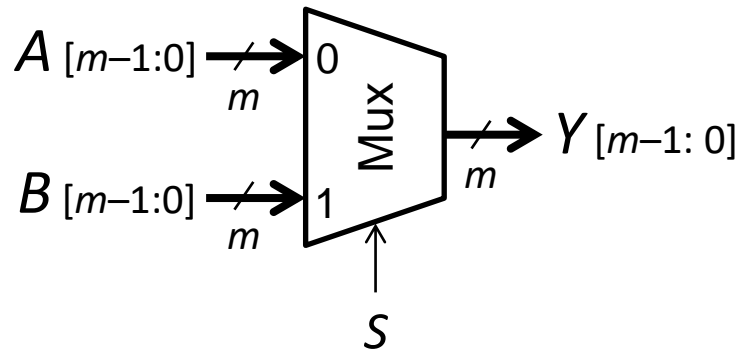


Building 8-to-1 Mux
using two 4-to-1 Muxes
and a 2-to-1 Mux



Multiplexers with Vector Input and Output

The inputs and output of a multiplexer can be m -bit vectors



2-to-1 Multiplexer with m bits
Inputs and output are m -bit vectors
Using m copies of a 2-to-1 Mux

4-to-1 Multiplexer with m bits
Inputs and output are m -bit vectors
Using m copies of a 4-to-1 Mux

Implementing a Function with a Multiplexer

- ❖ A Multiplexer can be used to implement any logic function
- ❖ The function must be expressed using its minterms
- ❖ Example: Implement $F(a, b, c) = \Sigma(1, 2, 6, 7)$ using a Mux

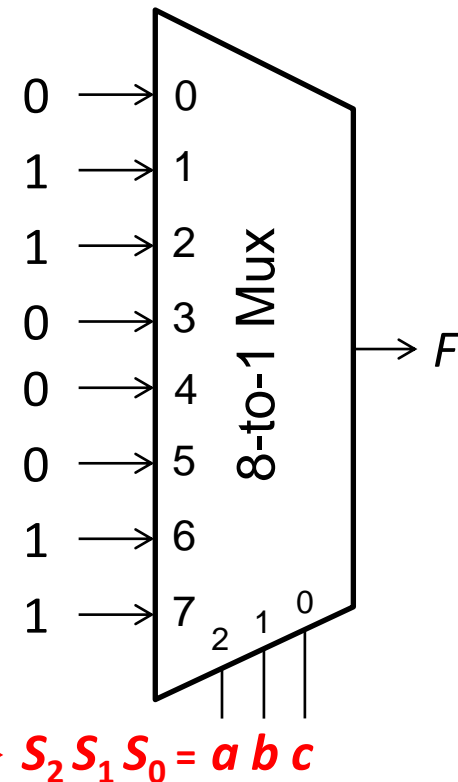
❖ Solution:

The inputs are used as select lines to a Mux.

An 8-to-1

Mux is used because there are 3 variables

Inputs			Output
a	b	c	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

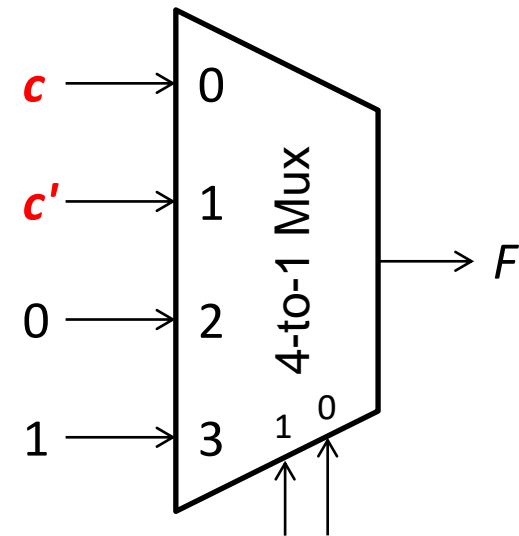


$S_2 S_1 S_0 = a b c$

Better Solution with a Smaller Multiplexer

- ❖ Re-implement $F(a, b, c) = \Sigma(1, 2, 6, 7)$ using a 4-to-1 Mux
- ❖ We will use the two select lines for variables a and b
- ❖ Variable c and its complement are used as inputs to the Mux

