Combinational Logic Design

Presentation Outline

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

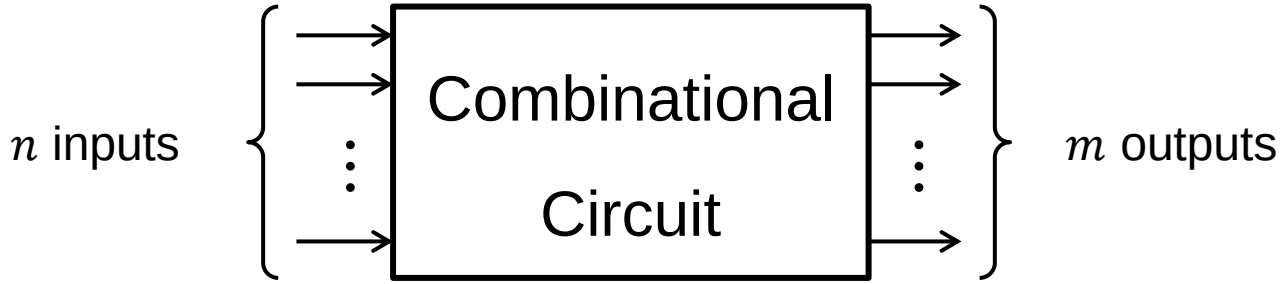
Combinational Circuit

- A combinational circuit is a block of logic gates having:

  n inputs: $x_1, x_2, \ldots, x_n$

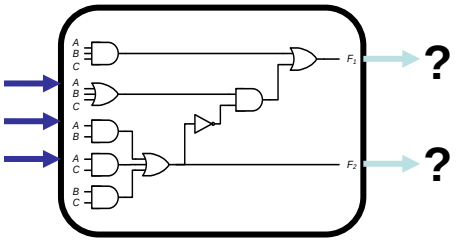
  m outputs: $f_1, f_2, \ldots, f_m$

- 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

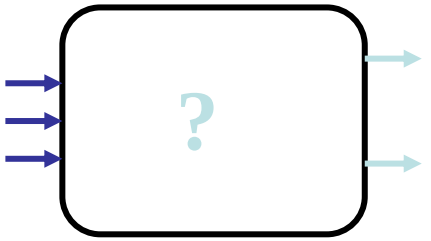


Combinational Circuits

- Analysis
  - Given a circuit, find out its *function*
  - Function may be expressed as:
    - Boolean function
    - Truth table



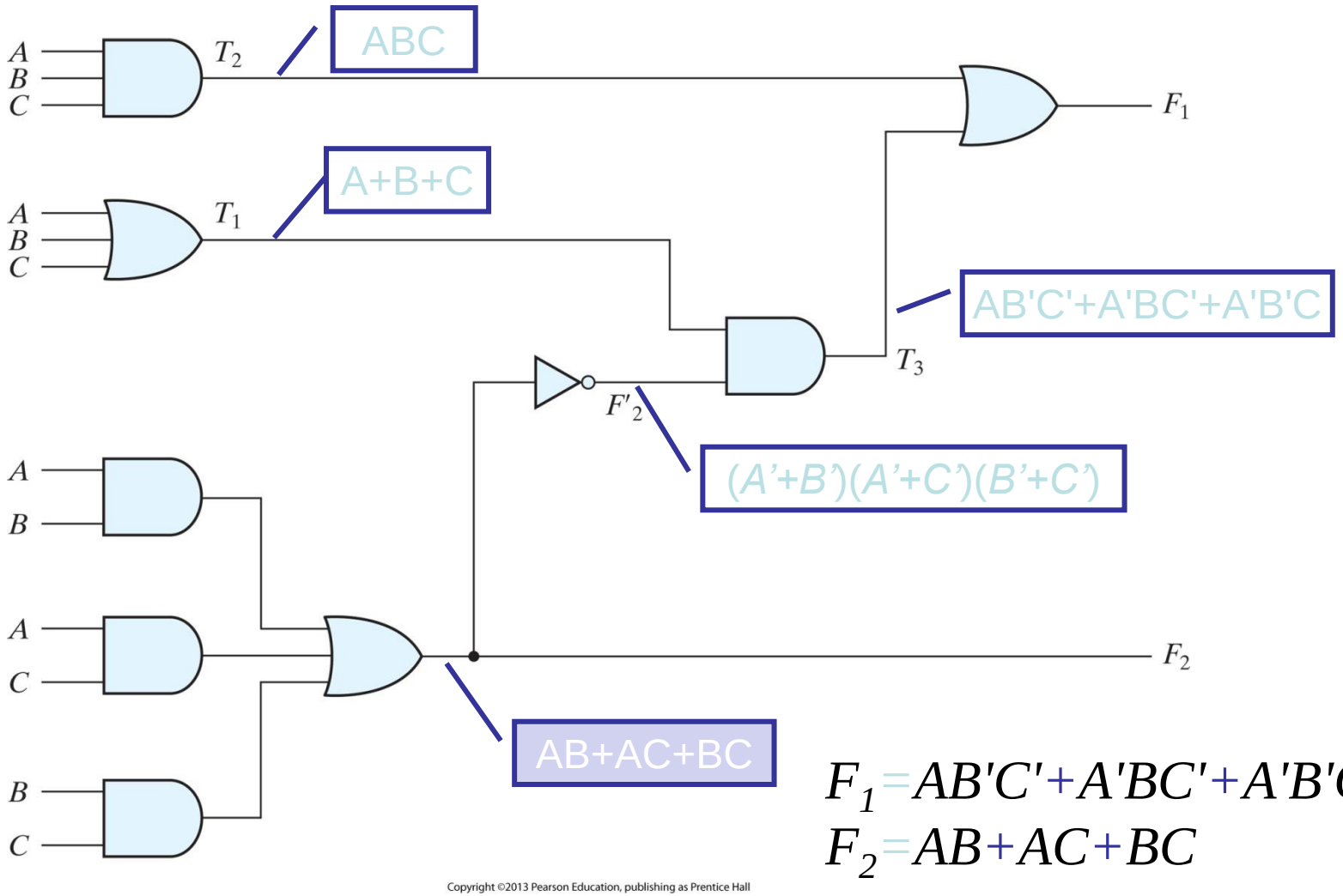
- Design
  - Given a desired function, determine its *circuit*
  - Function may be expressed as:
    - Boolean function
    - Truth table



ANALYSIS PROCEDURE

1. Label all gate outputs that are a function of input variables. Determine the Boolean function for each gate output
2. Label the gates that are a function of input variables and previously labeled gates. 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



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

$$F_2 = AB + AC + BC$$

Analysis Procedure

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 a determined

Analysis Procedure

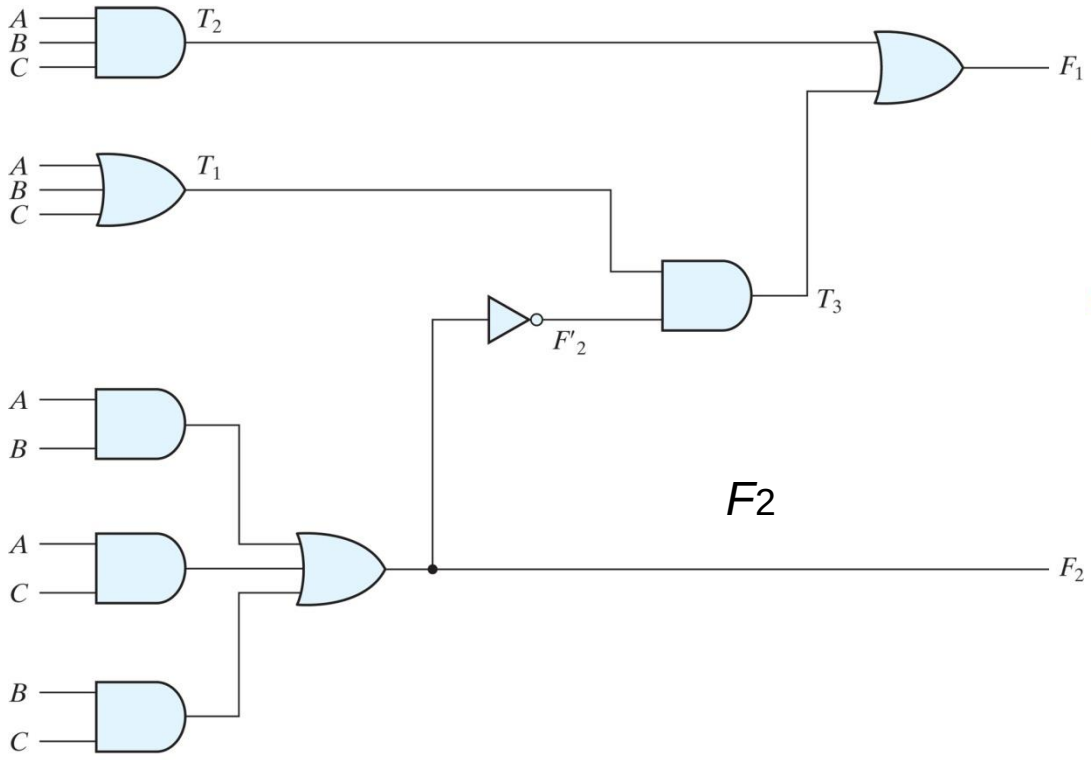
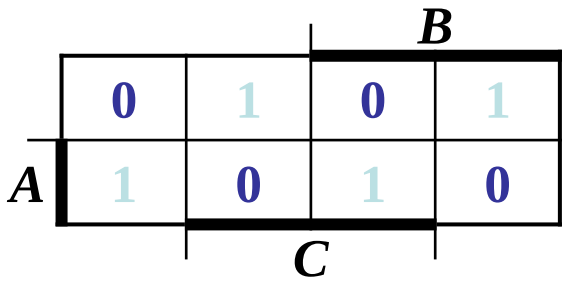
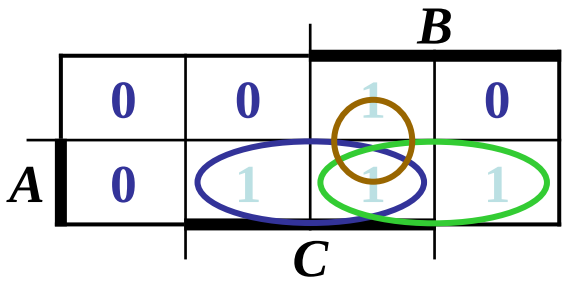


Table 4.1
Truth Table for the Logic Diagram of Fig. 4.2

A	B	C	F_2	F'_2	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1



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



$F_2 = AB + AC + BC$

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

Designing a BCD to Excess-3 Code Converter

1. Specification

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

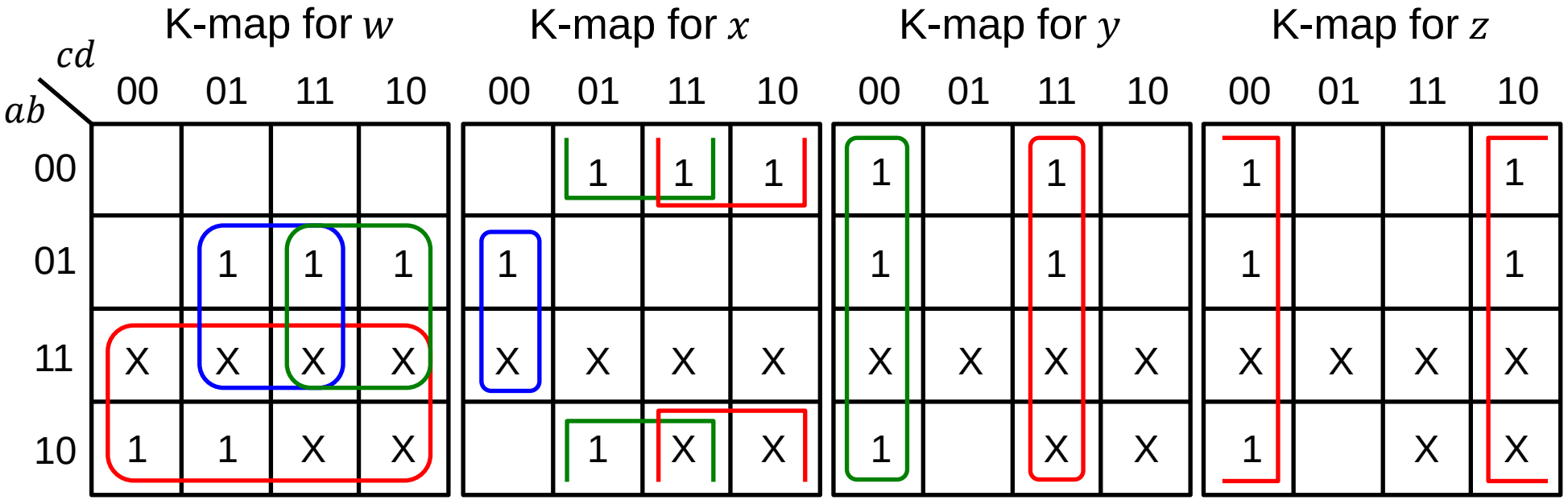
2. Formulation

- ✧ 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-Product expressions:

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

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

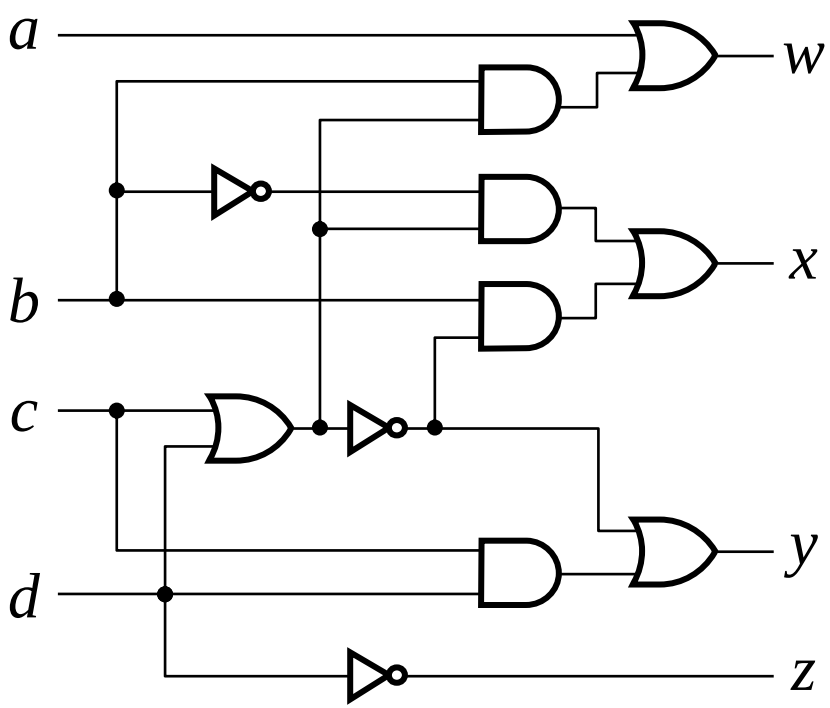
$w = a + b(c + d)$, $x = b'(c + d) + b(c + d)'$, $y = cd + (c + d)'$

Designing a BCD to Excess-3 Code Converter

4. Technology Mapping

Draw a logic diagram using ANDs, ORs, and inverters

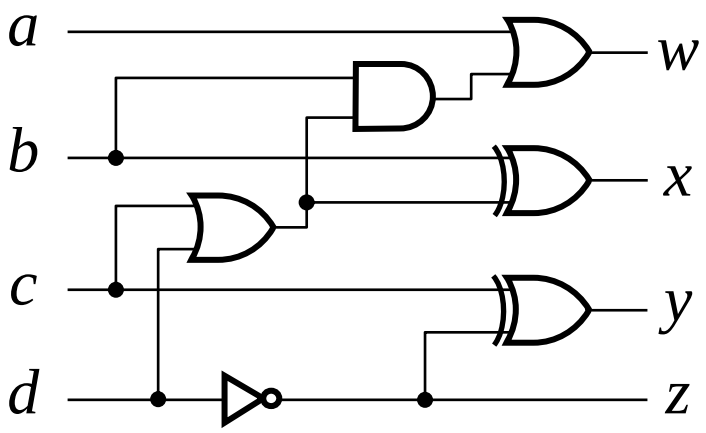
Other gates can be used, such as NAND, NOR, and XOR



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

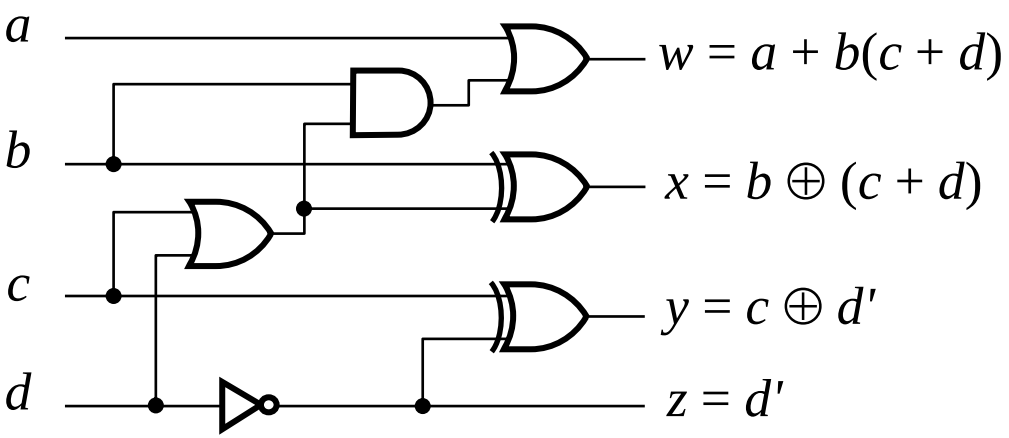
Extract output functions from circuit diagram

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

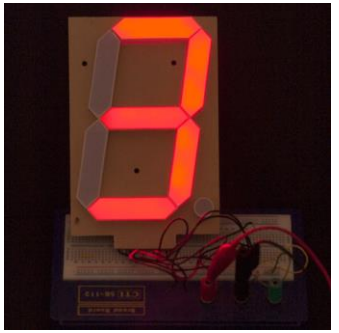


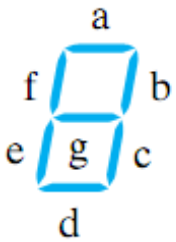
Truth Table of the Circuit Diagram

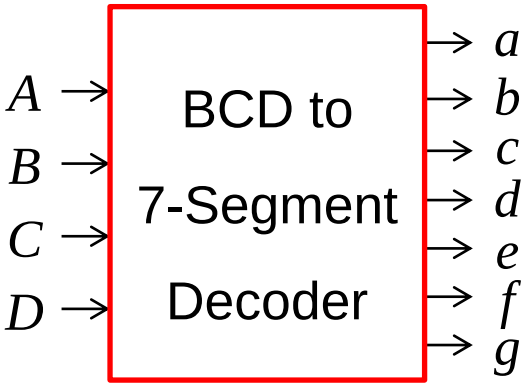
BCD a b c d	c+d	b(c+d)	Excess-3 w x y z
0 0 0 0	0	0	0 0 1 1
0 0 0 1	1	0	0 1 0 0
0 0 1 0	1	0	0 1 0 1
0 0 1 1	1	0	0 1 1 0
0 1 0 0	0	0	0 1 1 1
0 1 0 1	1	1	1 0 0 0
0 1 1 0	1	1	1 0 0 1
0 1 1 1	1	1	1 0 1 0
1 0 0 0	0	0	1 0 1 1
1 0 0 1	1	0	1 1 0 0

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

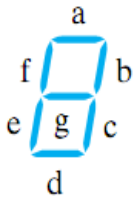
Designing a BCD to 7-Segment Decoder

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



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

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

K-map for b

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

K-map for c

AB \ CD	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

Let $T_1 = A'B$, $T_2 = B'C'$, $T_3 = A'D$

Optimized Logic Expressions

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

$$b = A'B' + T_2 + A'C'D' + T_3C$$

$$c = T_1 + T_2 + T_3$$

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

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

K-map for e

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

K-map for f

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

K-map for g

AB \ CD	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

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

$b = A'B' + T_2 + A'C'D' + T_3C$

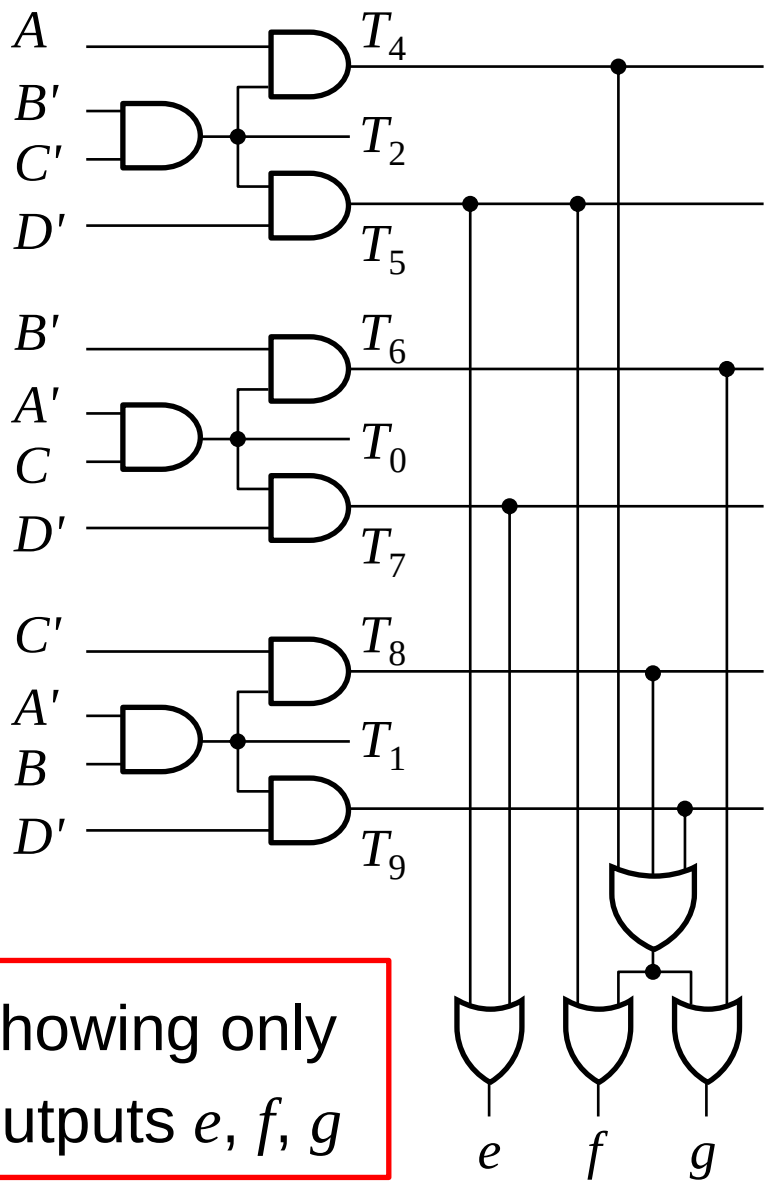
$c = T_1 + T_2 + T_3$

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

$e = T_5 + T_7$

$f = T_4 + T_5 + T_8 + T_9$

$g = T_4 + T_6 + T_8 + T_9$



Showing only Outputs e, f, g

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

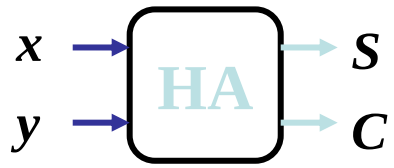
- The blocks are verified and well-document
- They are placed in a library for future use

