

Functional Blocks

COE 202

Digital Logic Design

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

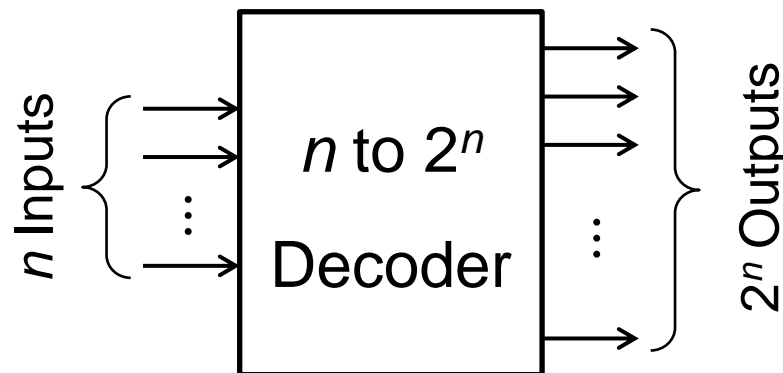
- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Demultiplexers
- ❖ Design Examples

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

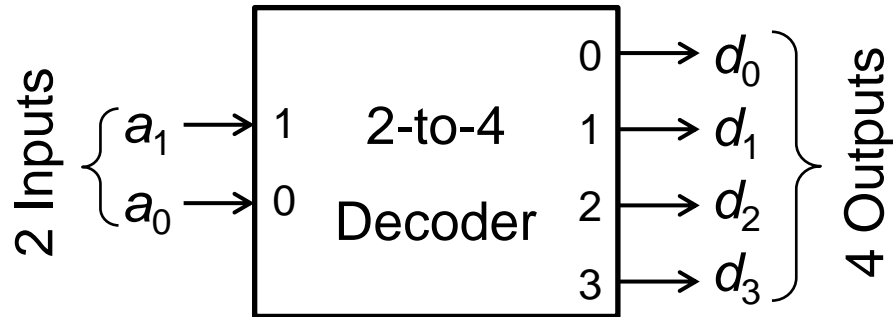
Binary Decoders

- ❖ 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**



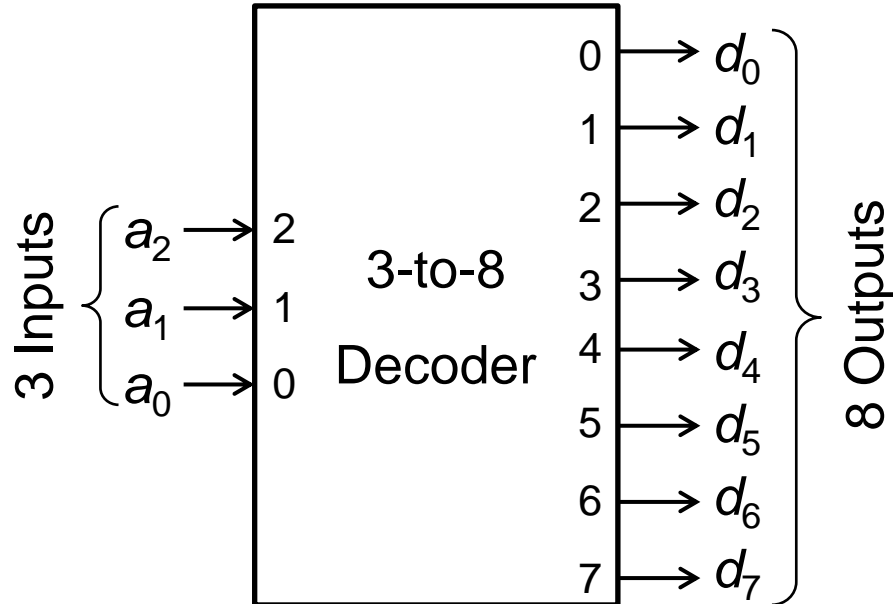
A decoder can have less than 2^n outputs if some input codes are unused