Inputs			Output	Comment
a	b	c	F	F
0	0	0	0	$F = c$
0	0	1	1	
0	1	0	1	$F = c'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

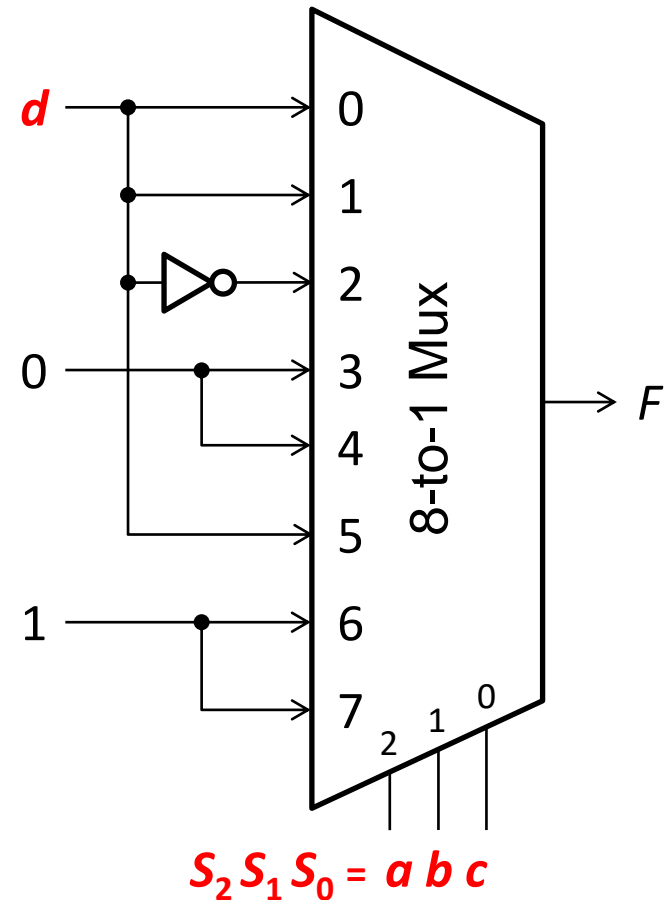
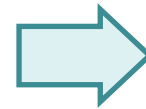


$S_1 S_0 = ab$

Implementing Functions: Example 2

Implement $F(a, b, c, d) = \Sigma(1,3,4,11,12,13,14,15)$ using 8-to-1 Mux

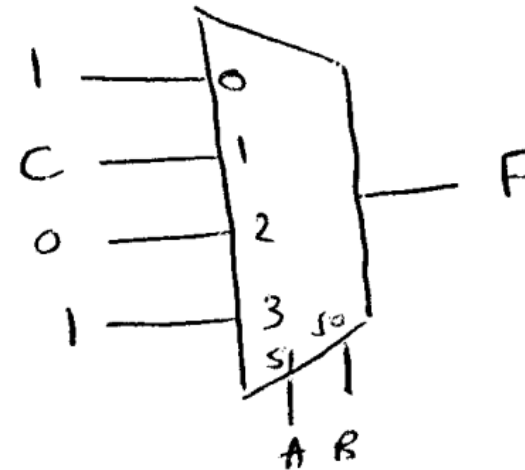
Inputs				Output	Comment
a	b	c	d	F	F
0	0	0	0	0	$F = d$
0	0	0	1	1	
0	0	1	0	0	$F = d$
0	0	1	1	1	
0	1	0	0	1	$F = d'$
0	1	0	1	0	
0	1	1	0	0	$F = 0$
0	1	1	1	0	
1	0	0	0	0	$F = 0$
1	0	0	1	0	
1	0	1	0	0	$F = d$
1	0	1	1	1	
1	1	0	0	1	$F = 1$
1	1	0	1	1	
1	1	1	0	1	$F = 1$
1	1	1	1	1	



Implementing Functions: Example 3

- ❖ Implement the Boolean function: $F(A, B, C) = AB + A'C + A'B'$
- a) Using a single 4x1 multiplexer.