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

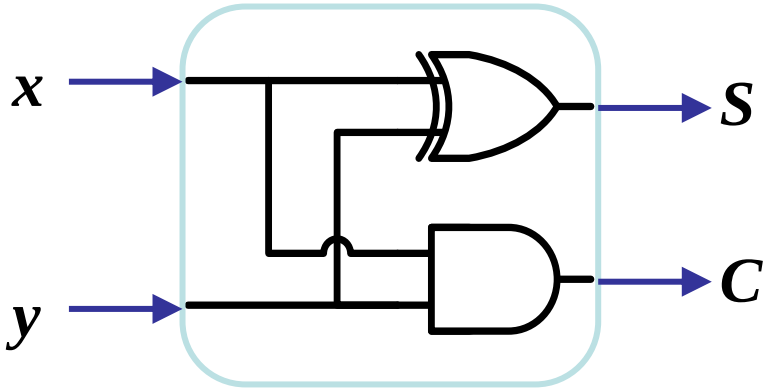
BINARY ADDER-SUBTRACTOR

- Half Adder
  - Adds 1-bit plus 1-bit
  - Produces Sum and Carry
    - S = x'y +xy'
    - C = xy



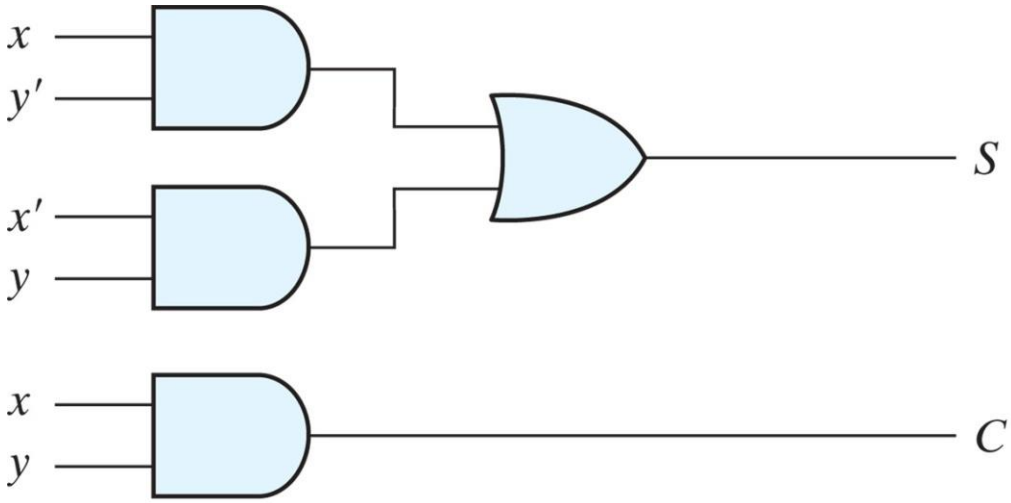
$$\begin{array}{rr} & x \\ + & y \\ \hline C & S \end{array}$$

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

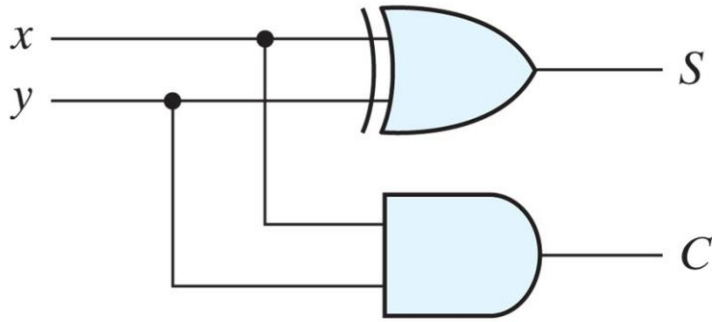


Half Adder

- Implementation of half adder



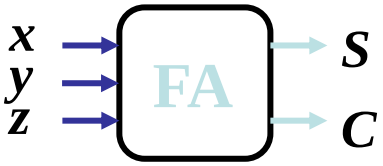
(a) $S = xy' + x'y$
$C = xy$



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

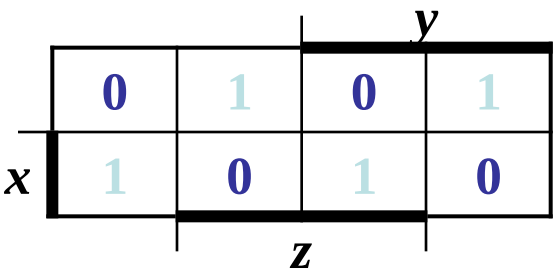
Full Adder

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

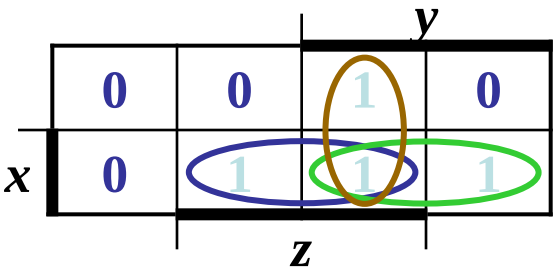


x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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



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

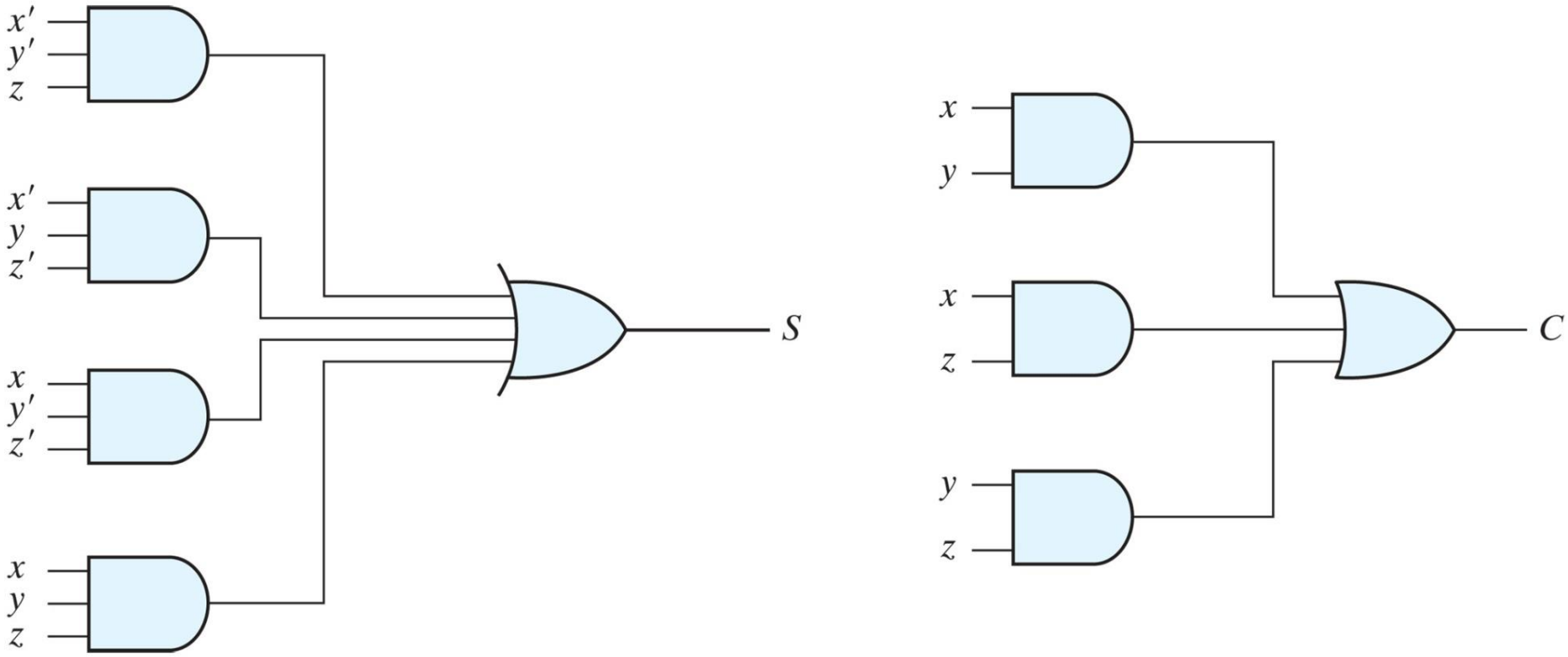


$$C = xy + xz + yz$$

Full Adder

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

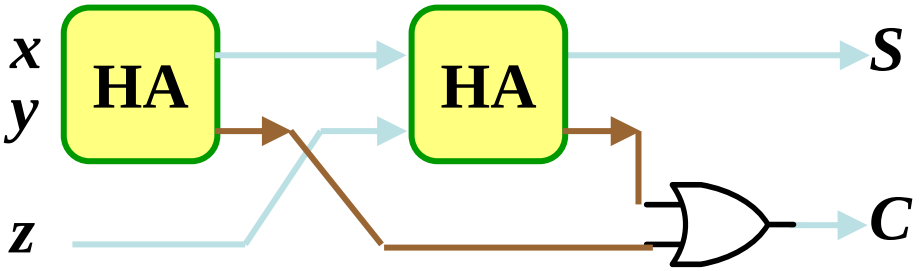
$C = xy + xz + yz$



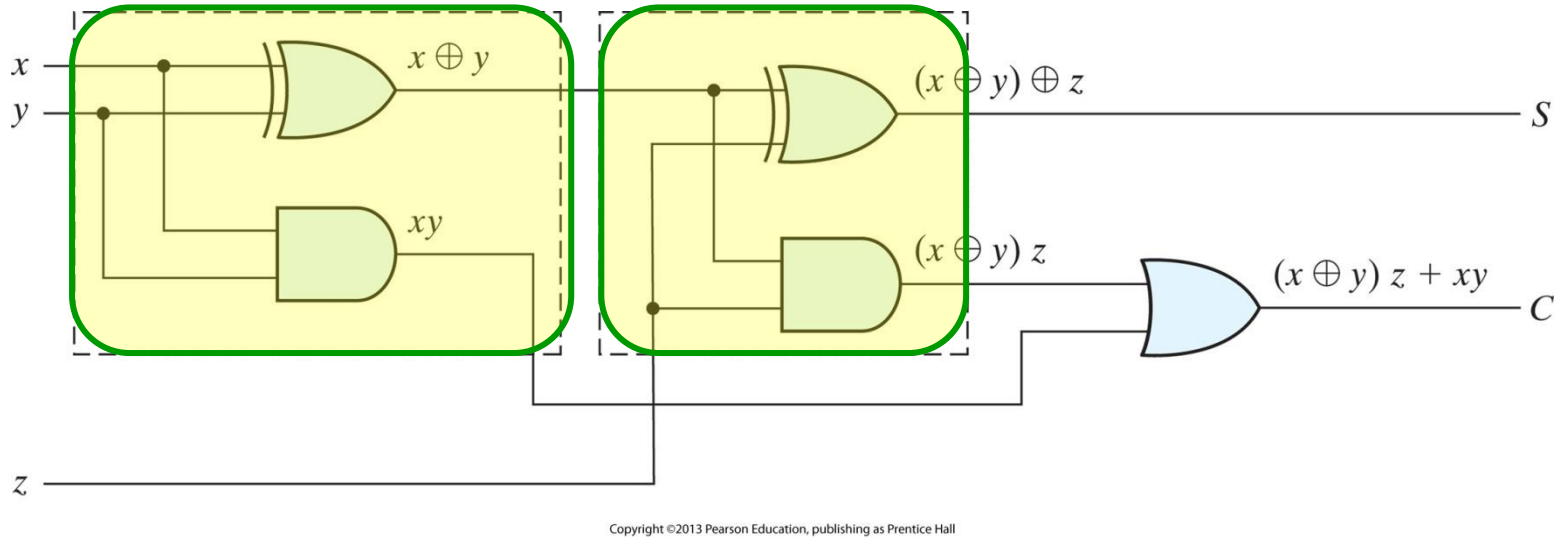
Full adder

- Implementation of full adder with two half adders and an OR gate

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



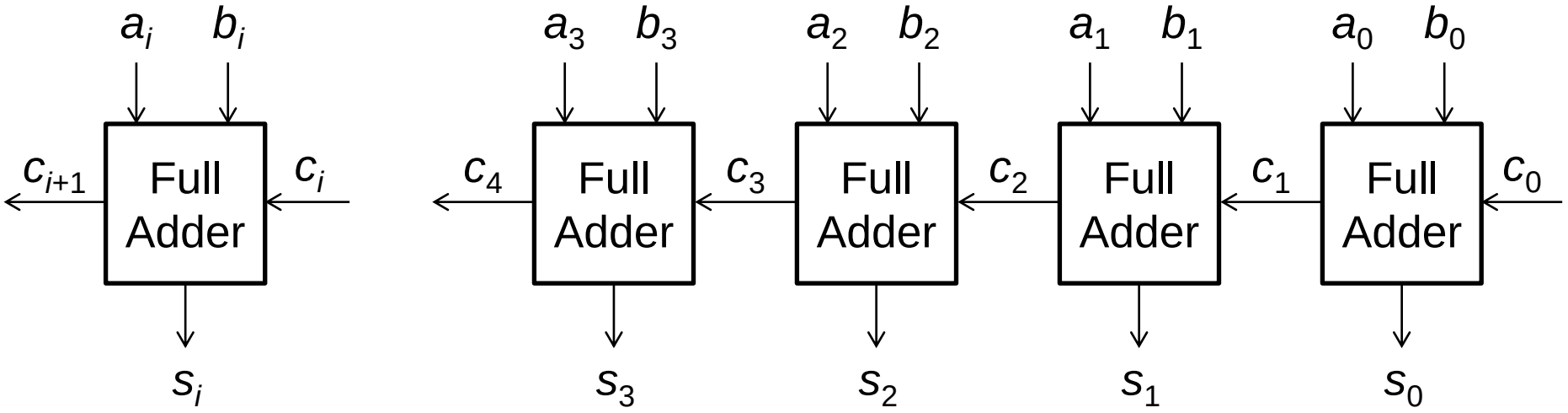
$$C = z(xy' + x'y) + xy = xy'z + x'yz + xy$$



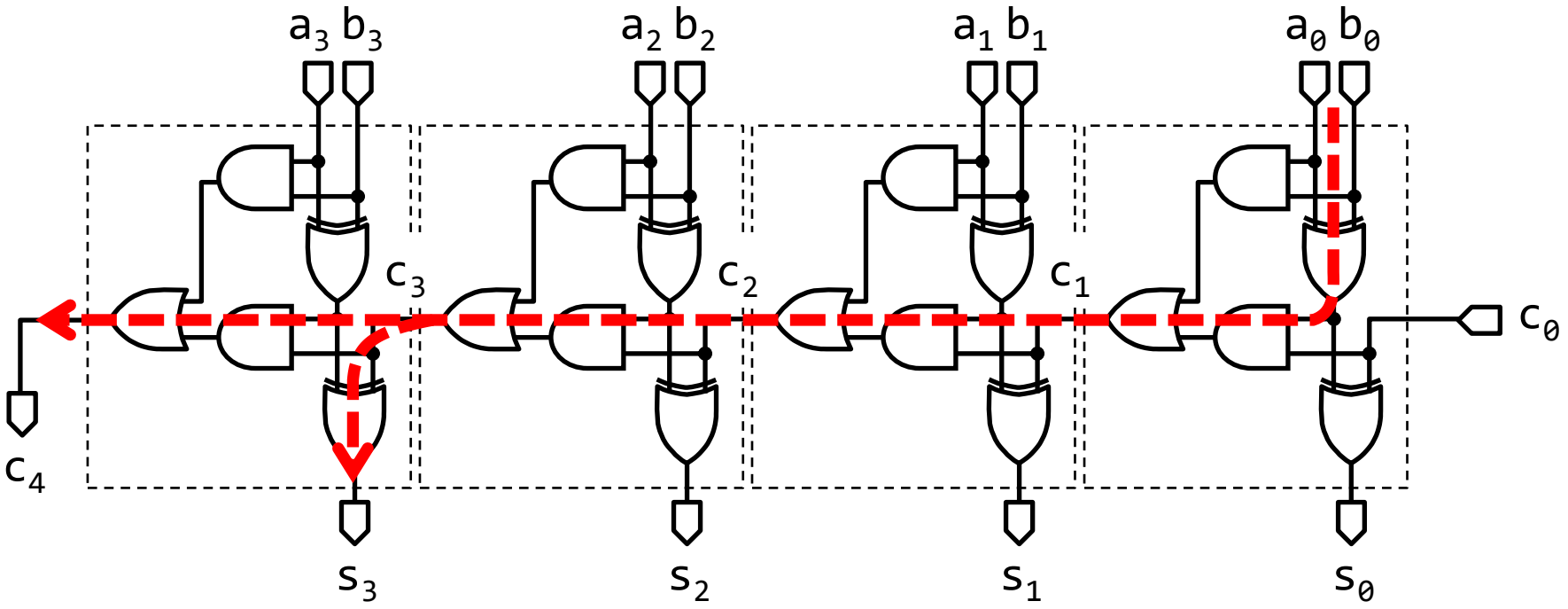
Iterative Design: Ripple Carry Adder

- Uses **identical copies** of a full adder to build a large adder
- Simple to implement: can be extended to add any number of bits
- 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}
- Carry-out of cell i becomes carry-in to cell (i+1)



Carry Propagation



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

Converting Subtraction into Addition

- When computing **A – B**, convert **B** to its 2's complement

  A – B = A + (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**	(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

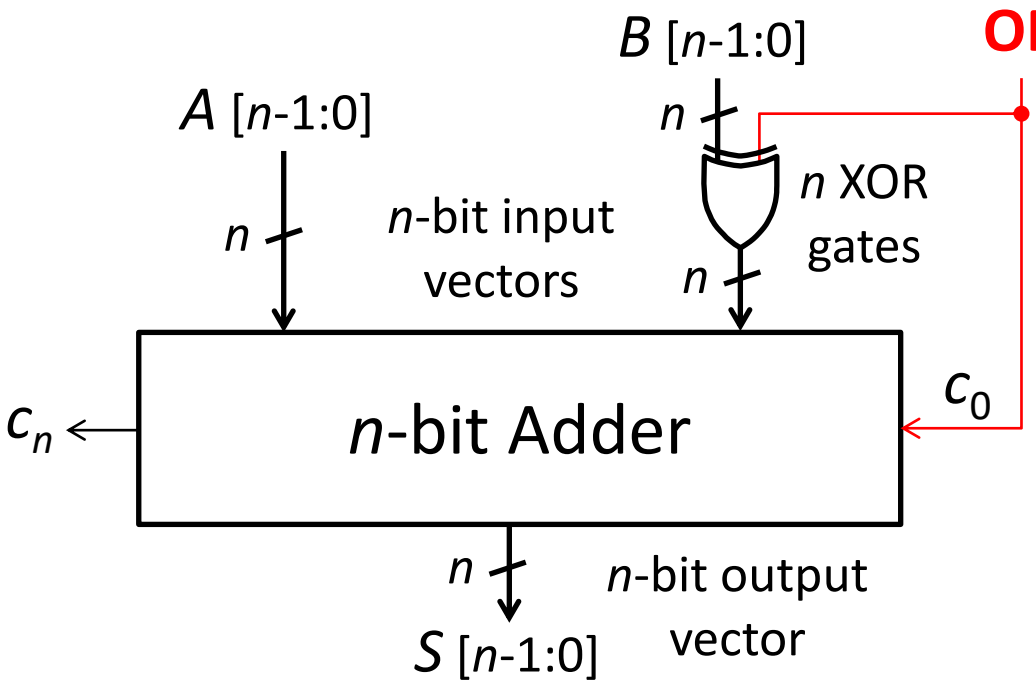
  $A + (2\text{'s complement of } B) = A + (2^n - B) = (A - B) + 2^n$

  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's complement of B)

  2's complement of B = (1's complement of B) + 1
- Two operations: **OP = 0 (ADD)**, **OP = 1 (SUBTRACT)**



OP = 0 (ADD)

B XOR 0 = B

S = A + B + 0 = A + B

OP = 1 (SUBTRACT)

B XOR 1 = 1's complement of B

S = A + (1's complement of B) + 1

S = A + (2'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
  - Sum > maximum unsigned *n*-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
	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)
	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
	1	0	0	0	1	1	1	1	143 (-113)