Examples of Binary Decoders



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**

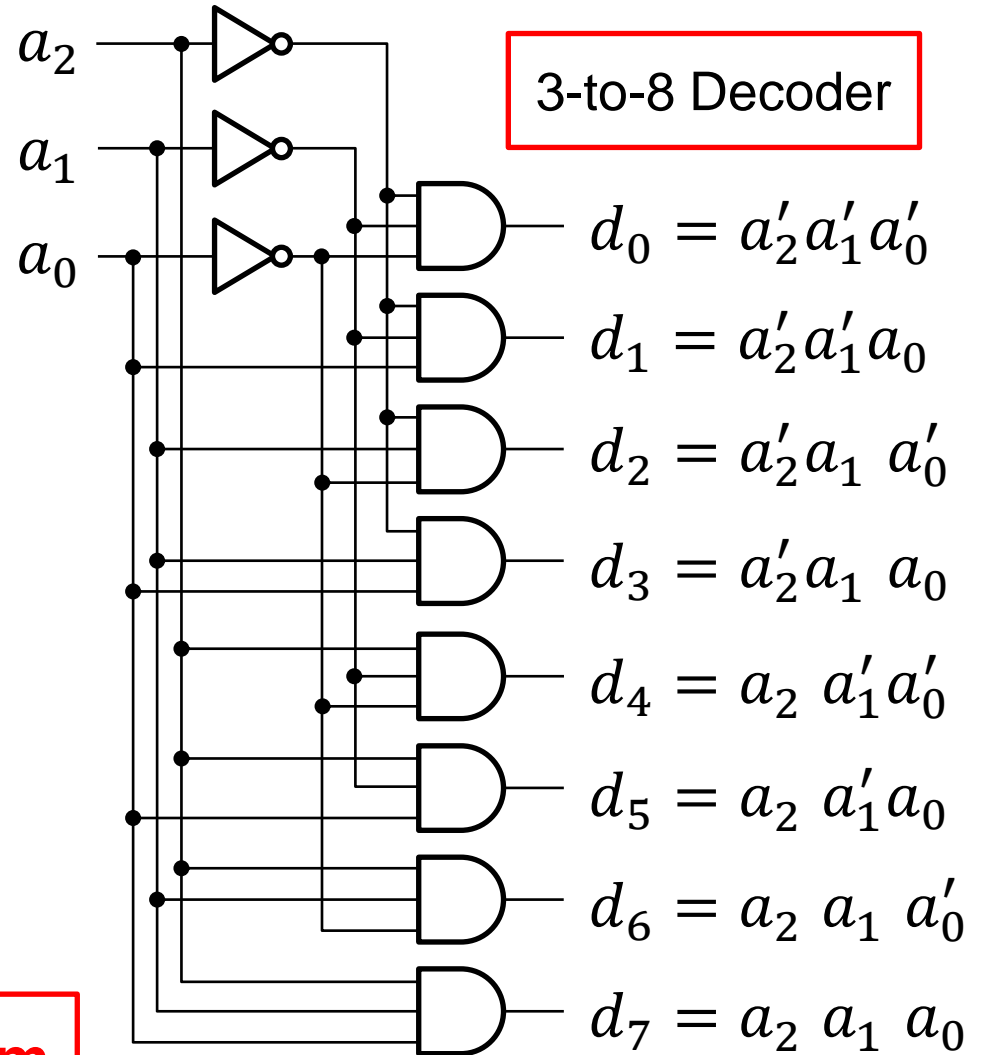
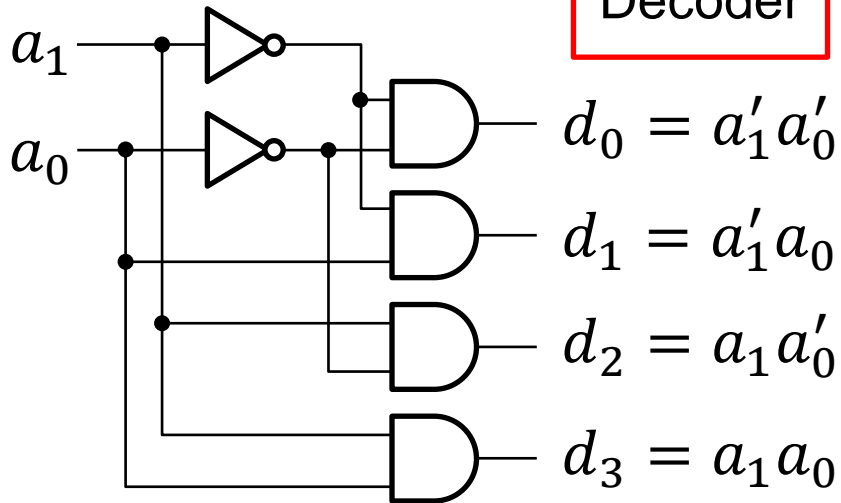