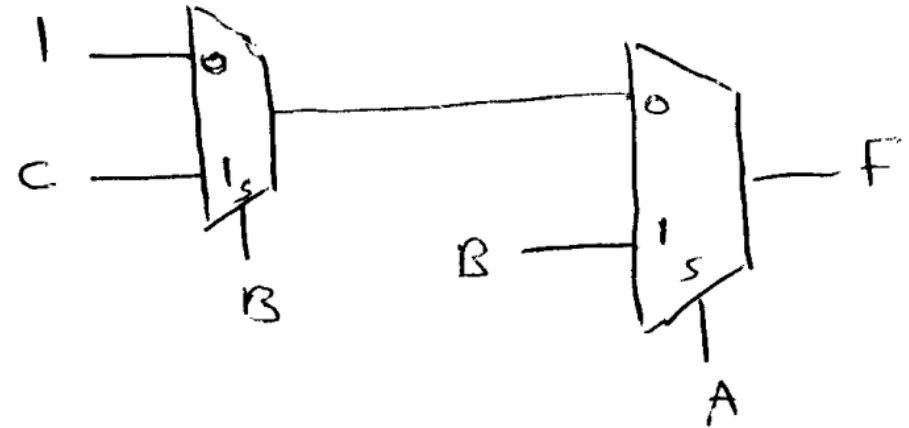
Inputs			Output	Comment
A	B	C	F	F
0	0	0	1	$F = 1$
0	0	1	1	
0	1	0	0	$F = C$
0	1	1	1	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	



Implementing Functions: Example 3

- ❖ Implement the Boolean function: $F(A, B, C) = AB + A'C + A'B'$
 - b) Using a minimum number of 2x1 multiplexers.

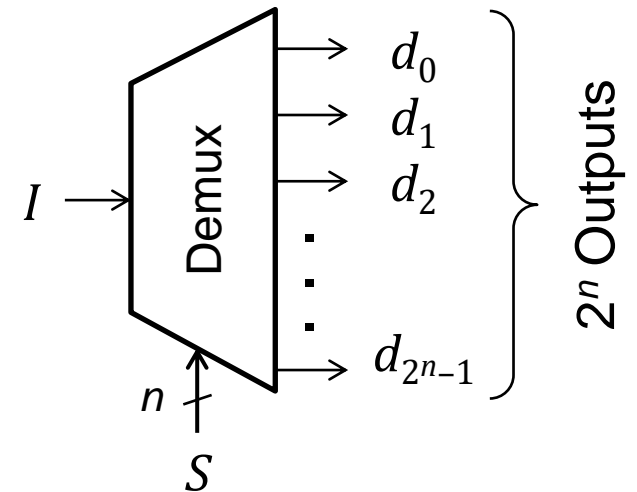
Inputs			Output	Comment
A	B	C	F	F
0	0	0	1	$F = 1$
0	0	1	1	
0	1	0	0	$F = C$
0	1	1	1	
1	0	0	0	$F = B$
1	0	1	0	
1	1	0	1	
1	1	1	1	



Demultiplexer

- ❖ Performs the inverse operation of a Multiplexer
- ❖ A Demultiplexer (or Demux) is a combinational circuit that has:

1. One data input I
2. An n -bit select input S
3. A maximum of 2^n data outputs



- ❖ The Demux directs the data input to one of the outputs

According to the select input S

Examples of Demultiplexers

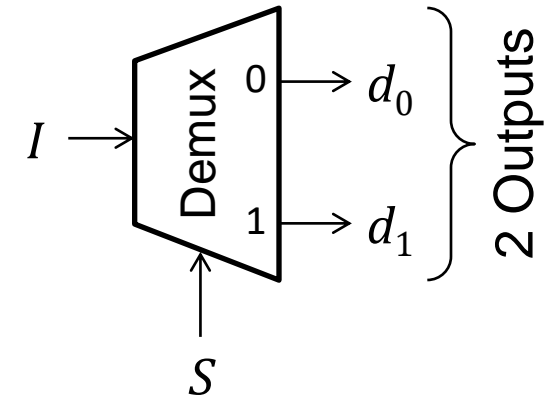
❖ 1-to-2 Demultiplexer

if ($S == 0$) { $d_0 = I$; $d_1 = 0$; }

else { $d_1 = I$; $d_0 = 0$; }

Output expressions:

$$d_0 = I S'; d_1 = I S$$



❖ 1-to-4 Demultiplexer

if ($S_1 S_0 == 00$) { $d_0 = I$; $d_1 = d_2 = d_3 = 0$; }

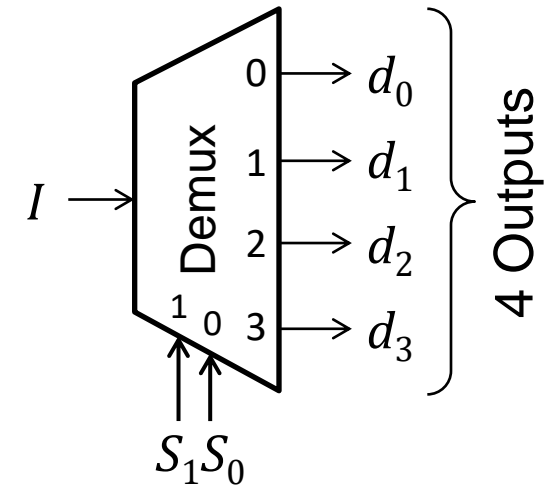
else if ($S_1 S_0 == 01$) { $d_1 = I$; $d_0 = d_2 = d_3 = 0$; }

else if ($S_1 S_0 == 10$) { $d_2 = I$; $d_0 = d_1 = d_3 = 0$; }

else { $d_3 = I$; $d_0 = d_1 = d_2 = 0$; }

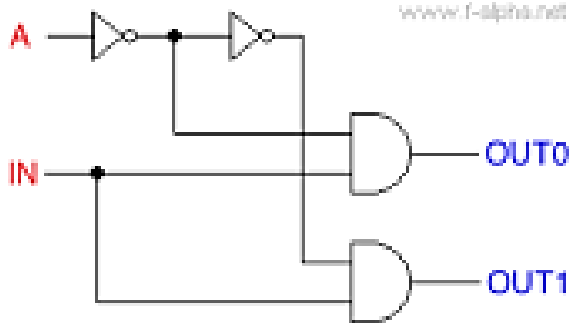
Output expressions:

$$d_0 = I S'_1 S'_0 ; d_1 = I S'_1 S_0 ; d_2 = I S_1 S'_0 ; d_3 = I S_1 S_0$$

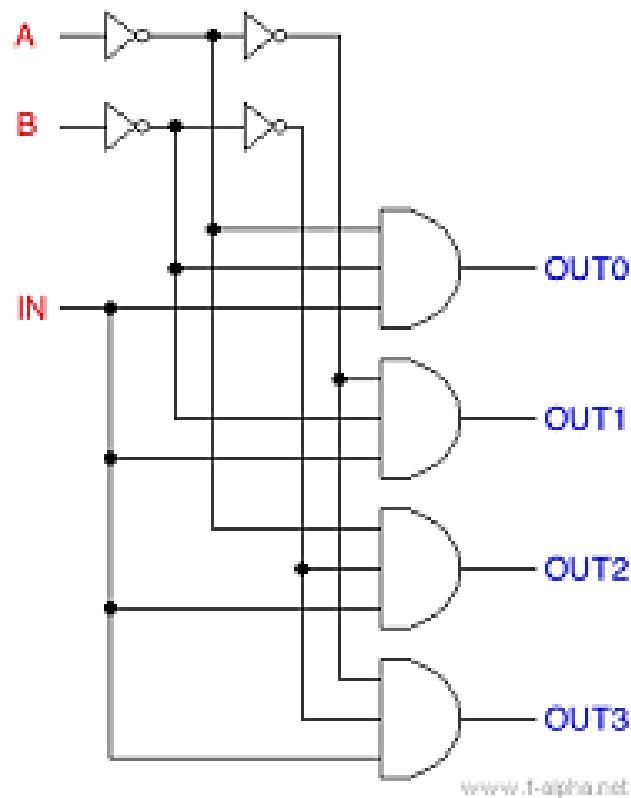


Examples of Demultiplexers

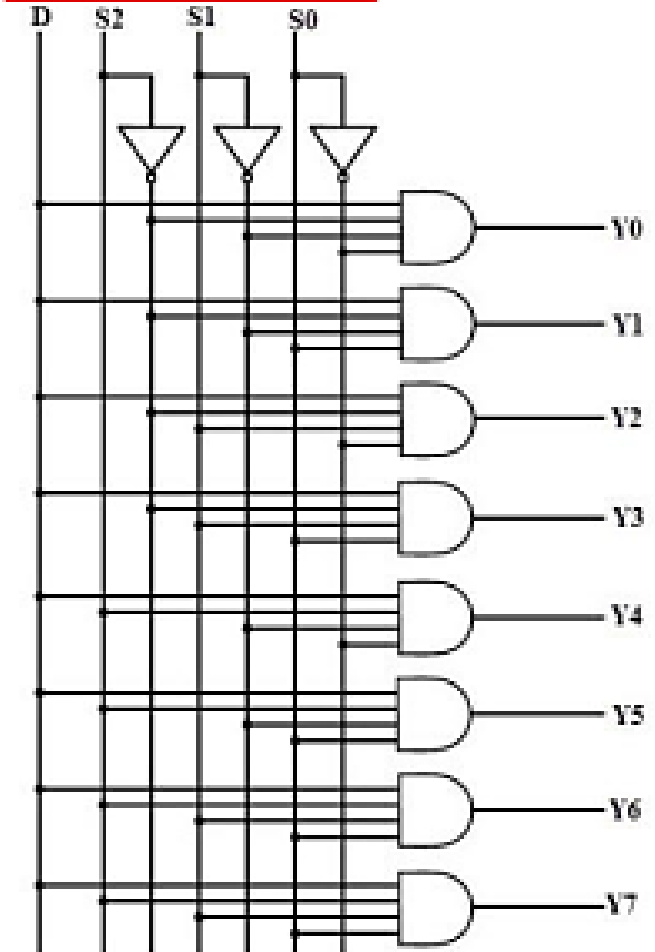
1 - 2



1 - 4



1 - 8



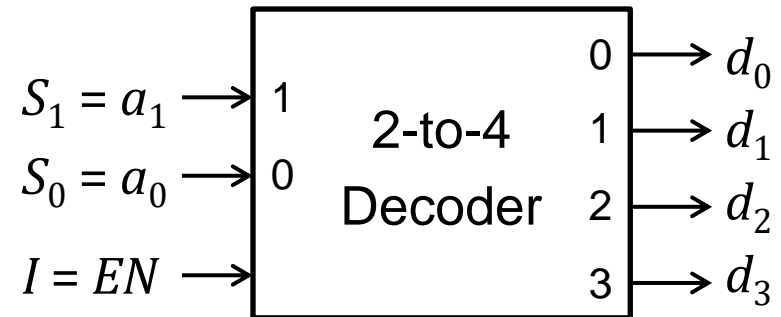
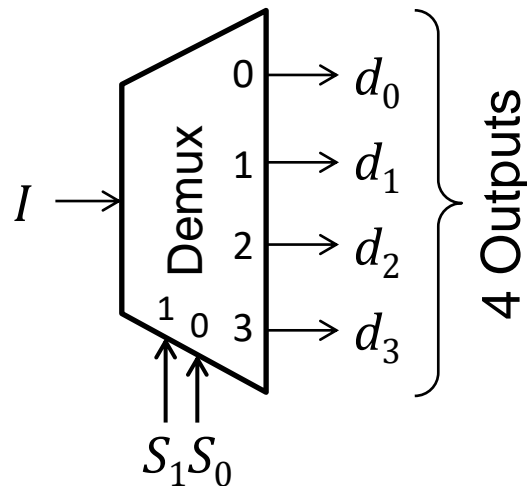
Demultiplexer = Decoder with Enable

❖ A 1-to-4 demux is equivalent to a 2-to-4 decoder with enable

Demux select input S_1 is equivalent to Decoder input a_1

Demux select input S_0 is equivalent to Decoder input a_0

Demux Input I is equivalent to Decoder Enable EN



Think of a decoder as directing the Enable signal to one output

❖ In general, a demux with n select inputs and 2^n outputs is equivalent to a n -to- 2^n decoder with enable input

Next . . .