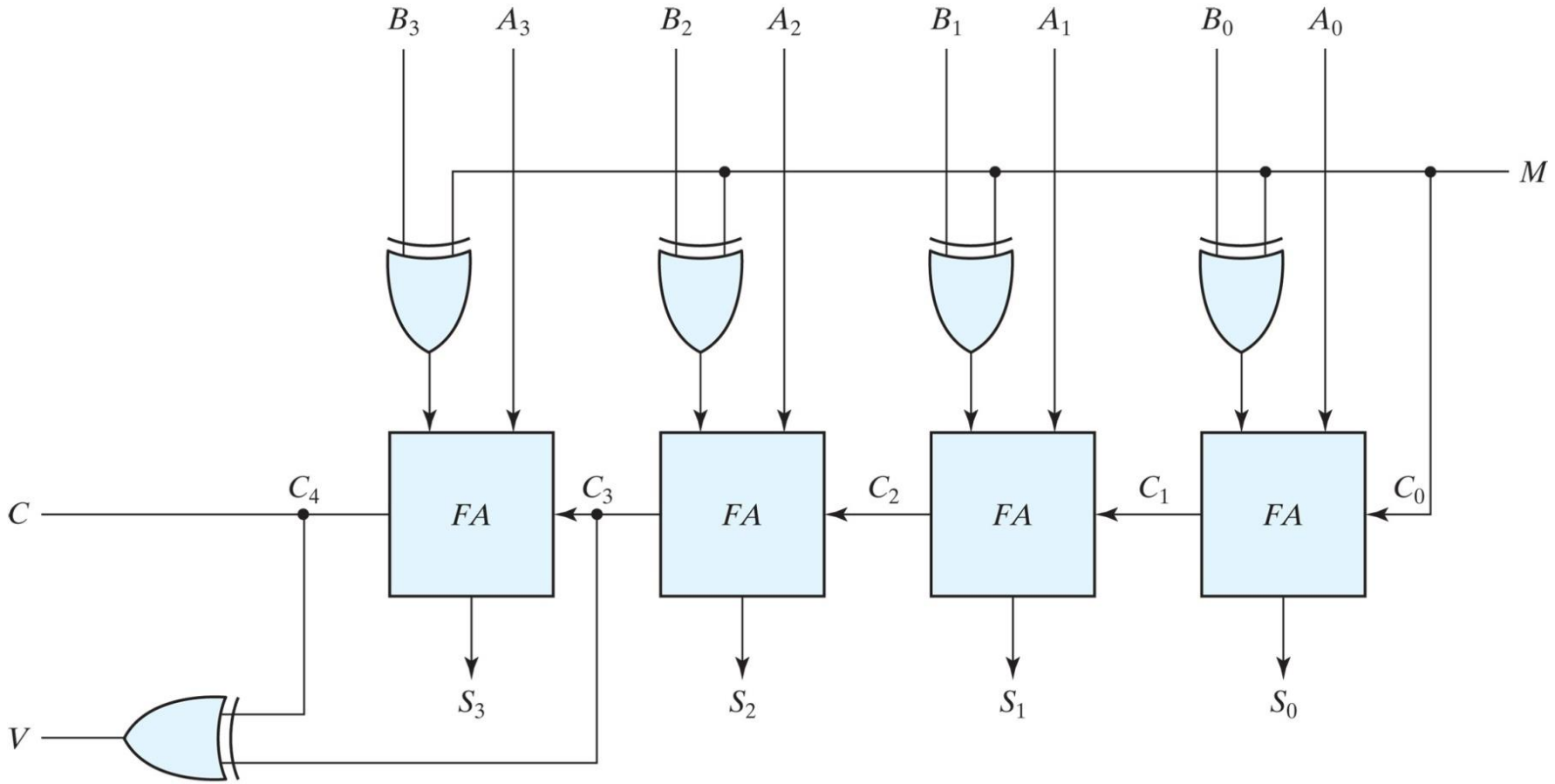
Carry = 0 Overflow = 1

	1				1	1			
	1	1	0	1	1	0	1	0	218 (-38)
+	1	0	0	1	1	1	0	1	157 (-99)
	0	1	1	1	0	1	1	1	119

Carry = 1 Overflow = 1

Four-bit adder-subtractor

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



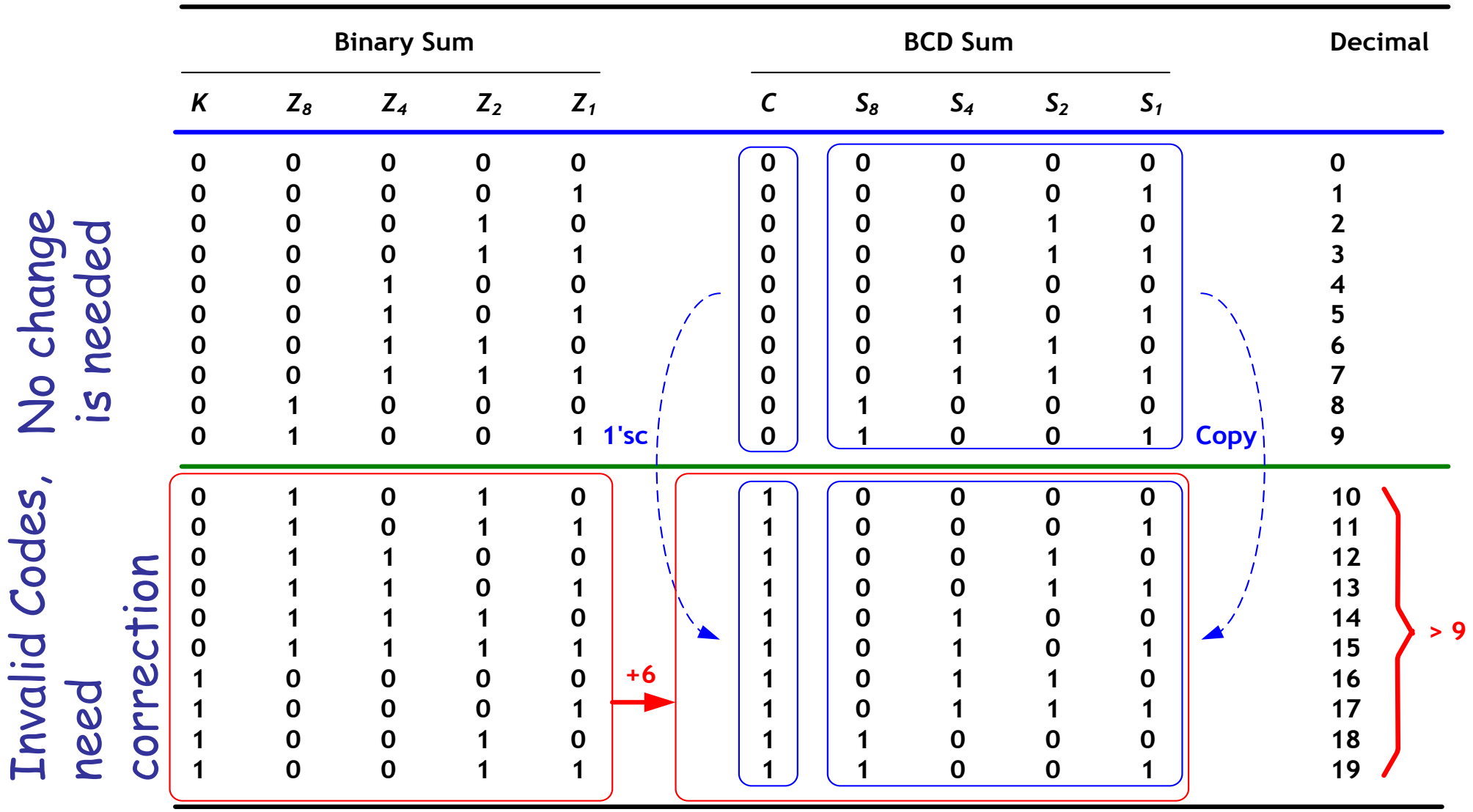
DECIMAL ADDER (BCD Adder)

- Consider adding two decimal digits in BCD
- Operands and Result: 0 to 9
- Output sum cannot exceed 9+9+1=19 (the last 1 is the carry from previous digit)
- 4-bits plus 4-bits

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

BCD	1	1		
	0001	1000	0100	184
	+0101	0111	0110	+576
Binary sum	0111	10000	1010	
Add 6		0110	0110	
BCD sum	0111	0110	0000	760

Derivation of BCD Adder

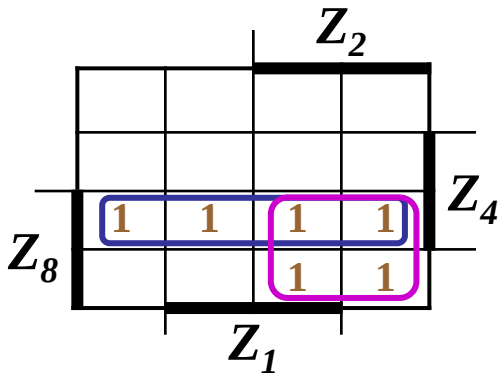


$$c = k + z_8 z_4 + z_8 z_2$$ Condition for correcting result

BCD Adder

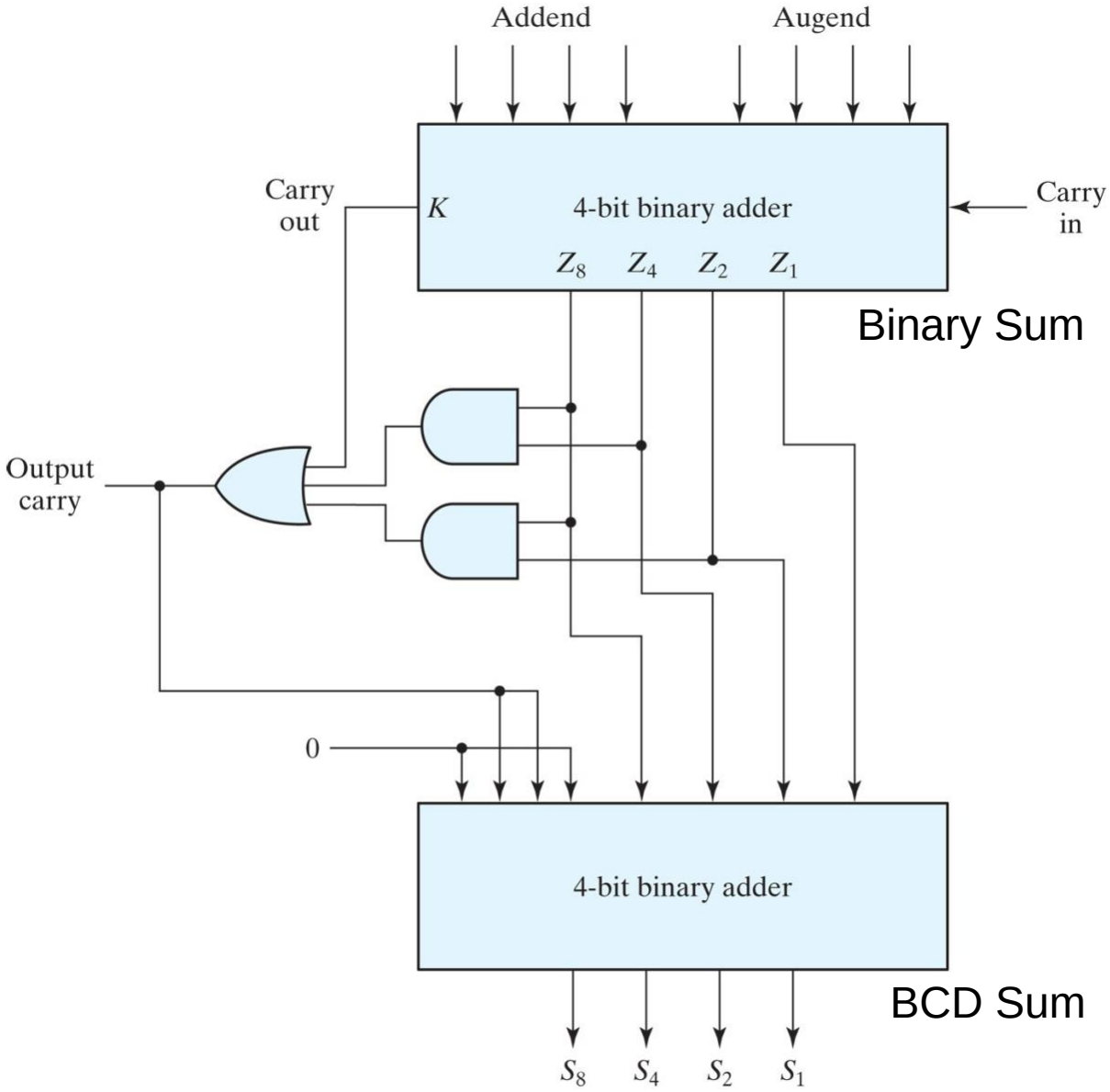
- Correct Binary Adder's Output (+6)
  - If the result is between 'A' and 'F'
  - If K= 1

$Z_8\ Z_4\ Z_2\ Z_1$	Err
0 0 0 0	0
1 0 0 0	0
1 0 0 1	0
1 0 1 0	1
1 0 1 1	1
1 1 0 0	1
1 1 0 1	1
1 1 1 0	1
1 1 1 1	1

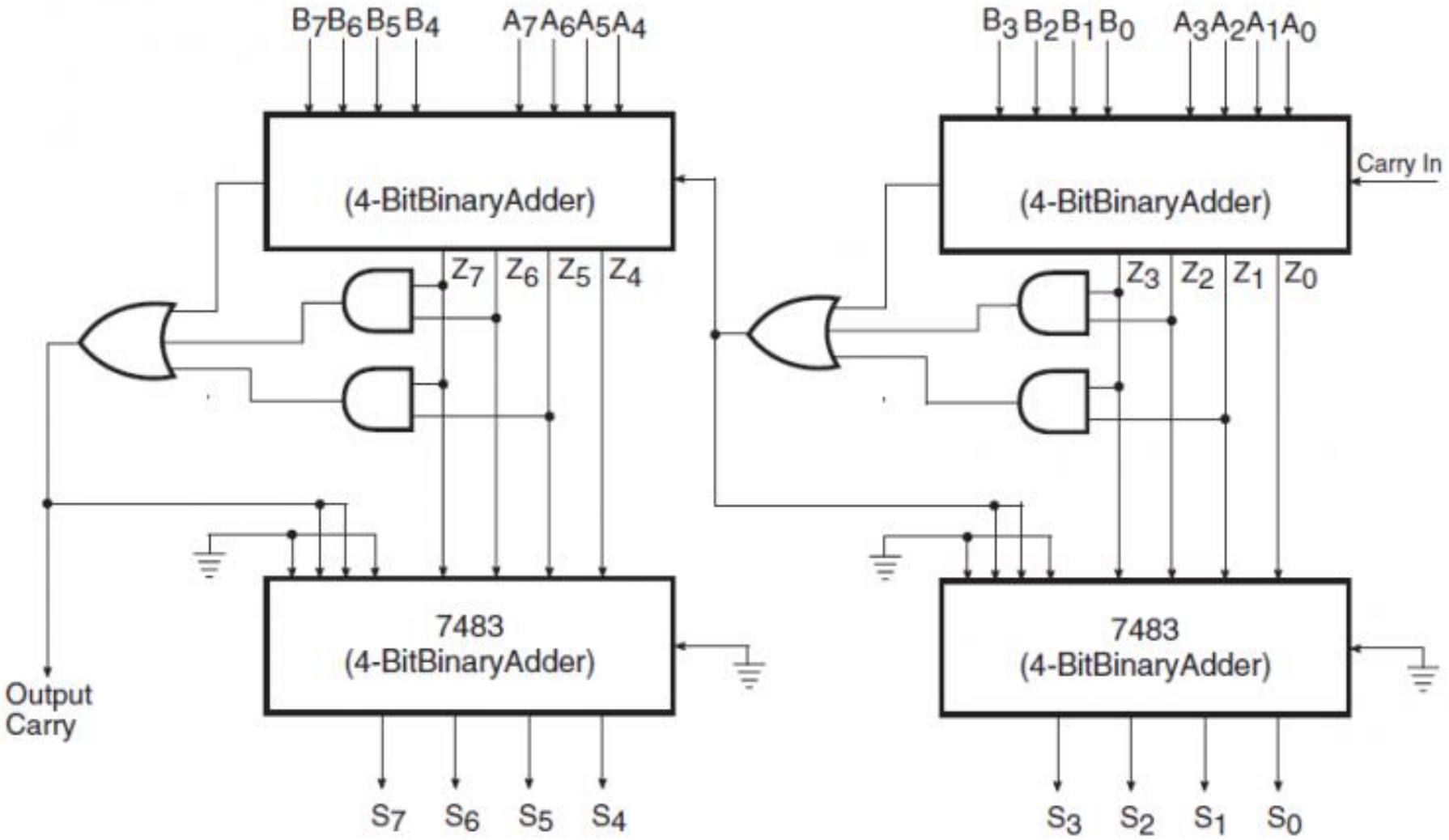


$$Err = Z_8 Z_4 + Z_8 Z_2$$

Block diagram of a 1-Digit BCD adder

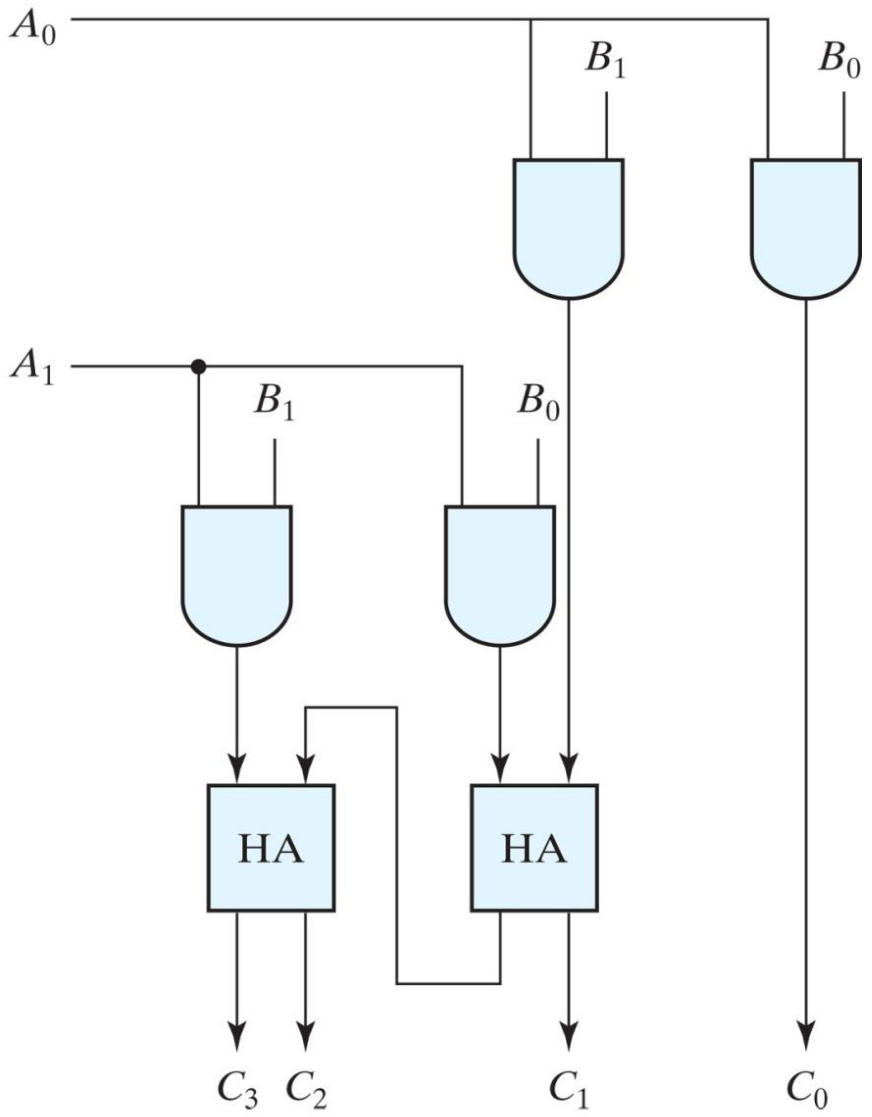


Block diagram of a 2-Digit BCD adder

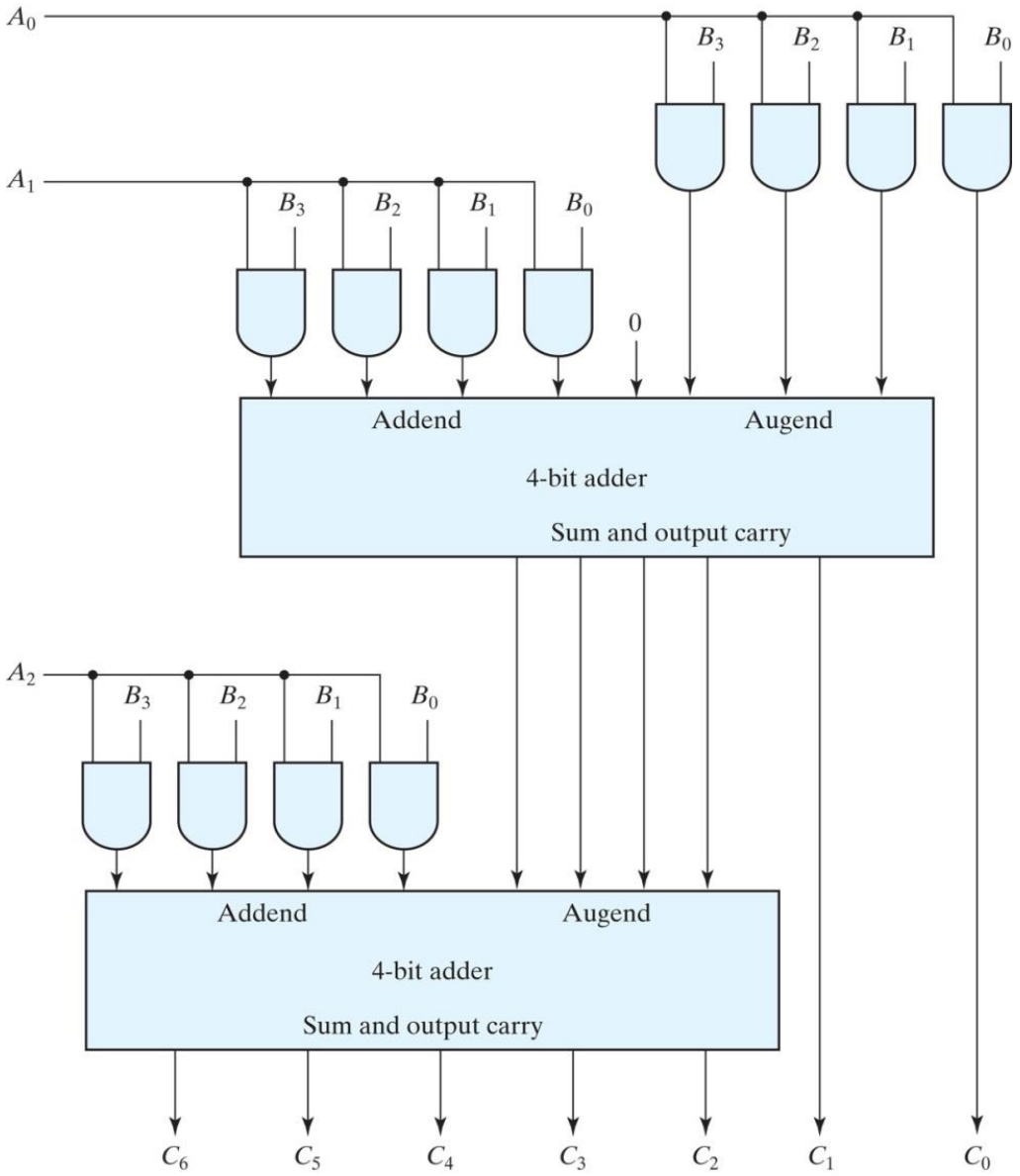


Two-bit by two-bit binary multiplier

$$
\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\hline
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
C_3 & C_2 & C_1 & C_0
\end{array}
$$