Inputs			Outputs							
a_2	a_1	a_0	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

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



Each decoder output is a **minterm**

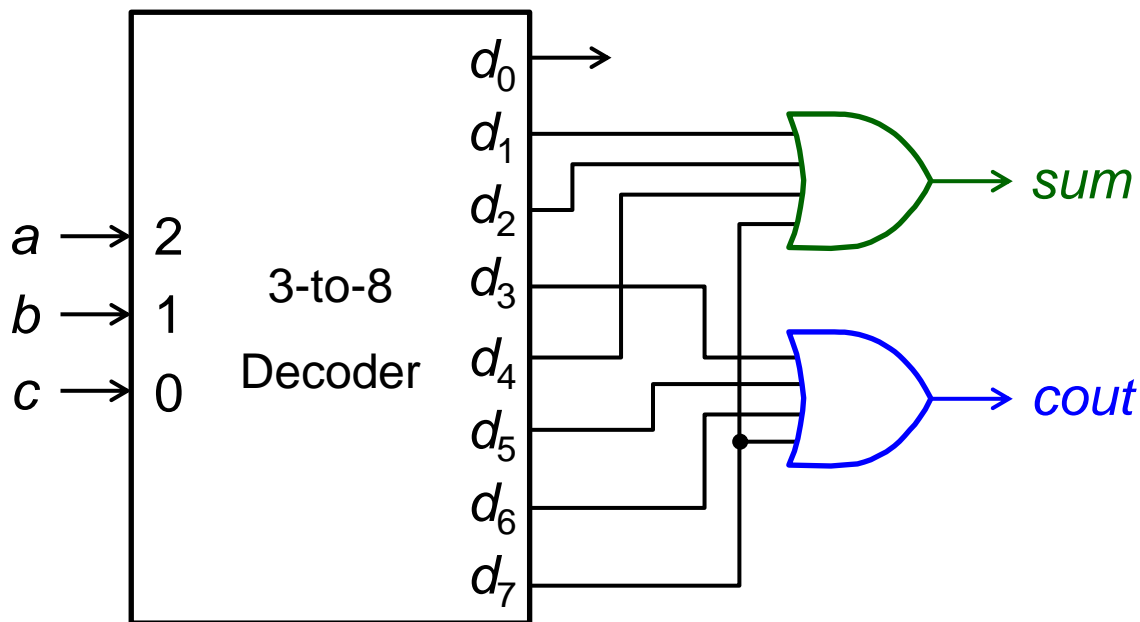
Using Decoders to Implement Functions

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

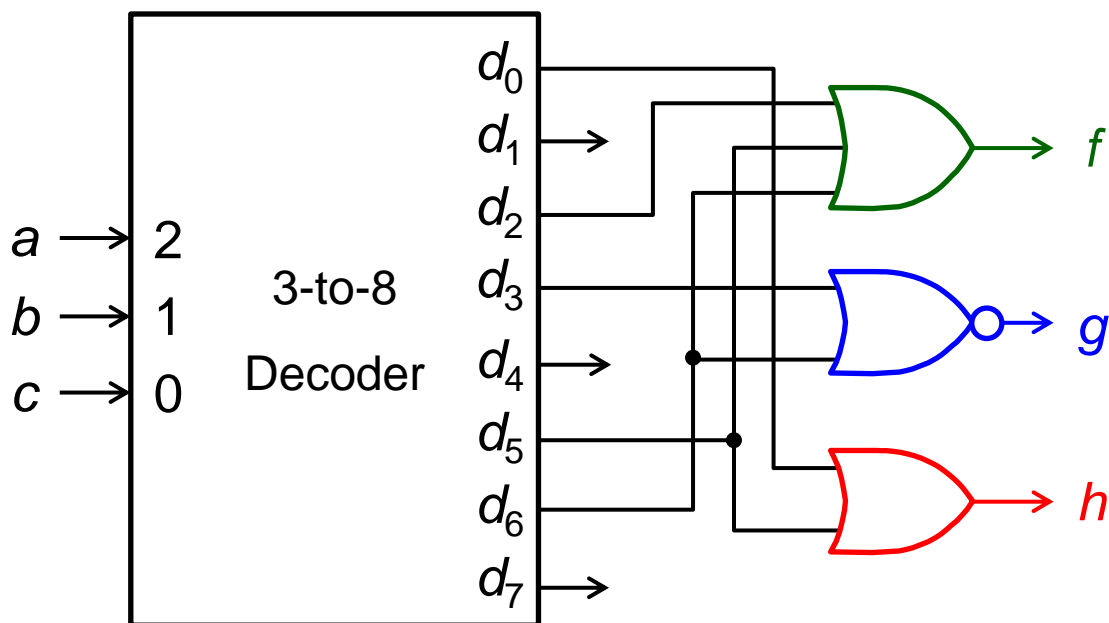