- ❖ Combinational Circuits
- ❖ Analysis Procedure
- ❖ Design Procedure
- ❖ Binary Adder-Subtractor
- ❖ Decimal Adder
- ❖ Binary Multiplier
- ❖ Magnitude Comparator
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ **Design Examples**

2-by-2 Crossbar Switch

❖ A 2×2 crossbar switch is a combinational circuit that has:

Two m -bit Inputs: A and B

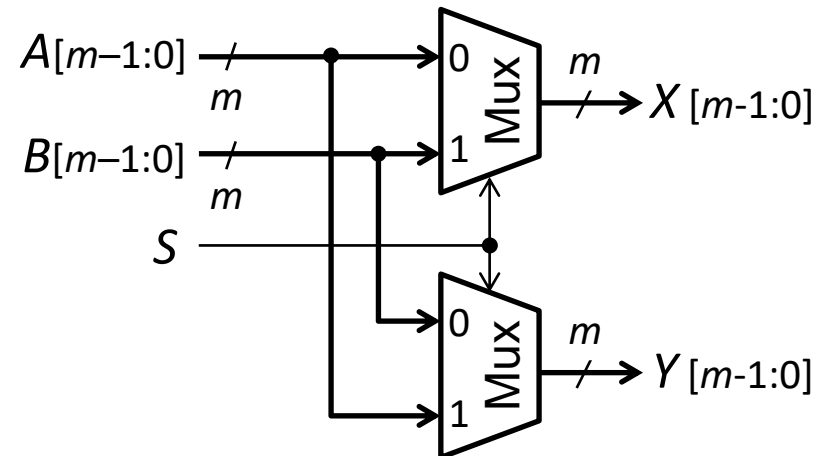
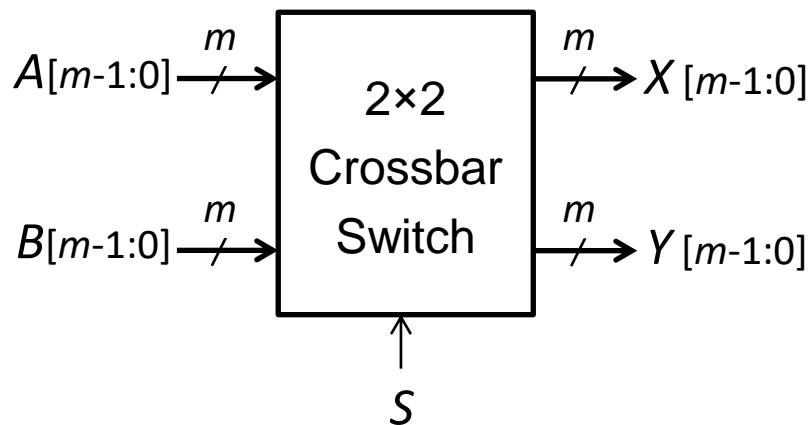
Two m -bit outputs: X and Y

1-bit select input S

```
if (S == 0) { X = A; Y = B; }  
else { X = B; Y = A; }
```