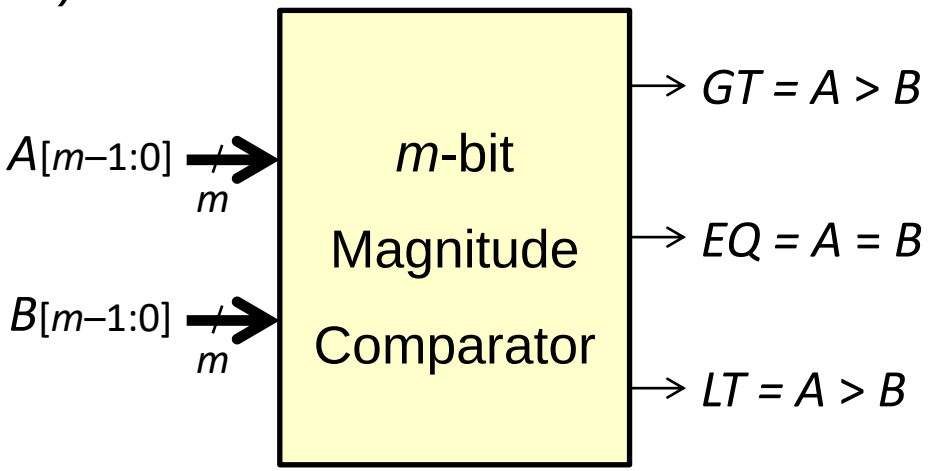


Four-bit by three-bit binary multiplier



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)
  - $EQ = 1 \leftrightarrow A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$, and $A_0 = B_0$
  - Define: $E_i = A_iB_i + A_i'B_i'$
  - Therefore, $EQ = E_3E_2E_1E_0$

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 $B_2 = 1)$

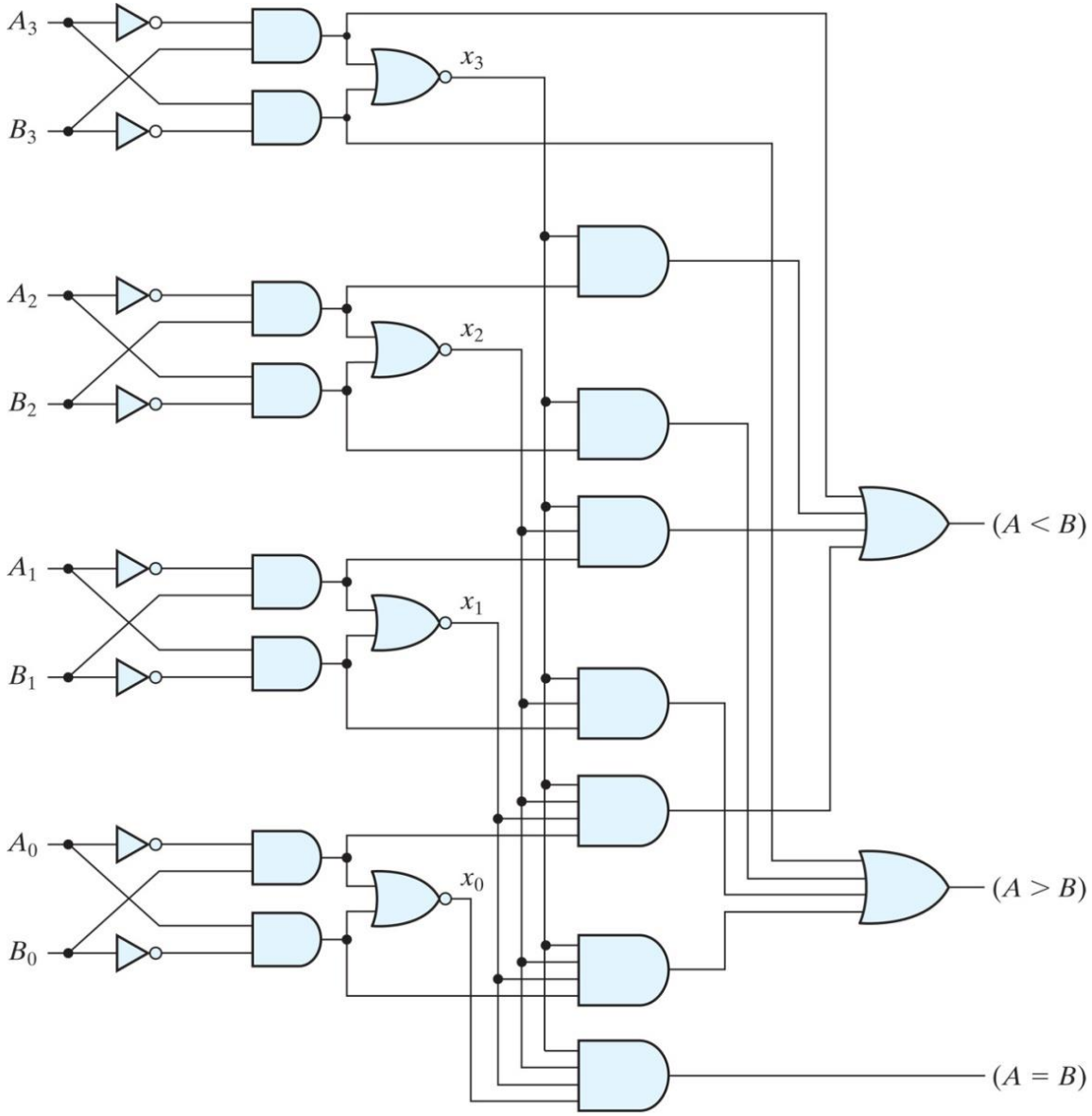
3. Define: $L_1 = A_1' B_1$ $(A_1 = 0$ and $B_1 = 1)$

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 $LT = (GT + EQ)'$

Magnitude Comparator



Iterative Magnitude Comparator Design

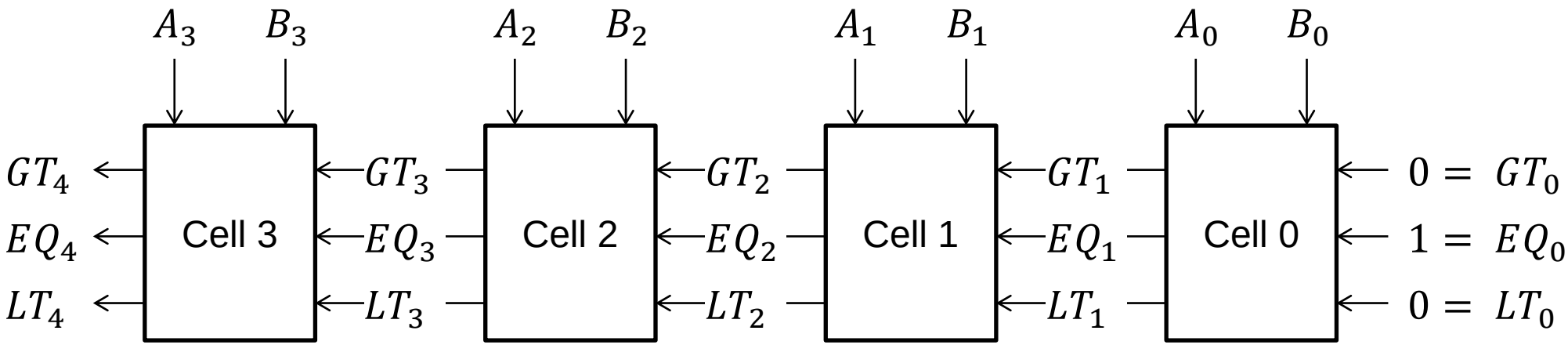
- The Magnitude comparator can also be designed iteratively

  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: $GT = GT_4\ ,\ EQ = EQ_4\ ,\ LT = LT_4$



Cell Implementation

- Each Cell i receives as inputs:

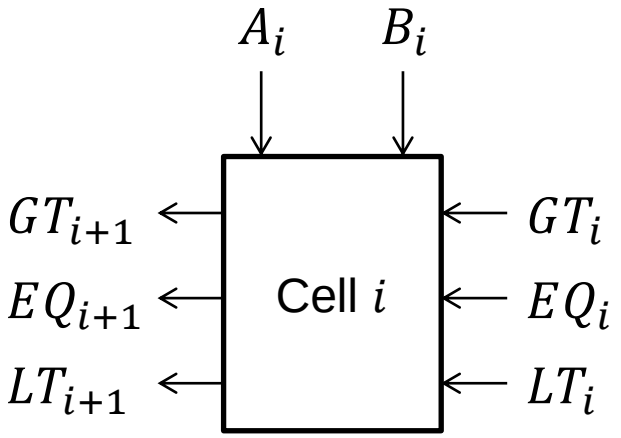
  Bit i of inputs A and B: A_i and B_i

  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

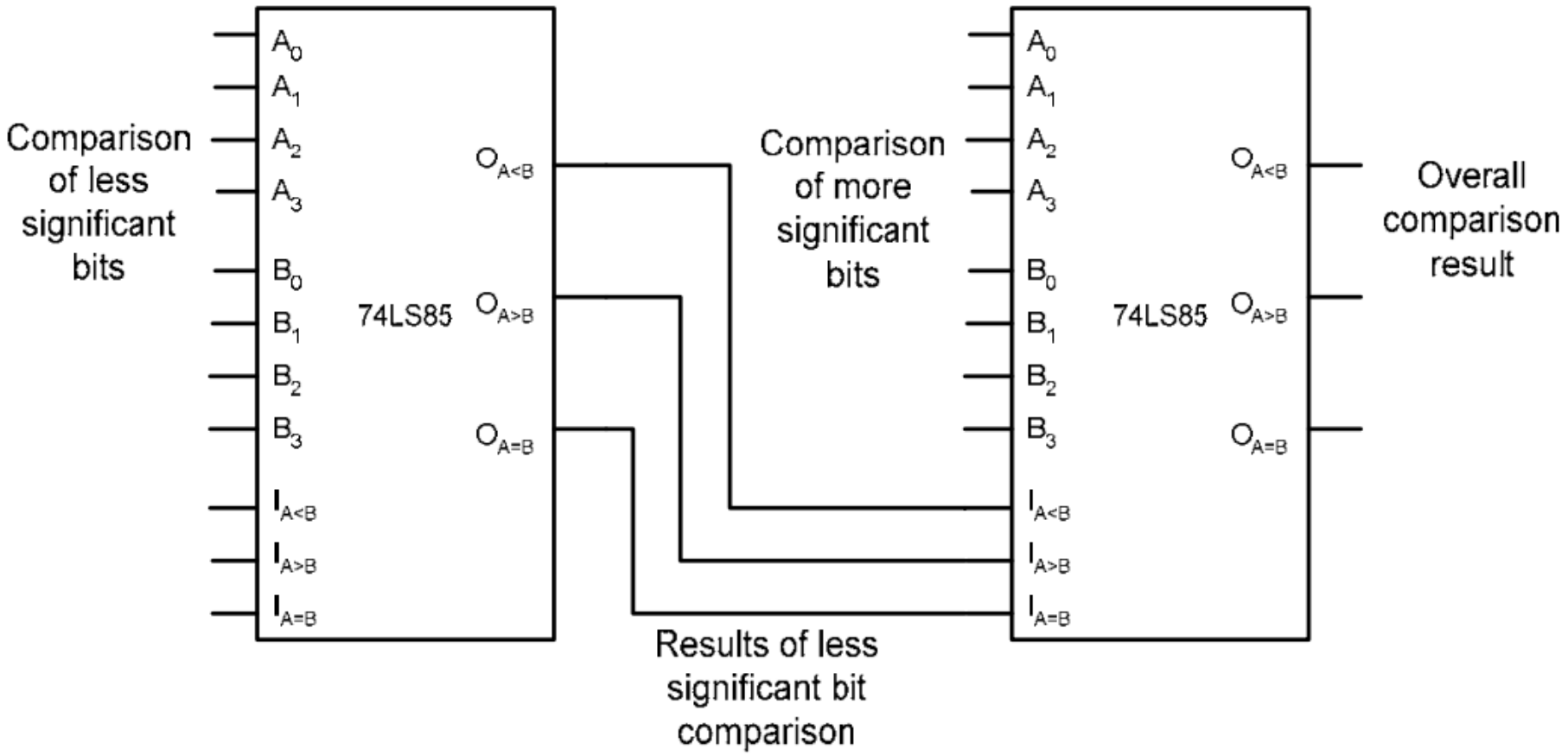
$$EQ_{i+1} = E_i\, EQ_i \qquad E_i = A_i'B_i' + A_iB_i \ (A_i \text{ equals } B_i)$$

$$GT_{i+1} = A_iB_i' + E_i\, GT_i \qquad A_iB_i' \ (A_i > B_i)$$

$$LT_{i+1} = A_i'B_i + E_i\, LT_i \qquad A_i'B_i \ (A_i < B_i)$$

Third output can be produced for first two: $LT = (EQ + GT)'$

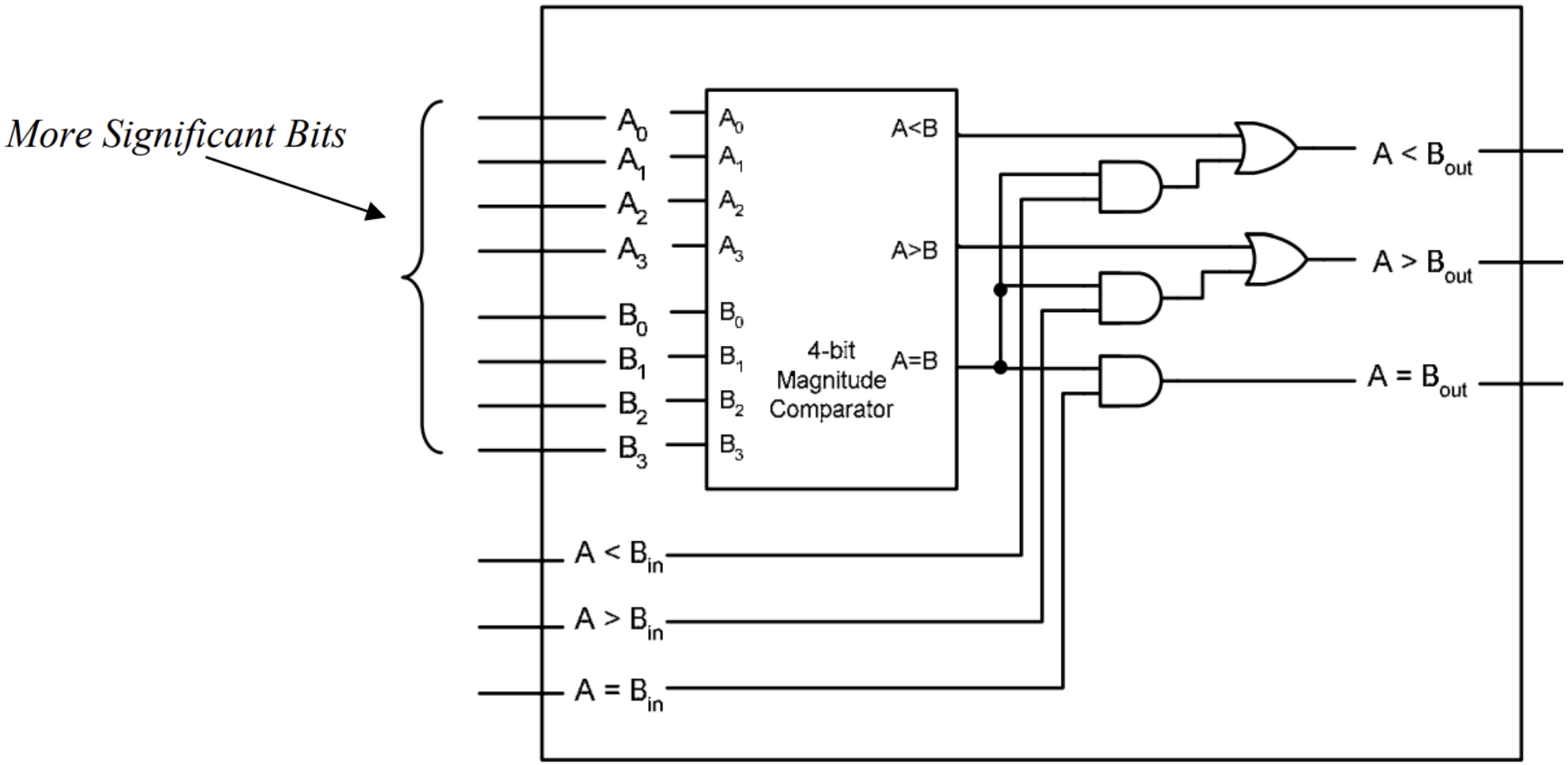
Cascading two Comparators



Cascading two Comparators

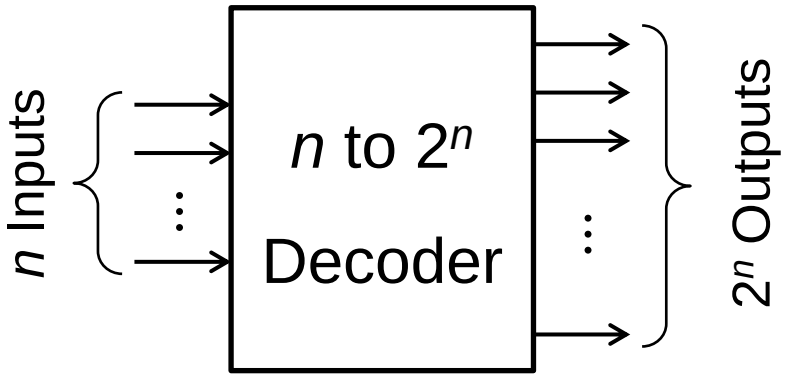
Comparing Inputs				Cascaded Inputs			Outputs		
A3,B3	A2,B2	A1,B1	A0,B0	Ain>Bin	Ain<Bin	Ain=Bin	A>B	A<B	A=B
A3>B3	x	x	x	x	x	x	1	0	0
A3<B3	x	x	x	x	x	x	0	1	0
A3=B3	A2>B2	x	x	x	x	x	1	0	0
A3=B3	A2<B2	x	x	x	x	x	0	1	0
A3=B3	A2=B2	A1>B1	x	x	x	x	1	0	0
A3=B3	A2=B2	A1<B1	x	x	x	x	0	1	0
A3=B3	A2=B2	A1=B1	A0>B0	x	x	x	1	0	0
A3=B3	A2=B2	A1=B1	A0<B0	x	x	x	0	1	0
A3=B3	A2=B2	A1=B1	A0=B0	1	0	0	1	0	0
A3=B3	A2=B2	A1=B1	A0=B0	0	1	0	0	1	0
A3=B3	A2=B2	A1=B1	A0=B0	0	0	1	0	0	1

Cascading two Comparators



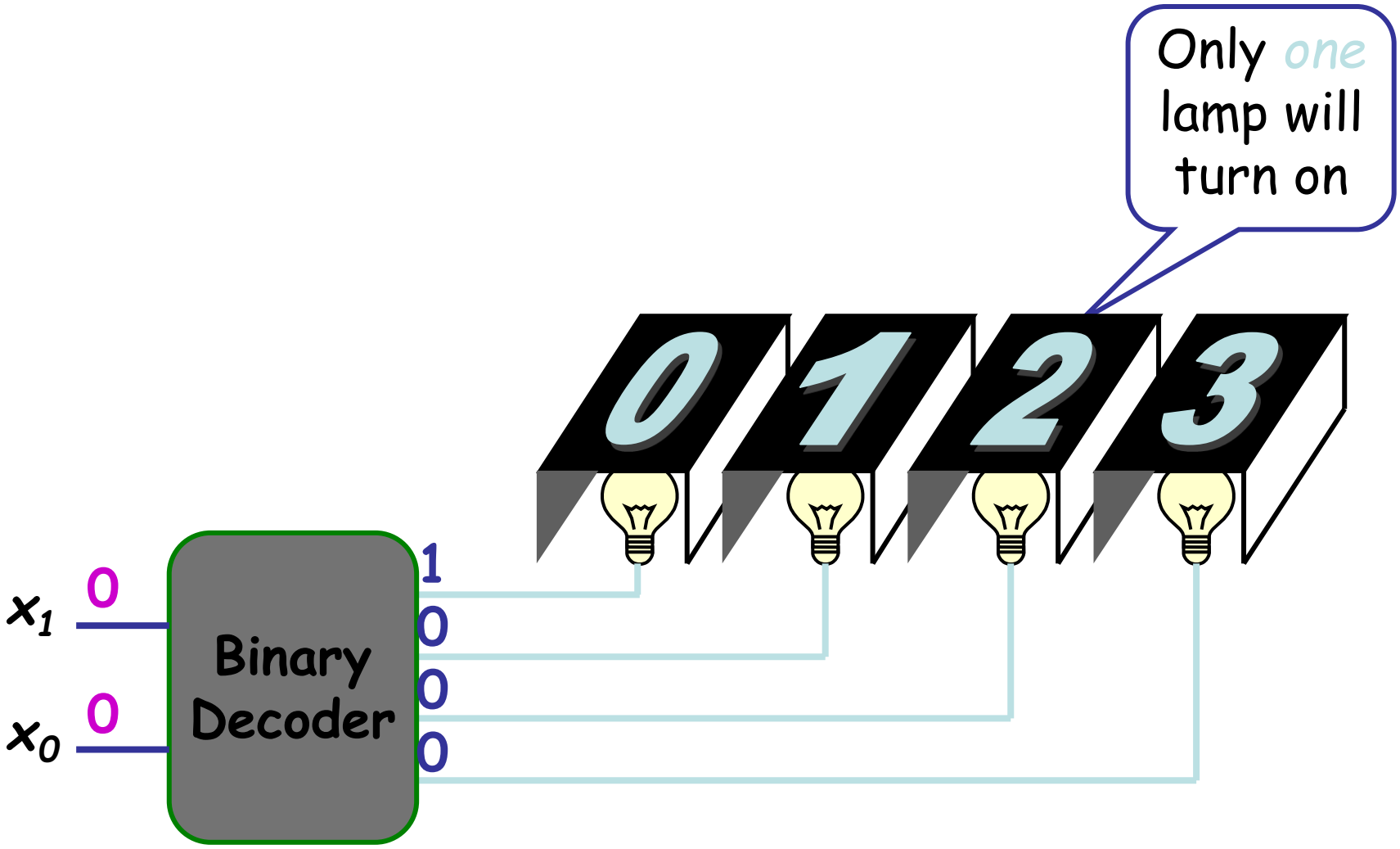
Binary Decoders

- 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**

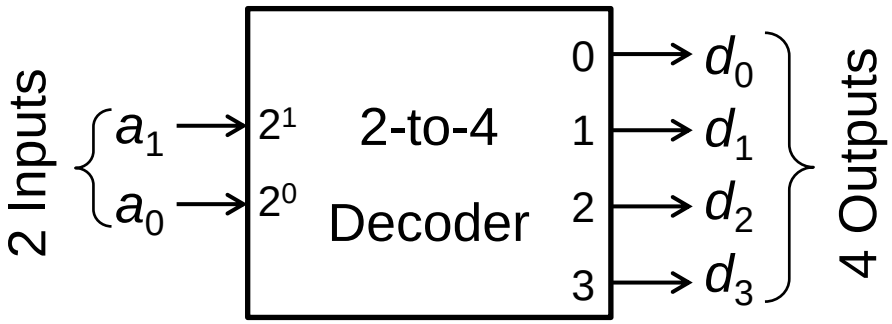


A decoder can have less than 2^n outputs if some input codes are unused

Binary Decoders

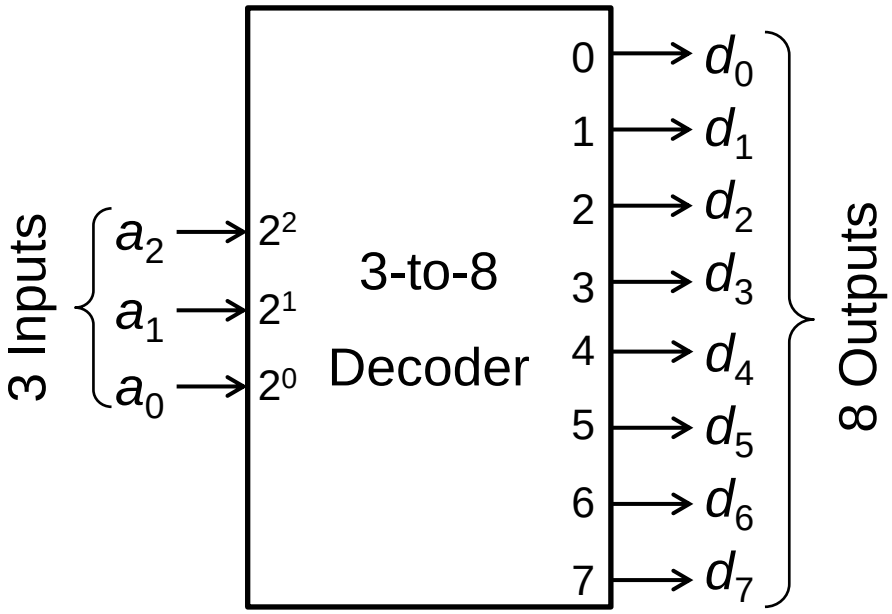


Examples of Binary Decoders



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

Truth Tables

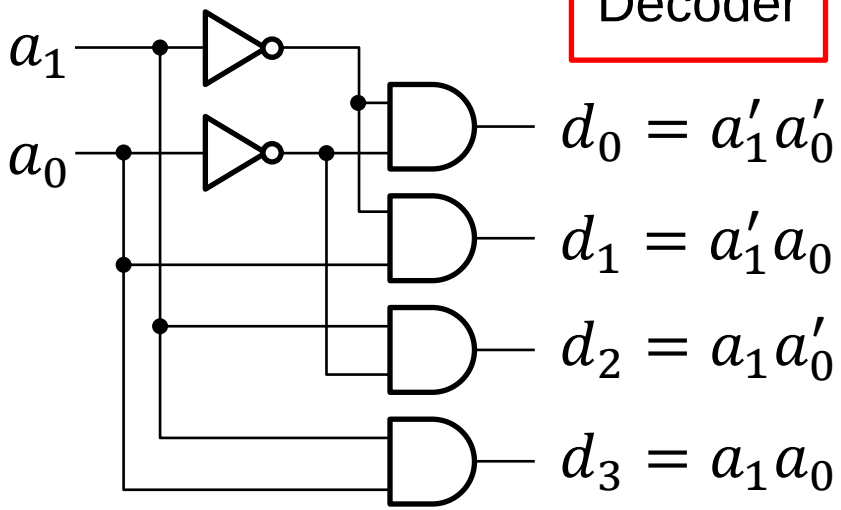


Inputs	Outputs							
a_2 a_1 a_0	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
0 0 0	1	0	0	0	0	0	0	0
0 0 1	0	1	0	0	0	0	0	0
0 1 0	0	0	1	0	0	0	0	0
0 1 1	0	0	0	1	0	0	0	0
1 0 0	0	0	0	0	1	0	0	0
1 0 1	0	0	0	0	0	1	0	0
1 1 0	0	0	0	0	0	0	1	0
1 1 1	0	0	0	0	0	0	0	1

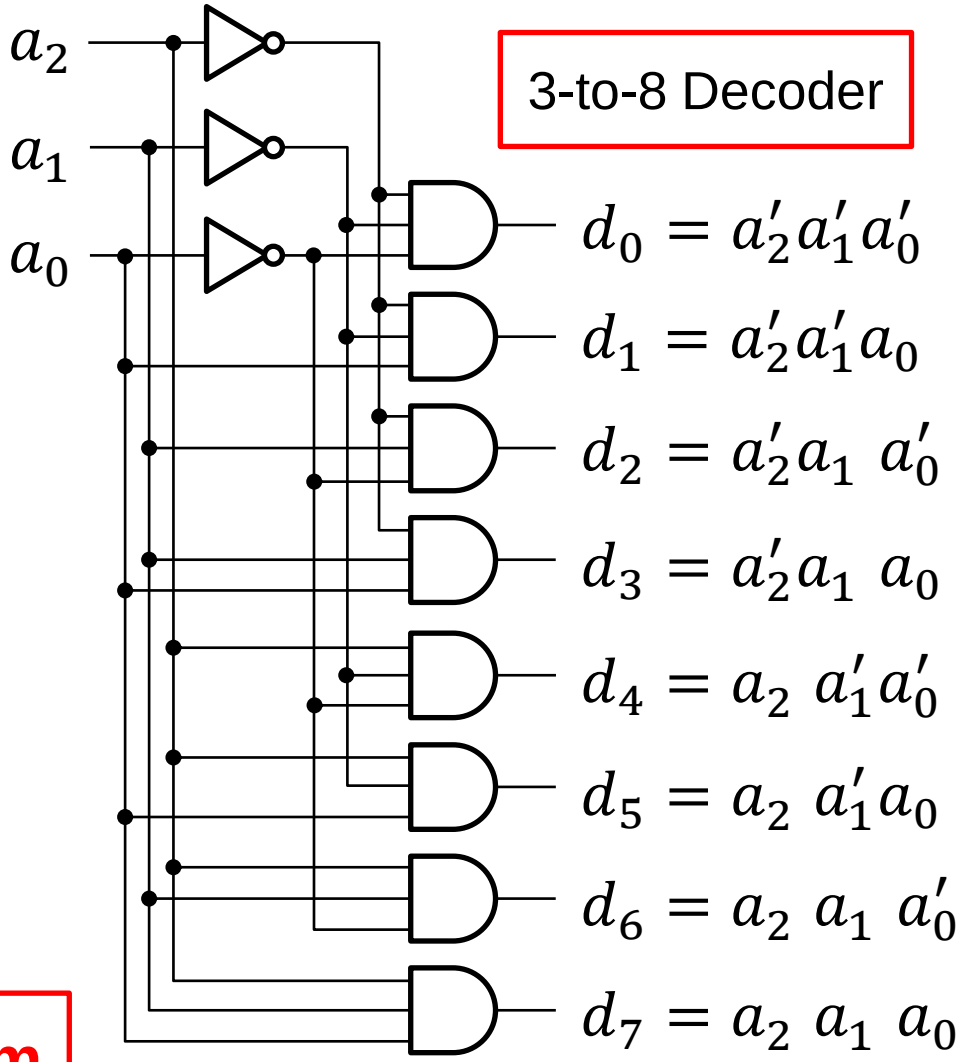
Decoder Implementation

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



3-to-8 Decoder



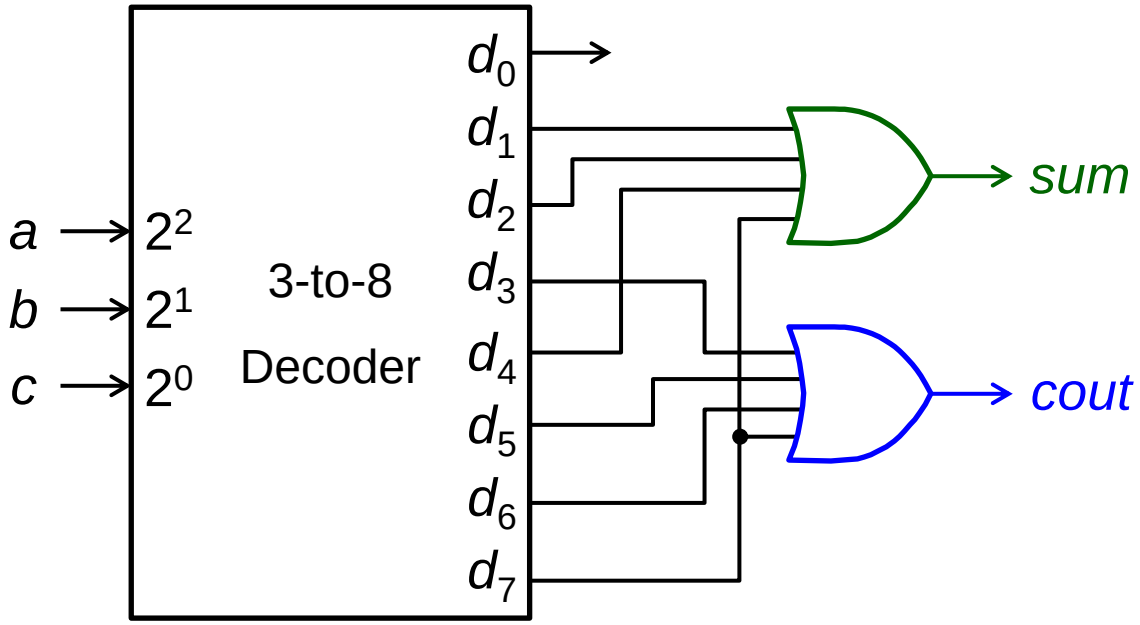
Each decoder output is a **minterm**

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 sum = $\sum(1, 2, 4, 7)$, cout = $\sum(3, 5, 6, 7)$

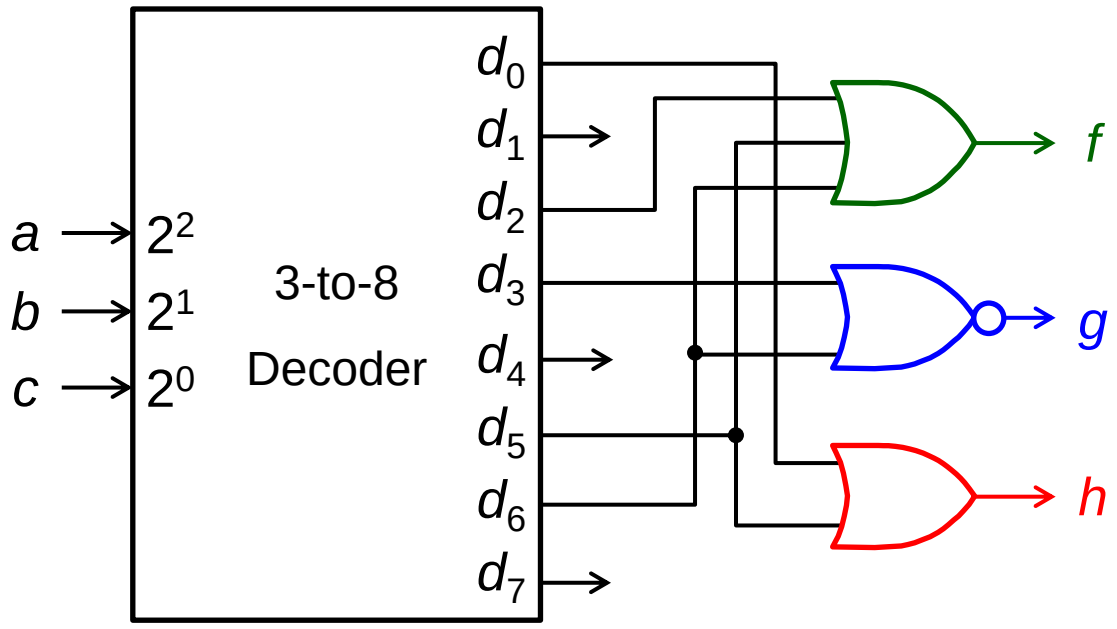
Inputs			Outputs	
a	b	c	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



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:** $f = \Sigma(2, 5, 6)$, $g = \Pi(3, 6)$ ➔ $g' = \Sigma(3, 6)$, $h = \Sigma(0, 5)$

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

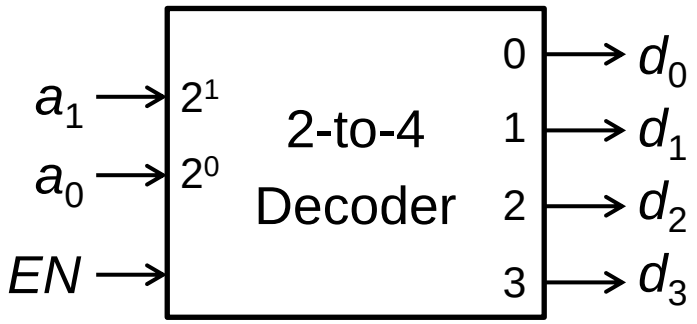


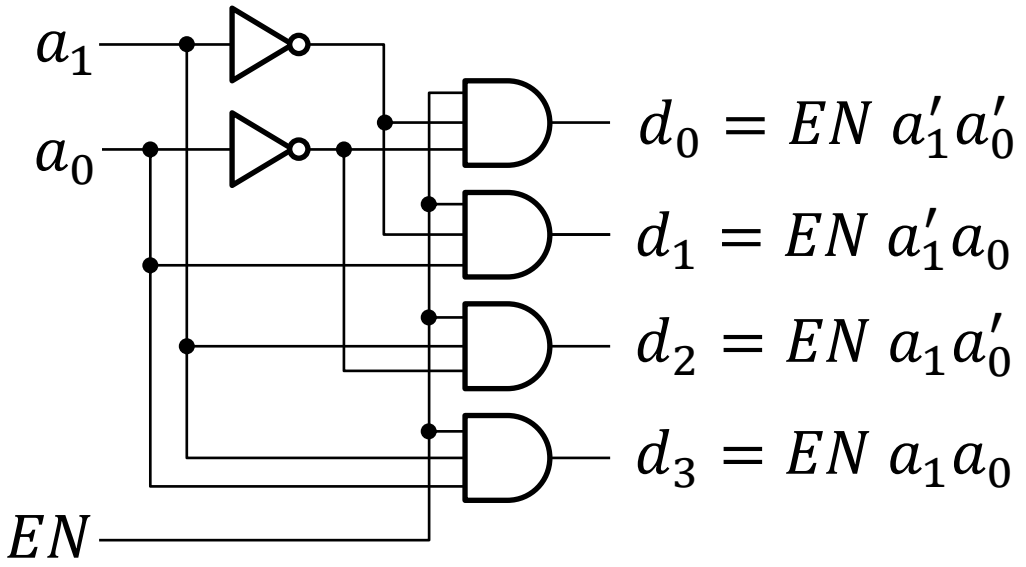
2-to-4 Decoder with Enable Input

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



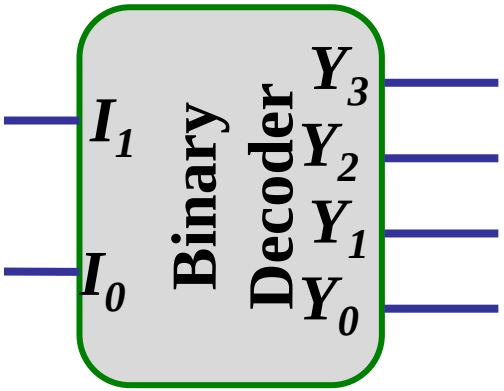


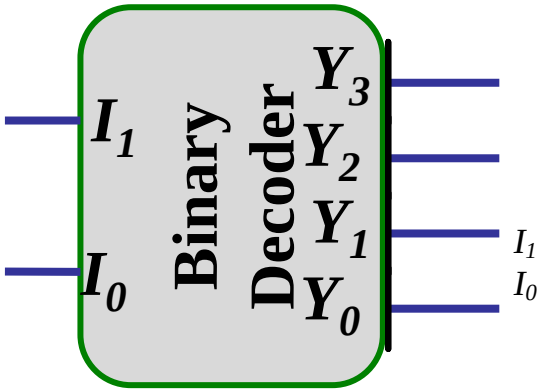
Decoders

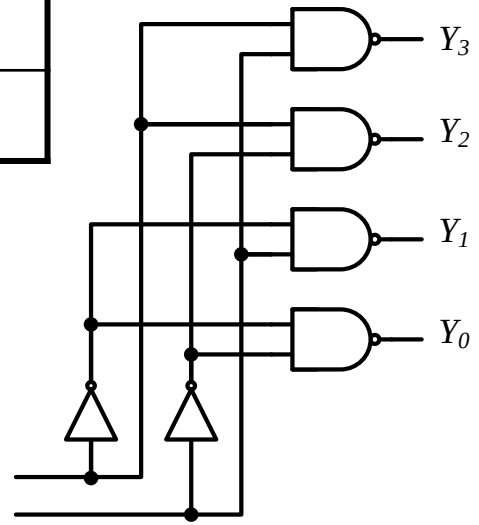
- Active-High / Active-Low

I_1	I_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

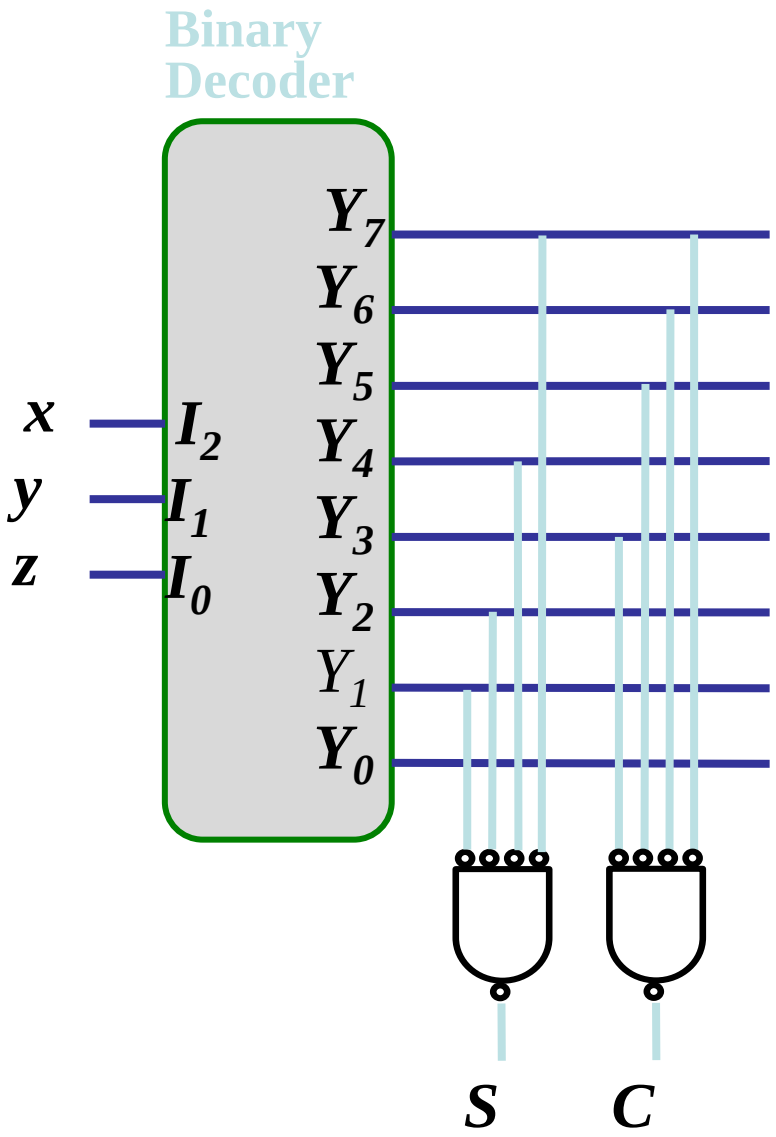
I_1	I_0	Y_3	Y_2	Y_1	Y_0
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	1