❖ Implement the 2×2 crossbar switch using multiplexers

❖ **Solution:** Two 2-input multiplexers are used



Sorting Two Unsigned Integers

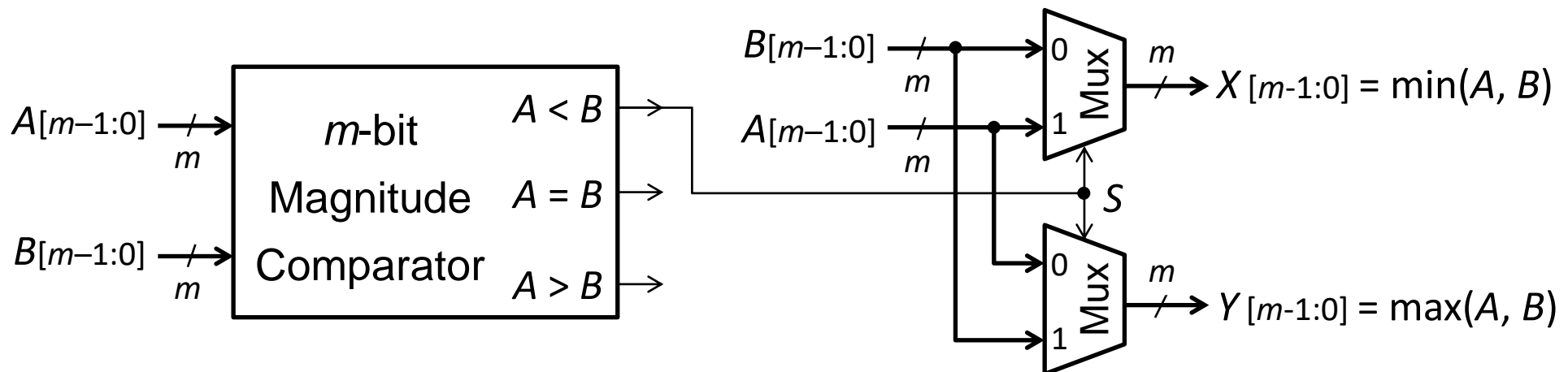
❖ Design a circuit that sorts two m -bit unsigned integers A and B

Inputs: Two m -bit unsigned integers A and B

Outputs: $X = \min(A, B)$ and $Y = \max(A, B)$

❖ Solution:

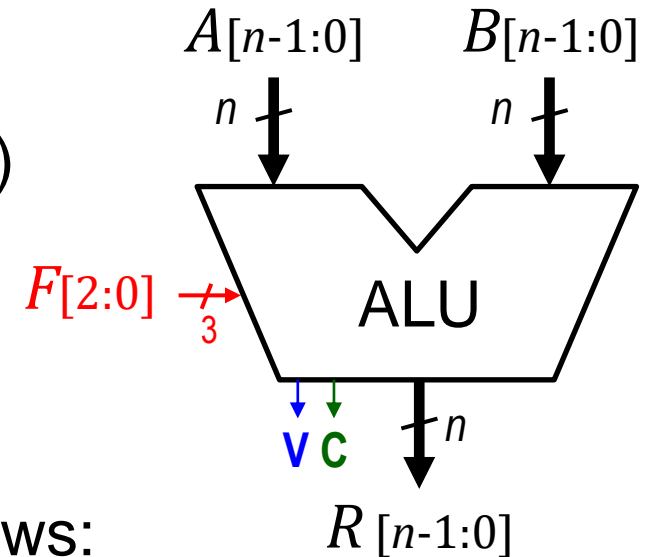
We will use a magnitude comparator to compare A with B , and 2×2 crossbar switch implemented using two 2-input multiplexers



Arithmetic and Logic Unit (ALU)

- ❖ Can perform many functions
- ❖ Most common ALU functions
 - Arithmetic functions: ADD, SUB (Subtract)
 - Logic functions: AND, OR, XOR, etc.
- ❖ We will design an ALU with 8 functions
- ❖ The function F is coded with 3 bits as follows:

ALU Symbol



Function	ALU Result	Function	ALU Result
$F = 000$ (ADD)	$R = A + B$	$F = 100$ (AND)	$R = A \& B$
$F = 001$ (ADD + 1)	$R = A + B + 1$	$F = 101$ (OR)	$R = A B$
$F = 010$ (SUB - 1)	$R = A - B - 1$	$F = 110$ (NOR)	$R = \sim(A B)$
$F = 011$ (SUB)	$R = A - B$	$F = 111$ (XOR)	$R = (A \wedge B)$

Designing a Simple ALU