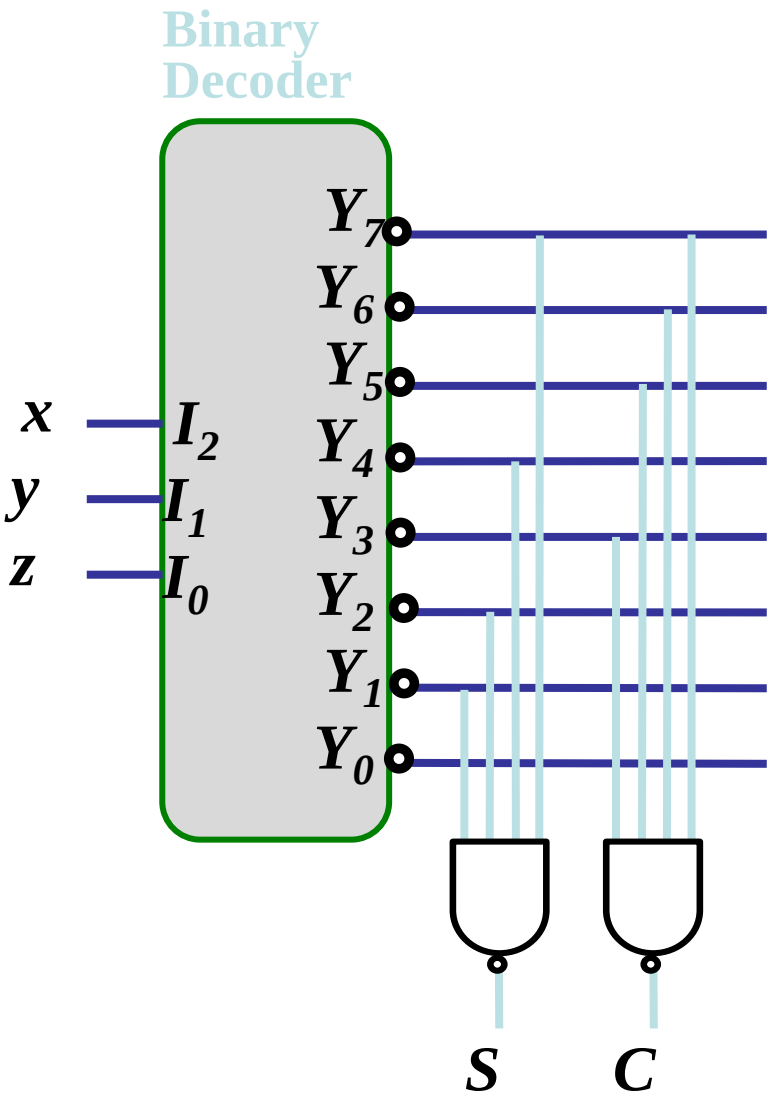






Implementation Using Decoders



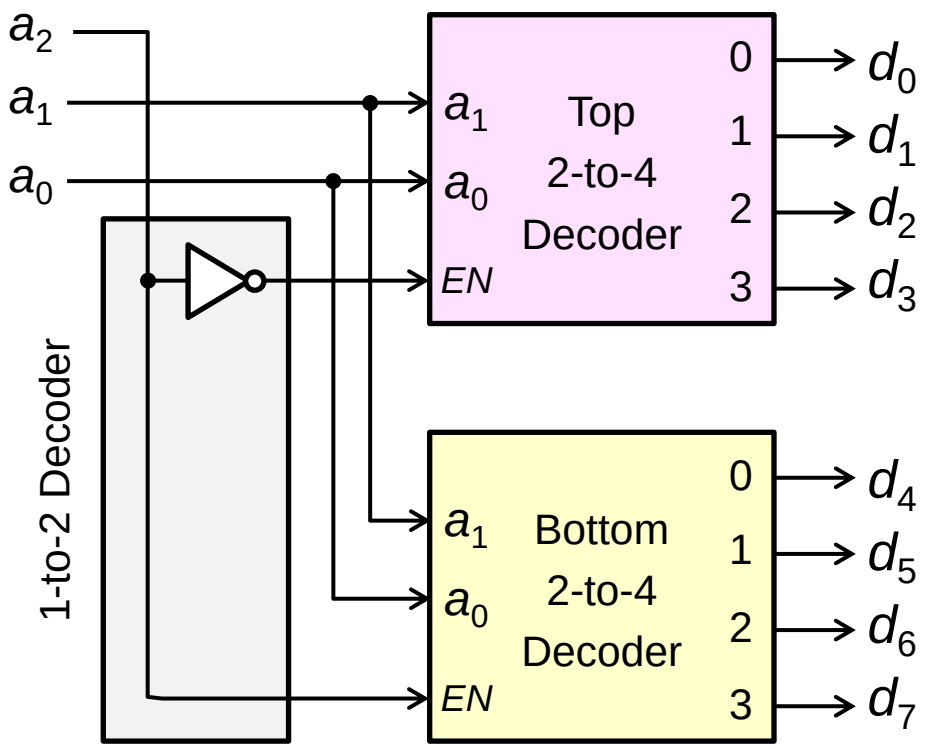


Building Larger Decoders

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

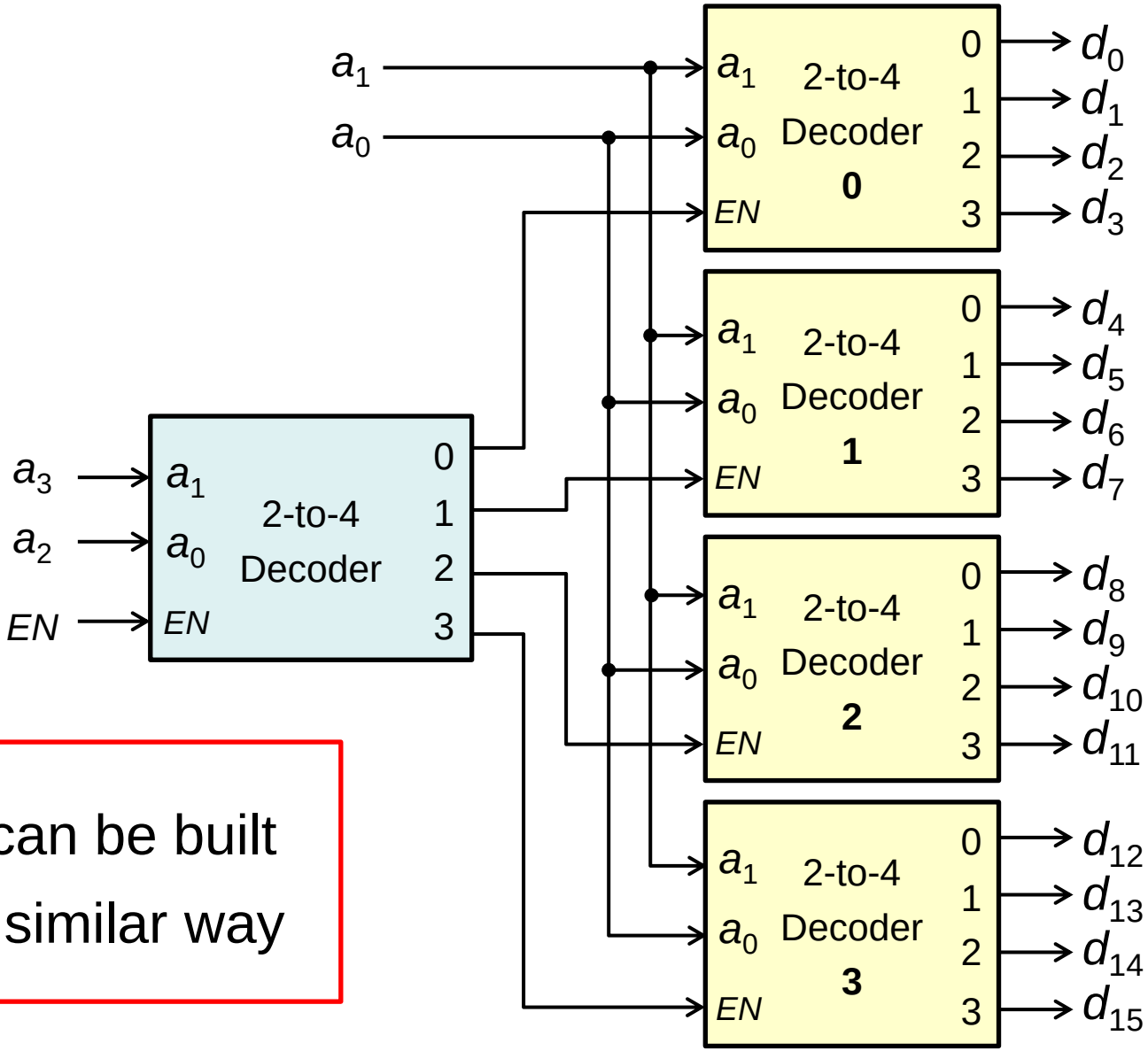
  Two 2-to-4 decoders with Enable and an inverter (1-to-2 decoder)

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

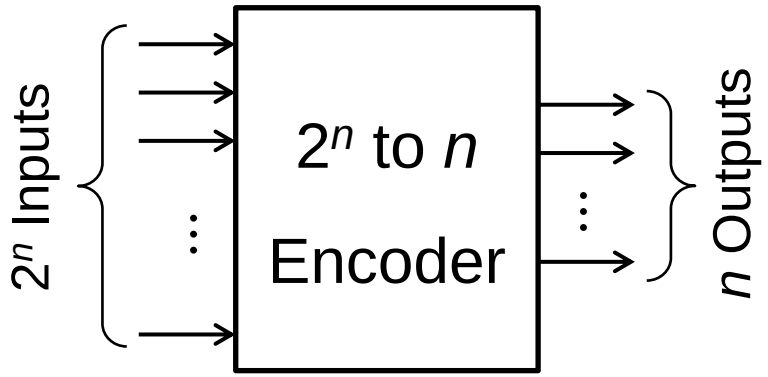


Larger decoders can be built hierarchically in a similar way

Encoders

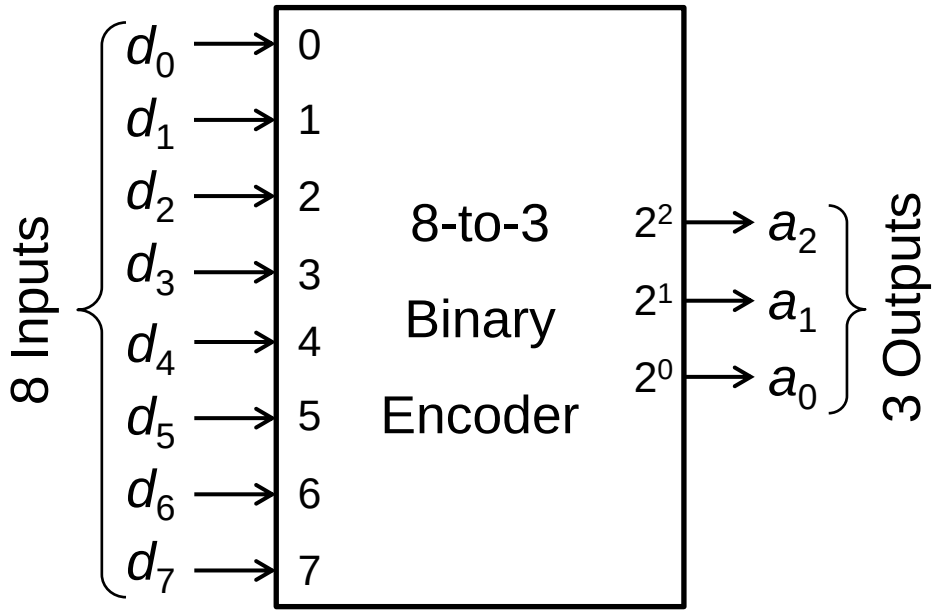
- An encoder performs the opposite operation of a decoder
- It converts a 2^n input to an n-bit output code
- The output indicates which input is active (logic **1**)
- Typically, **one** input should be **1** and all others must be **0**'s
- The conversion of input to output is called **encoding**

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



Example of an 8-to-3 Binary Encoder

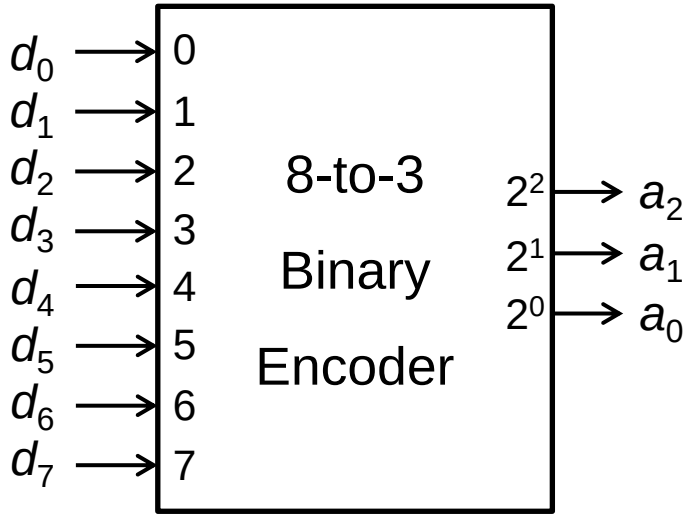
- 8 inputs, 3 outputs, only **one input** is **1**, all others are **0**'s
- Encoder generates the output binary code for the active input
- Output is **not specified** if more than one input is **1**



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

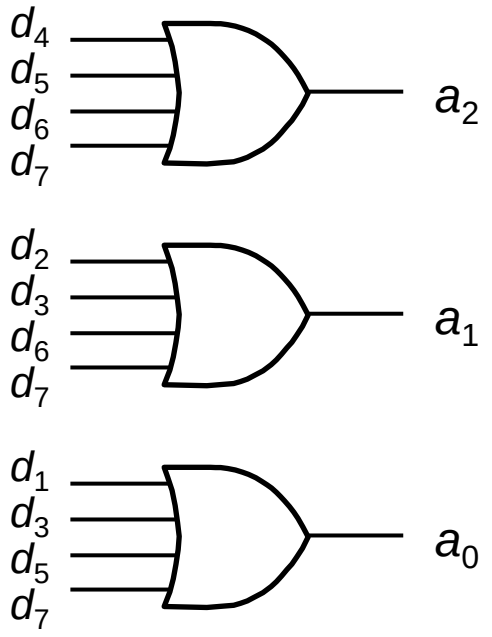


$$a_2 = d_4 + d_5 + d_6 + d_7$$

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

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

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 $a_2\ a_1\ a_0$ = **111** (**incorrect**)

$a_2 = d_4 + d_5 + d_6 + d_7$
$a_1 = d_2 + d_3 + d_6 + d_7$
$a_0 = d_1 + d_3 + d_5 + d_7$

- Two problems to resolve:

  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

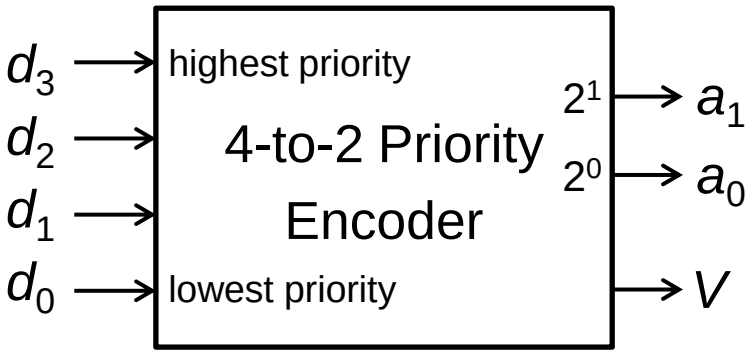
  How to resolve this ambiguity?

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

K-Map of a_1

d_3d_2 \ d_1d_0	00	01	11	10
00	X			
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

K-Map of a_0

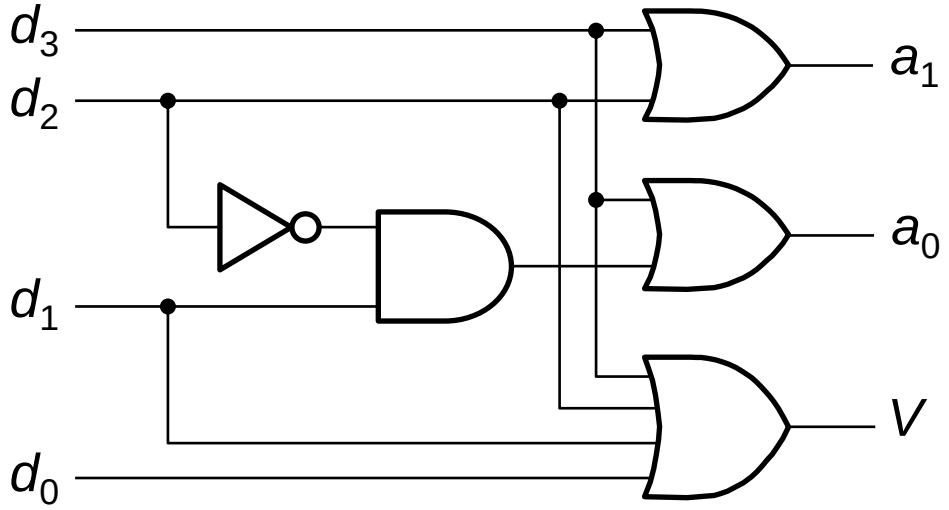
d_3d_2 \ d_1d_0	00	01	11	10
00	X		1	1
01				
11	1	1	1	1
10	1	1	1	1

Output Expressions:

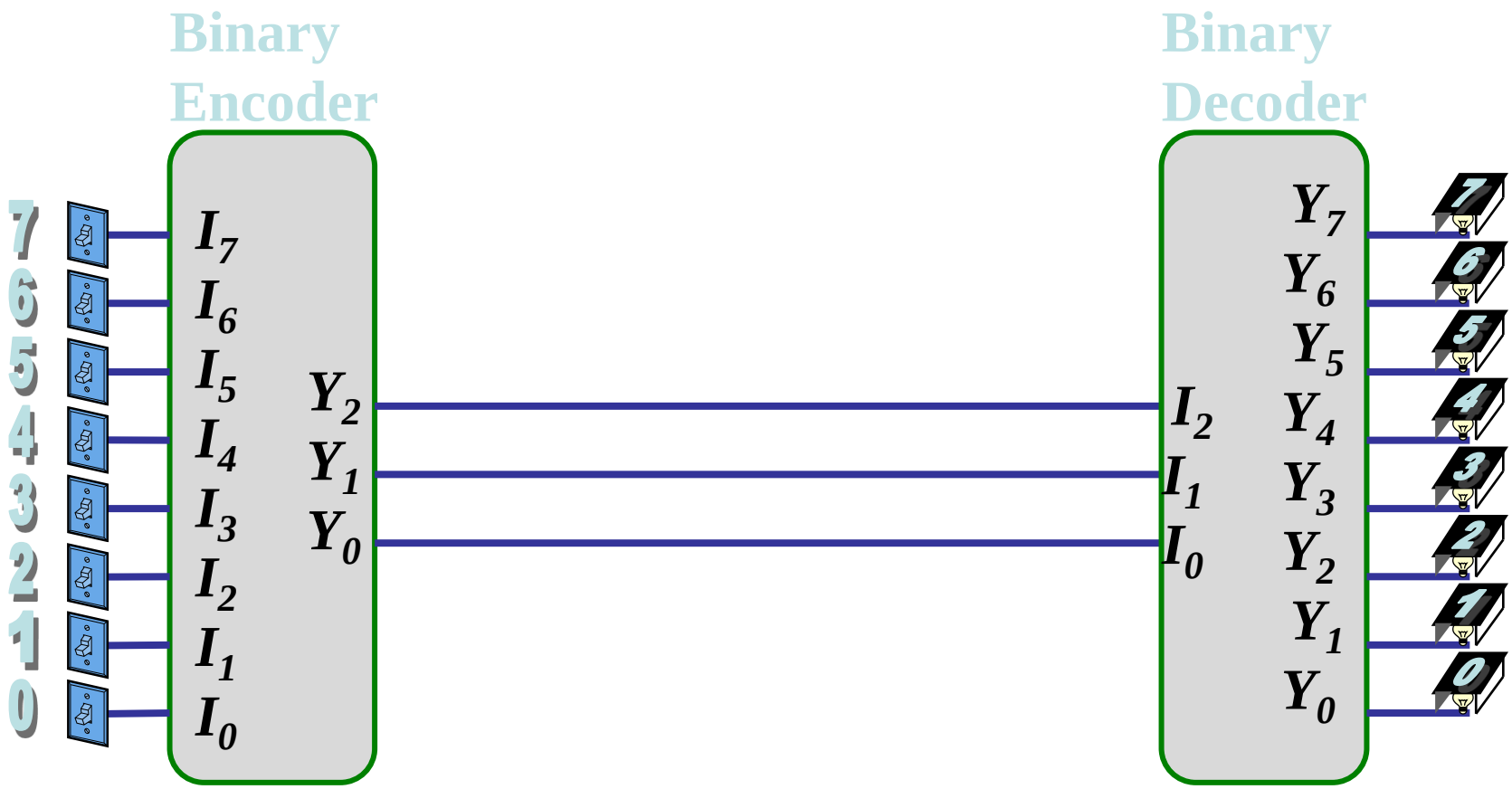
$a_1 = d_3 + d_2$

$a_0 = d_3 + d_1\, d_2'$

$V = d_3 + d_2 + d_1 + d_0$

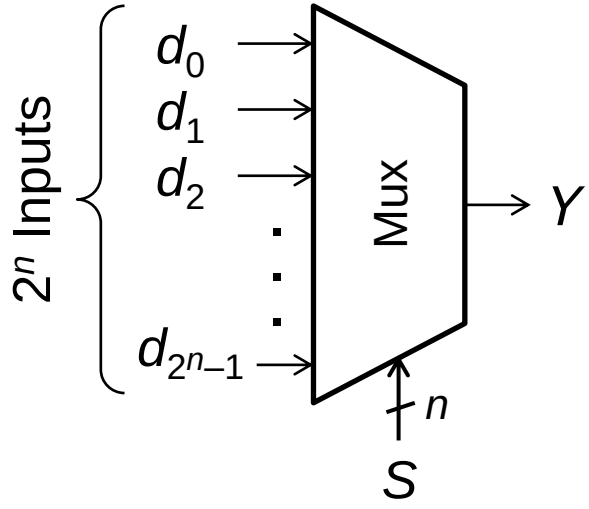


Encoder / Decoder Pairs



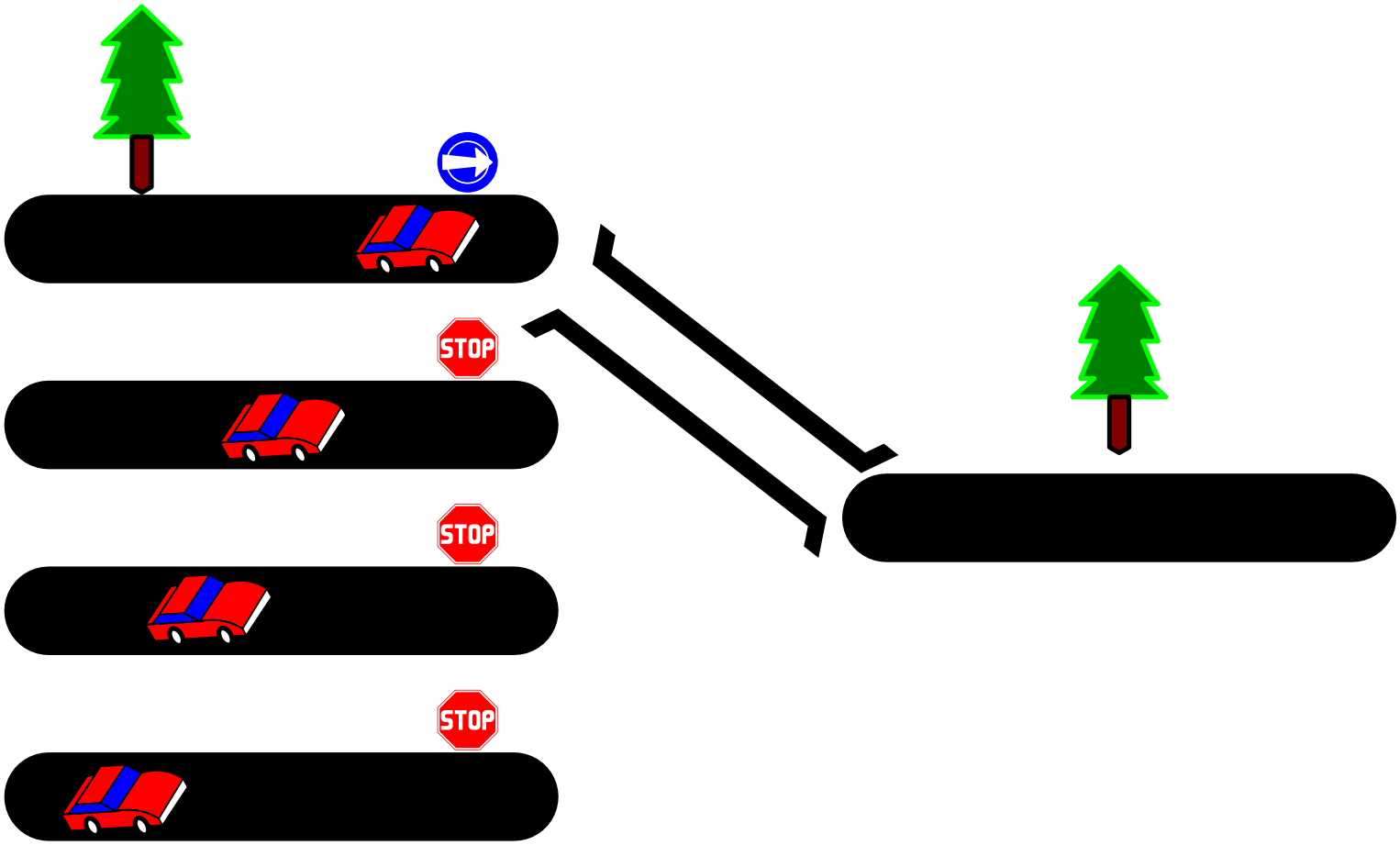
Multiplexers

- 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

Multiplexers



Examples of Multiplexers

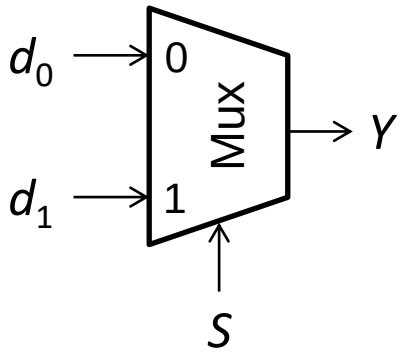
❖ 2-to-1 Multiplexer

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_1S_0 == 00) $Y = d_0$;

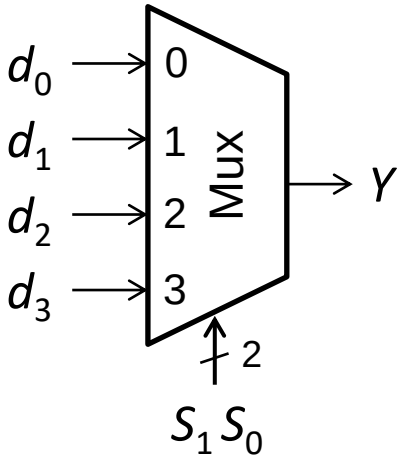
else if (S_1S_0 == 01) $Y = d_1$;

else if (S_1S_0 == 10) $Y = d_2$;

else $Y = d_3$;

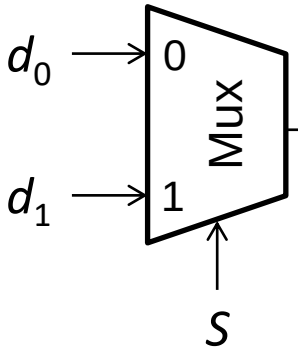
Logic expression:

$$Y = d_0\, S_1'S_0' + d_1\, S_1'S_0 \;\; + d_2\, S_1S_0' + d_3\, S_1S_2$$

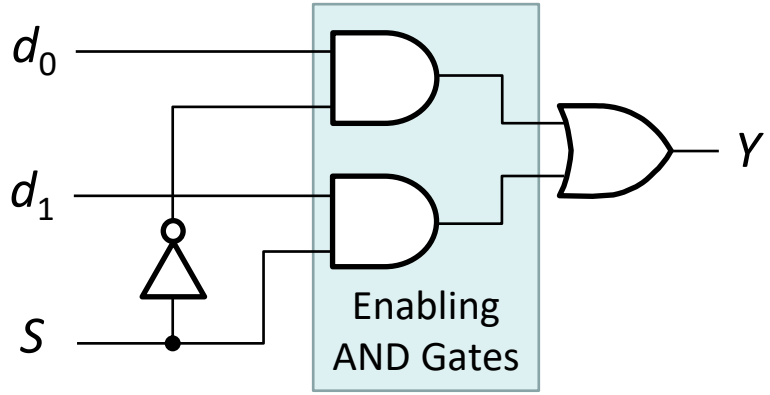


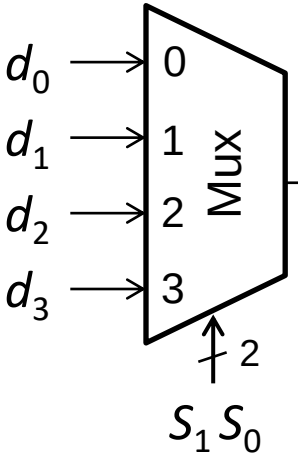
Inputs						Output
S_1	S_0	d_0	d_1	d_2	d_3	Y
0	0	0	X	X	X	$0 = d_0$
0	0	1	X	X	X	$1 = d_0$
0	1	X	0	X	X	$0 = d_1$
0	1	X	1	X	X	$1 = d_1$
1	0	X	X	0	X	$0 = d_2$
1	0	X	X	1	X	$1 = d_2$
1	1	X	X	X	0	$0 = d_3$
1	1	X	X	X	1	$1 = d_3$

Implementing Multiplexers

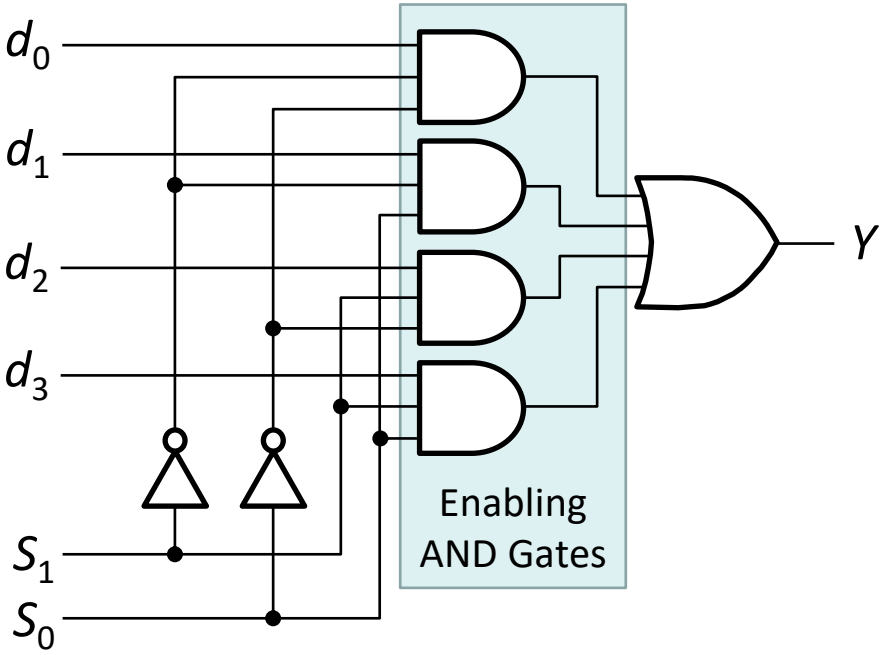


$$Y = d_0\,S' + d_1\,S$$



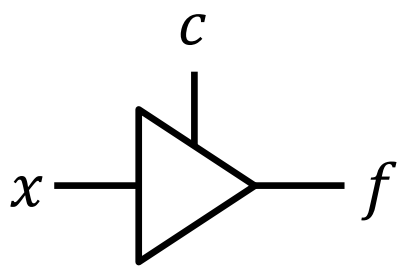


$$Y = d_0\,S_1'S_0' + d_1\,S_1'S_0 + d_2\,S_1S_0' + d_3\,S_1S_0$$



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***

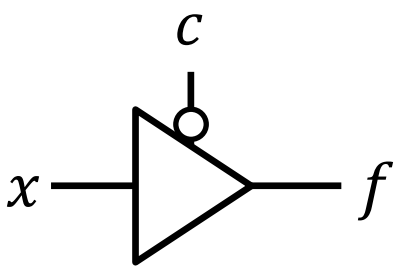


3-state gate

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

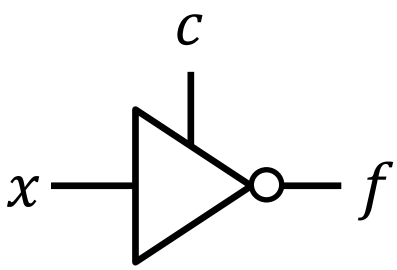
Variations of the 3-State Gate

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



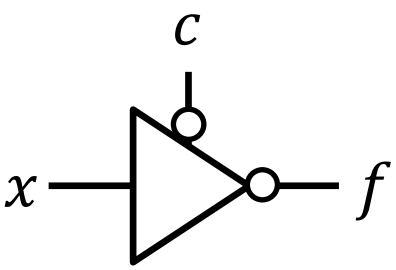
inverted *c*

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



inverted *f*

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

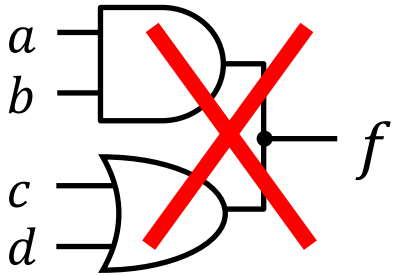


inverted *c*, *f*

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

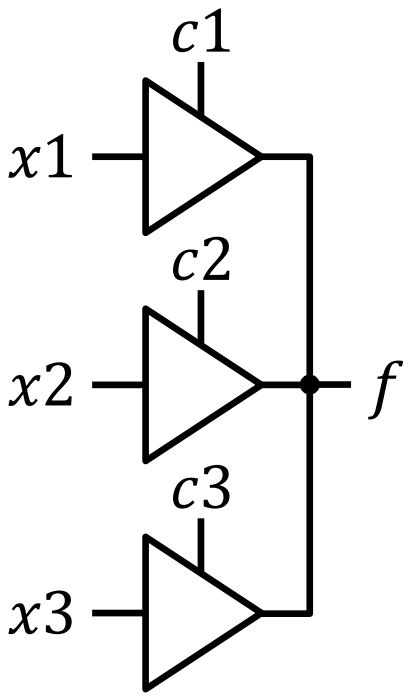


This will result in a **short circuit** that will burn the gates

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

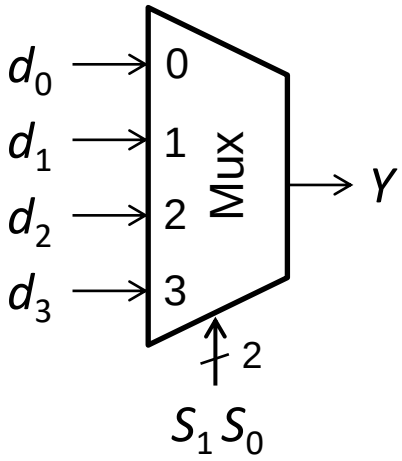


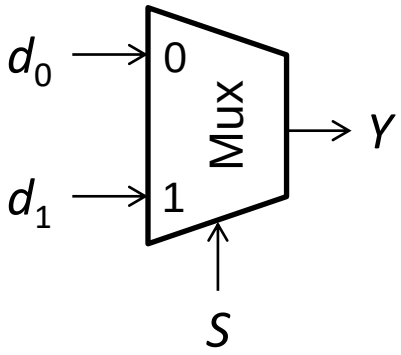
c1	c2	c3	f
0	0	0	z
1	0	0	x1
0	1	0	x2
0	0	1	x3
0	1	1	Burn
1	0	1	Burn
1	1	0	Burn
1	1	1	Burn

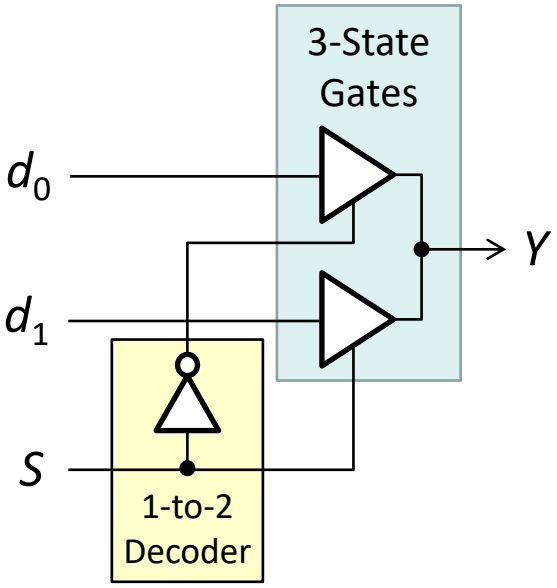
Implementing Multiplexers with 3-State Gates

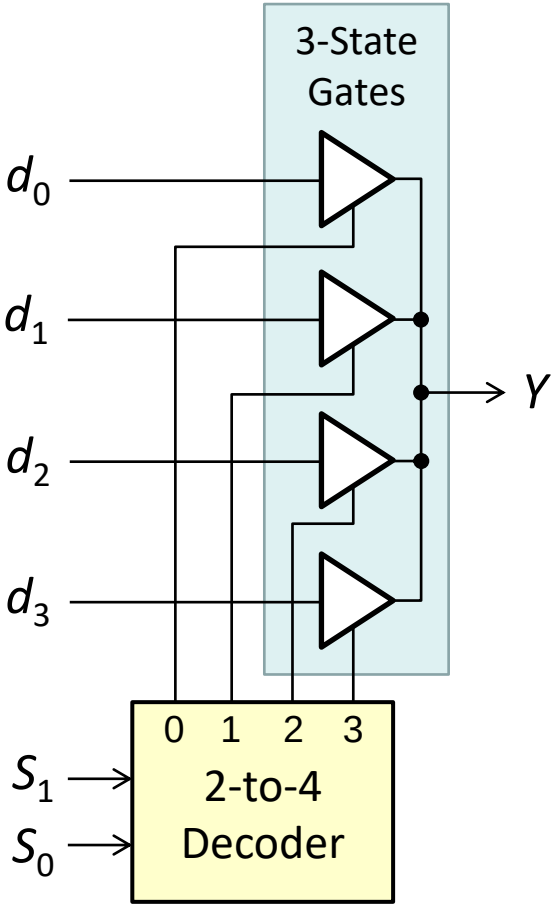
A Multiplexer can also be implemented using:

1. A decoder
2. Three-state gates



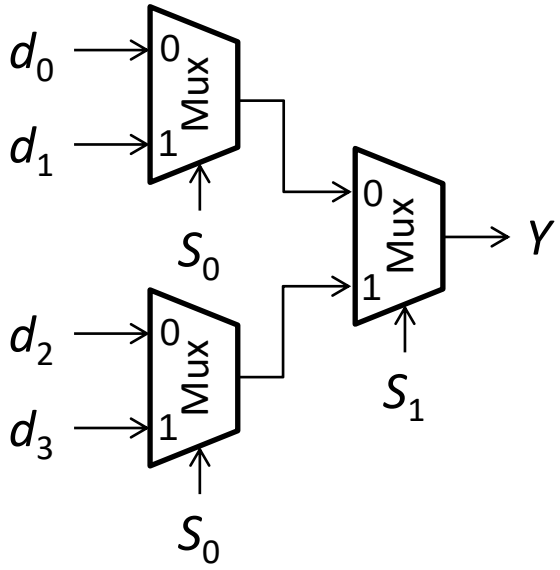




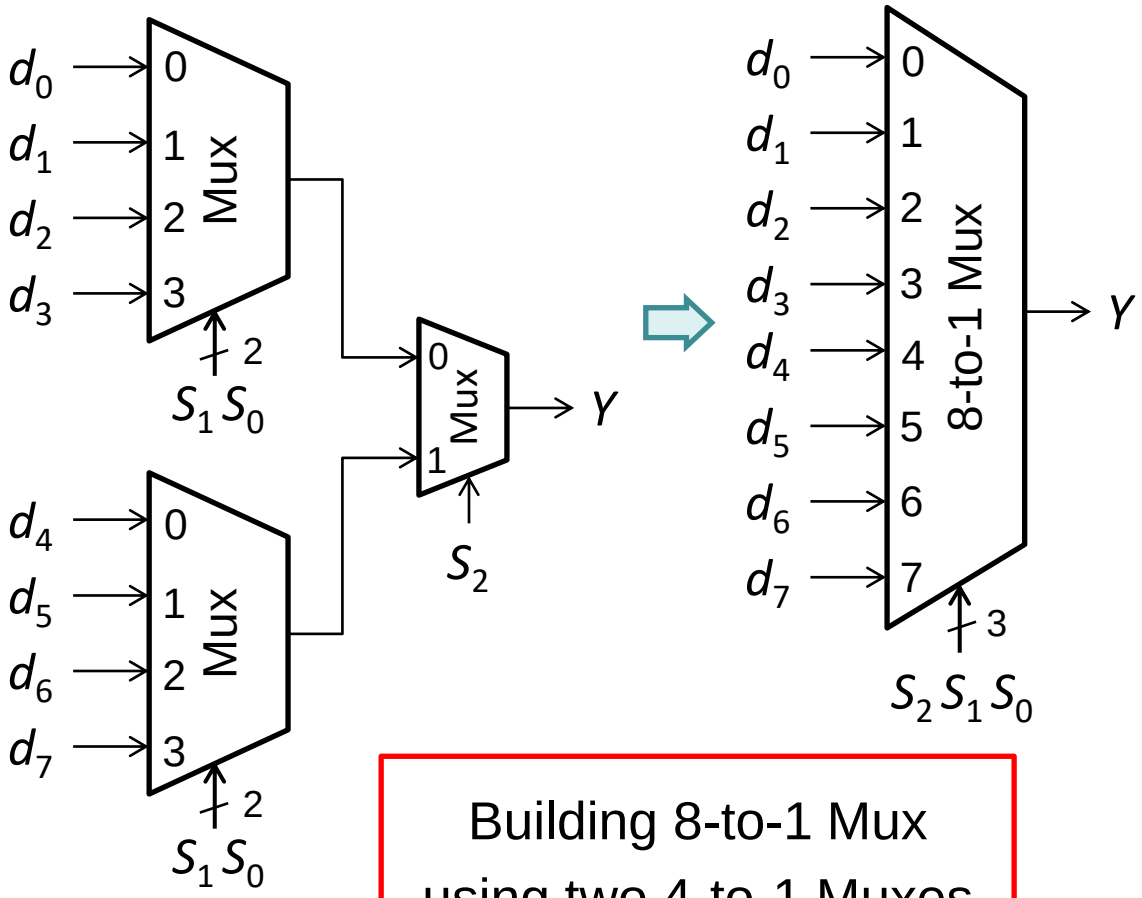


Building Larger Multiplexers

Larger multiplexers can be built hierarchically using smaller ones



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



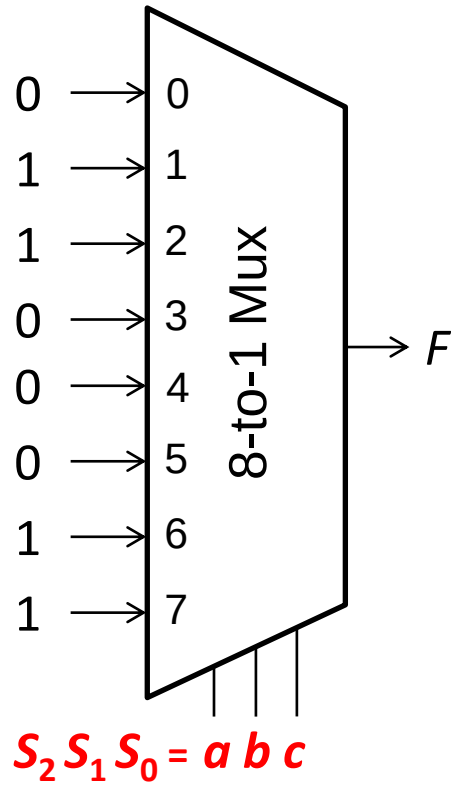
Building 8-to-1 Mux using two 4-to-1 Muxes and a 2-to-1 Mux

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 $F(a, b, c) = \sum(1, 2, 6, 7)$ 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

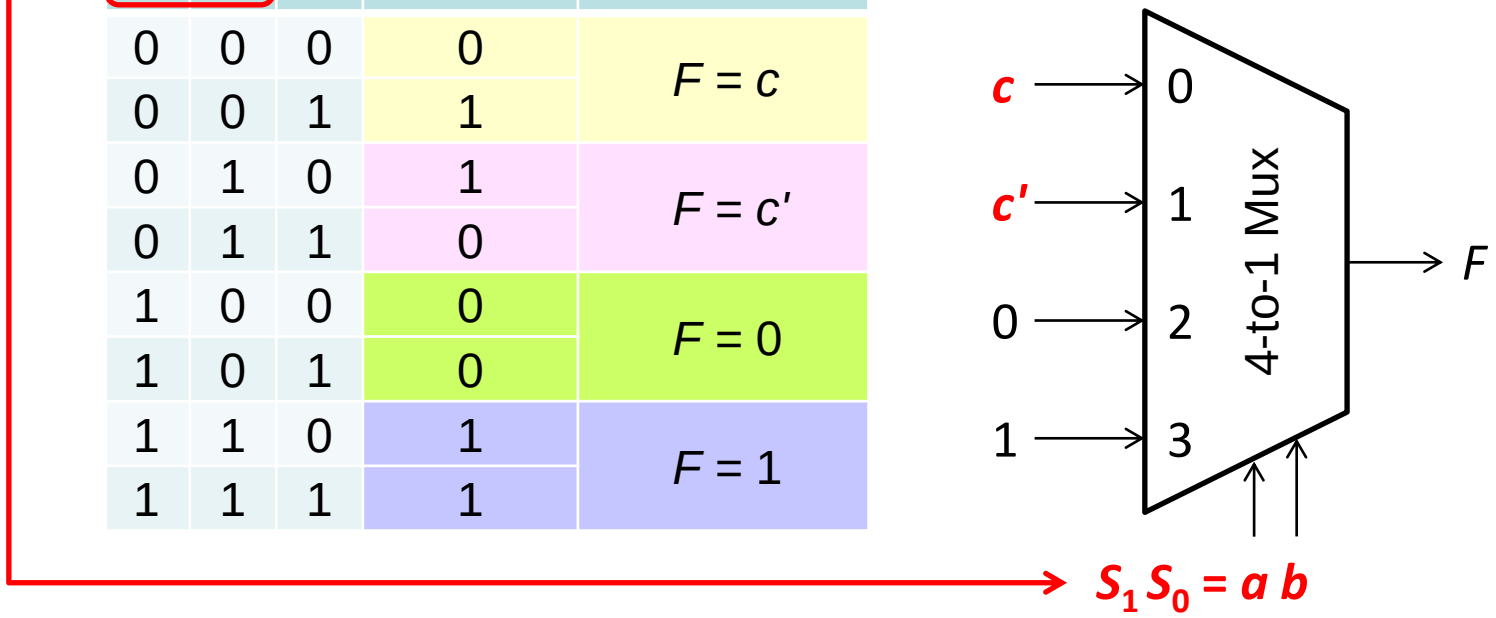
Inputs			Output
a	b	c	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Better Solution with a Smaller Multiplexer

- Re-implement $F(a, b, c) = \sum(1, 2, 6, 7)$ 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

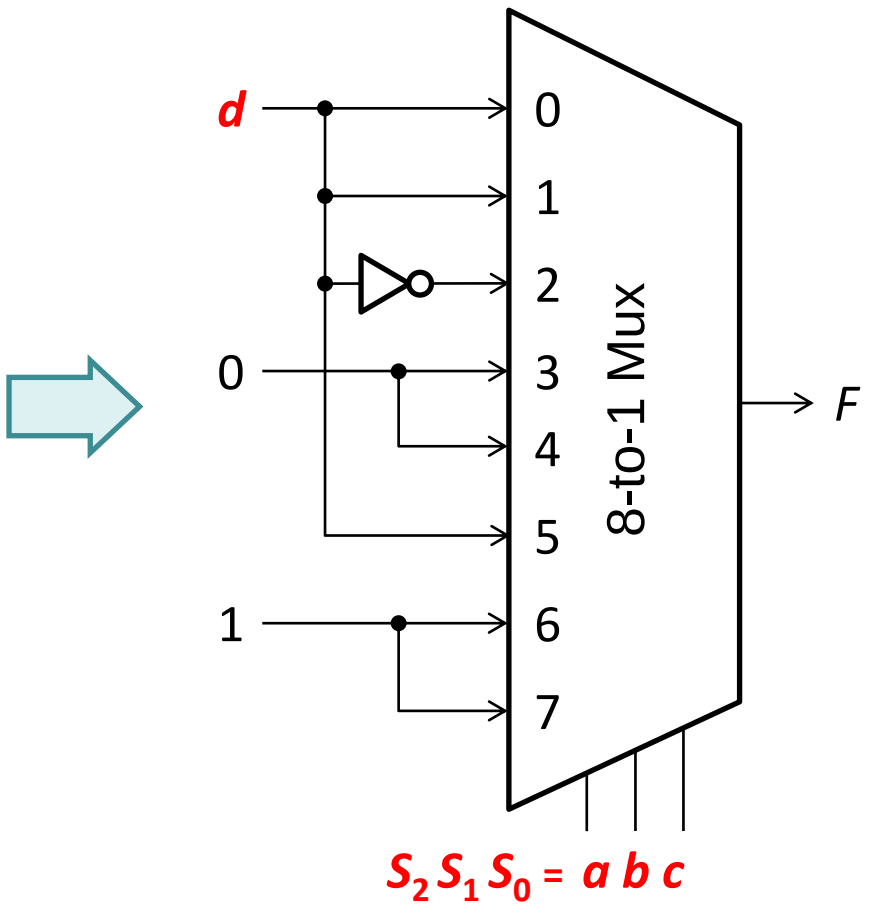
Inputs			Output	Comment
a	b	c	F	F
0	0	0	0	$F = c$
0	0	1	1	
0	1	0	1	$F = c'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	



Implementing Functions: Example 2

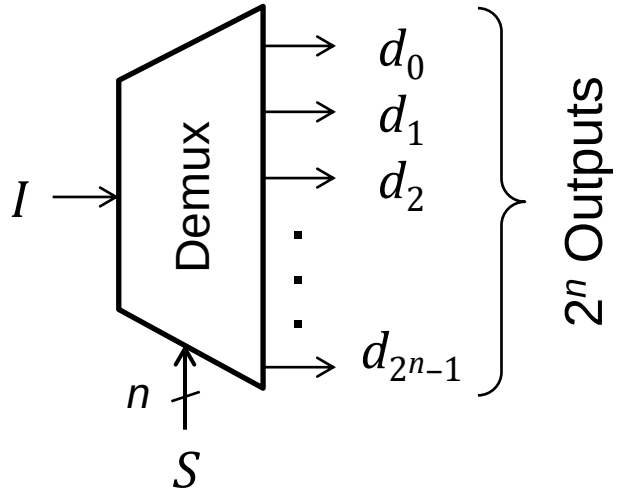
Implement $F(a, b, c, d) = \sum(1,3,4,11,12,13,14,15)$ using 8-to-1 Mux

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	



Demultiplexer

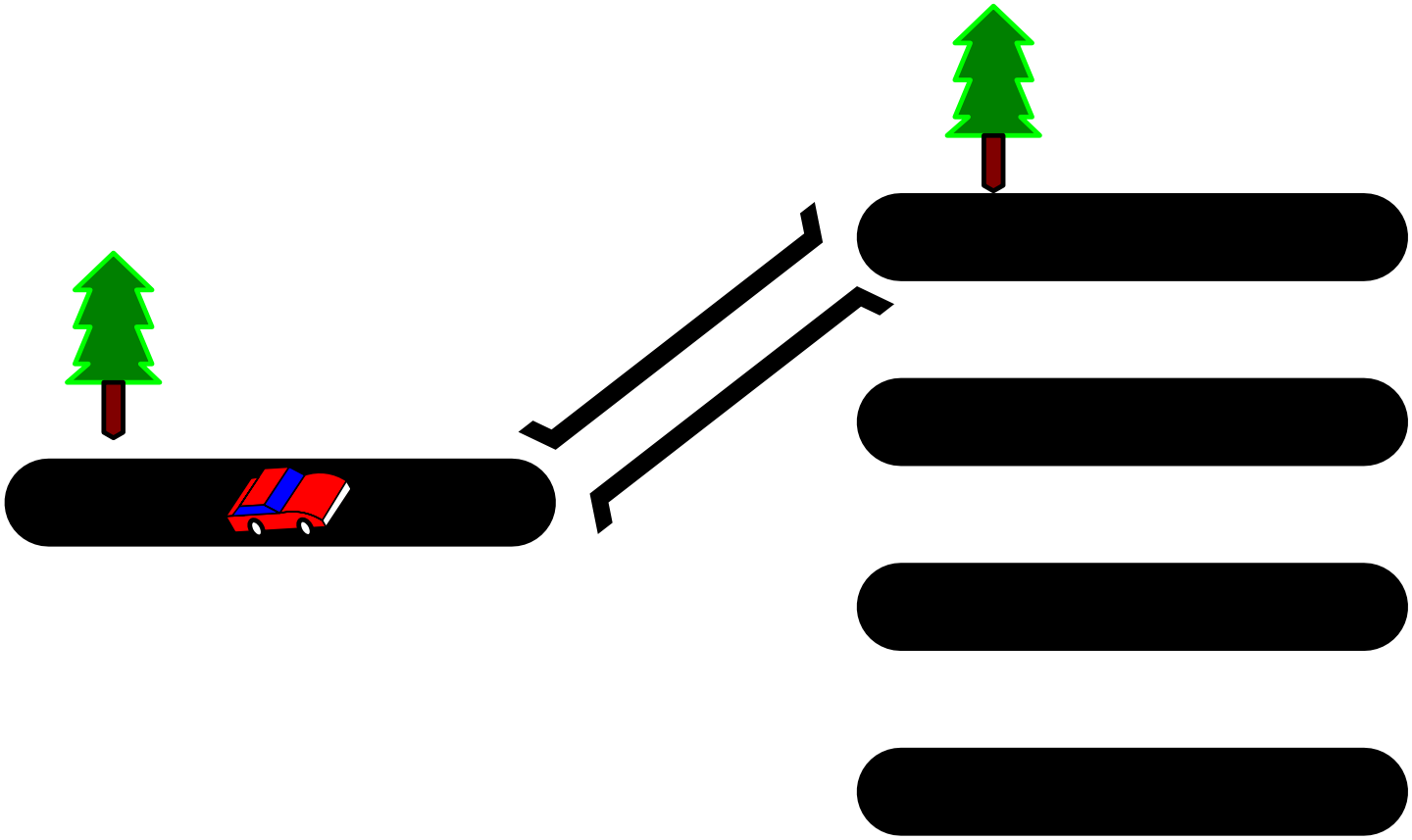
- Performs the inverse operation of a Multiplexer
- A Demultiplexer (or Demux) is a combinational circuit that has:
  1. One data input I
  2. An n-bit select input S
  3. A maximum of 2^n data outputs



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

  According to the select input S

Demultiplexer

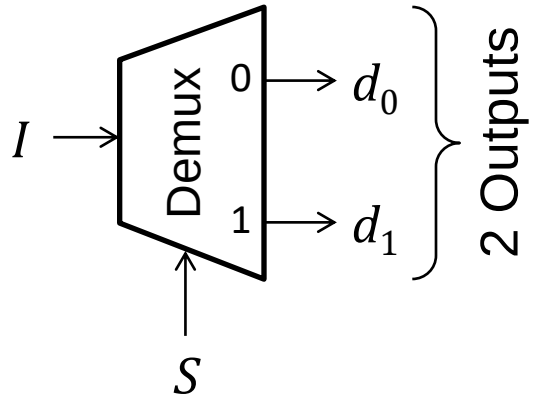


Examples of Demultiplexers

- 1-to-2 Demultiplexer

if (S == 0) { $d_0 = I$; $d_1 = 0$; }
else { $d_1 = I$; $d_0 = 0$; }
Output expressions:

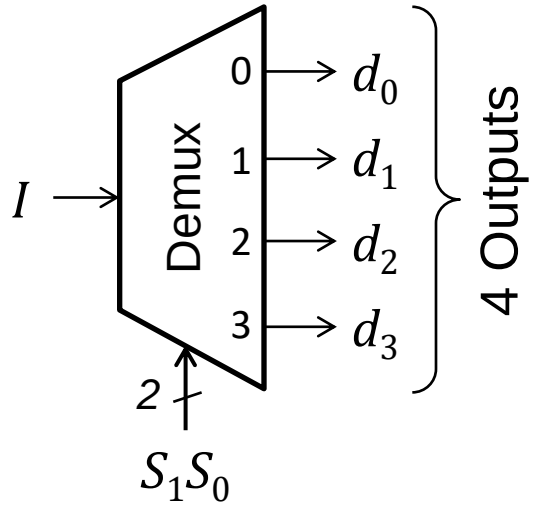
$d_0 = I\ S'$; $d_1 = I\ S$



- 1-to-4 Demultiplexer

if (S_1S_0 == 00) { $d_0 = I$; $d_1 = d_2 = d_3 = 0$; }
else if (S_1S_0 == 01) { $d_1 = I$; $d_0 = d_2 = d_3 = 0$; }
else if (S_1S_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:



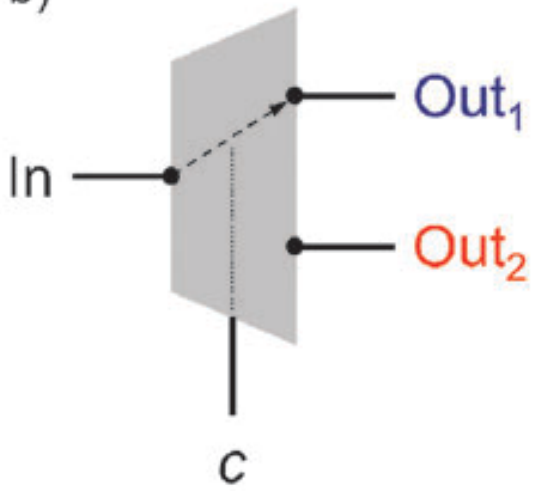
$d_0 = I\ S_1'S_0'$; $d_1 = I\ S_1'S_0$; $d_2 = I\ S_1\ S_0'$; $d_3 = I\ S_1\ S_0$

1-2 Demultiplexer

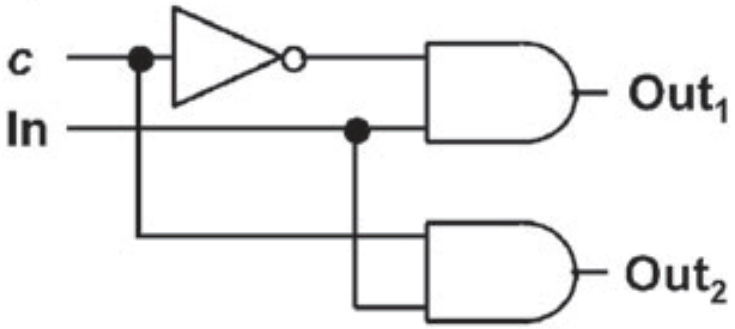
a)

In	c	Out_1	Out_2
0	0	0	0
1	0	1	0
0	1	0	0
1	1	0	1

b)

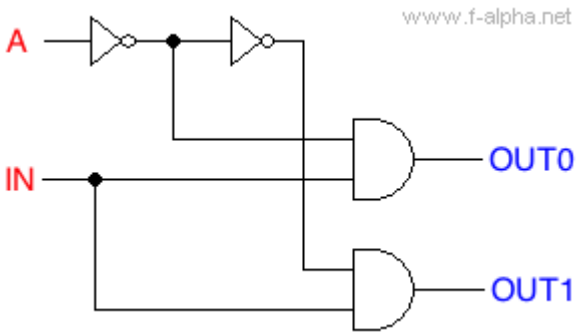


c)

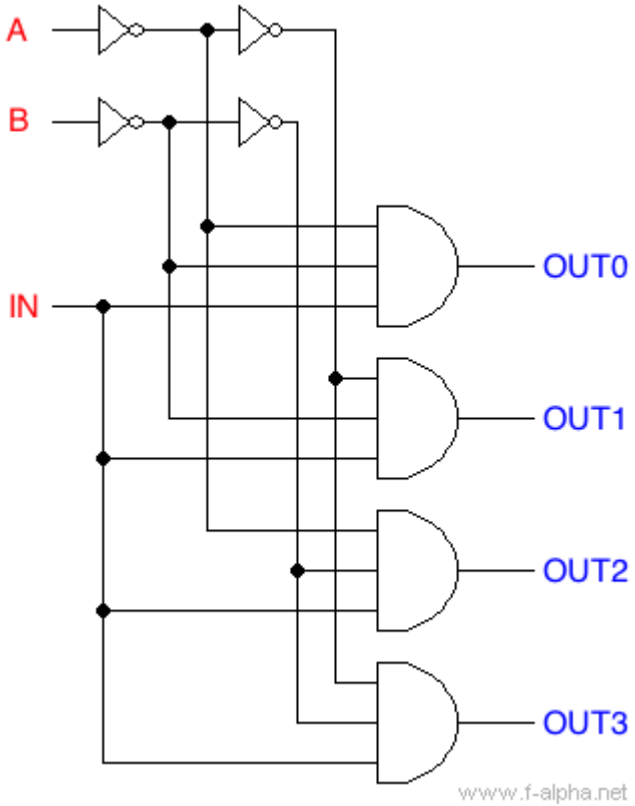


Examples of Demultiplexers

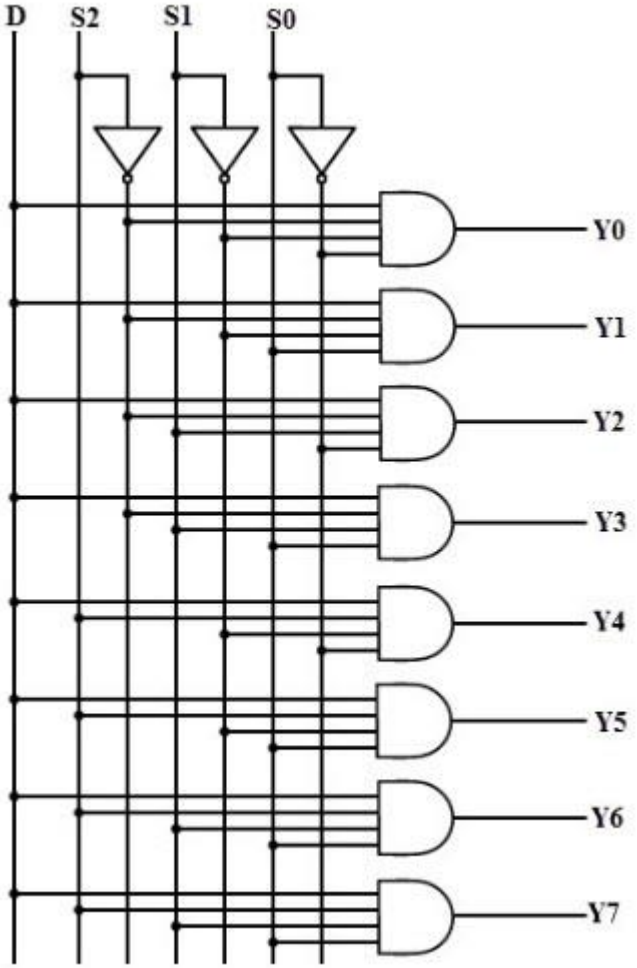
1 - 2



1 - 4



1 - 8



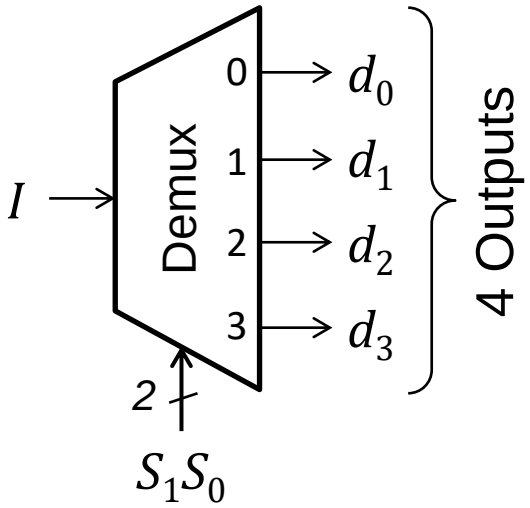
Demultiplexer = Decoder with Enable

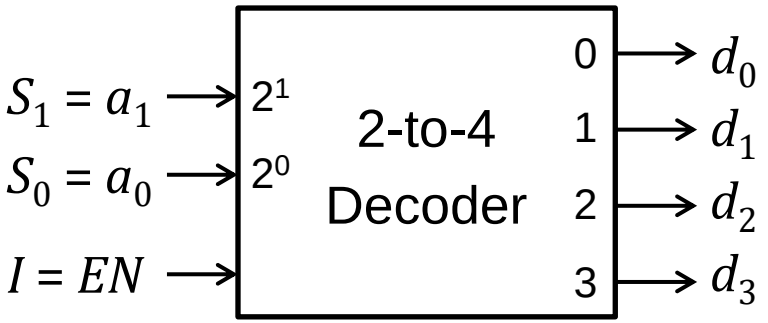
- A 1-to-4 demux is equivalent to a 2-to-4 decoder with enable

  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

Multiplexer / DeMultiplexer Pairs