Inputs			Outputs	
a	b	c	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



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

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

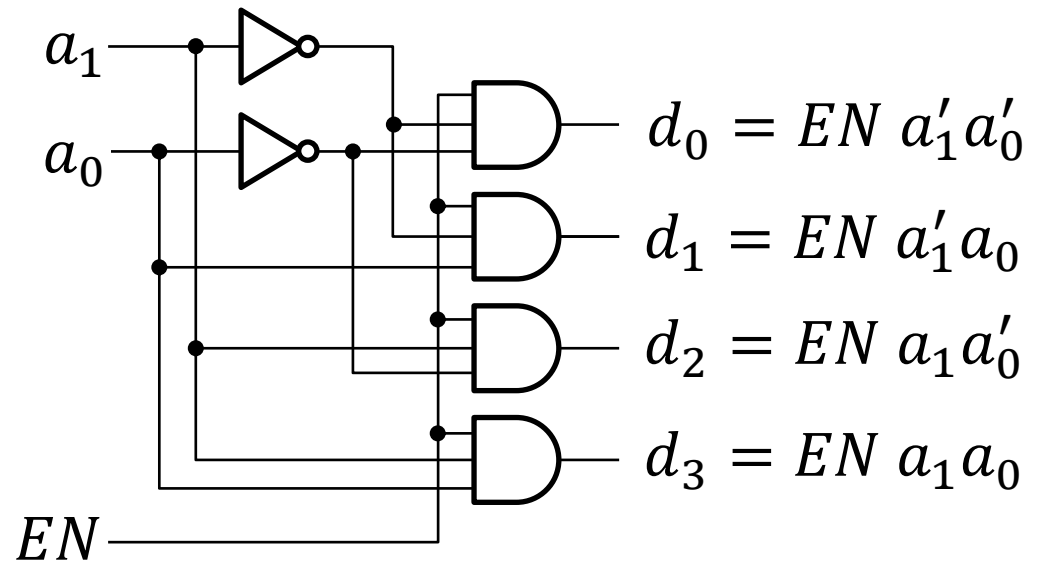
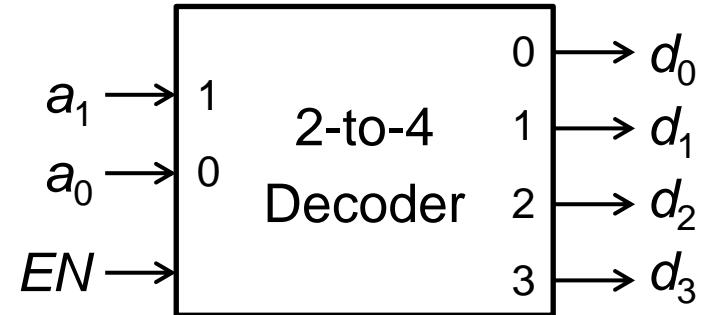


2-to-4 Decoder with Enable Input

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

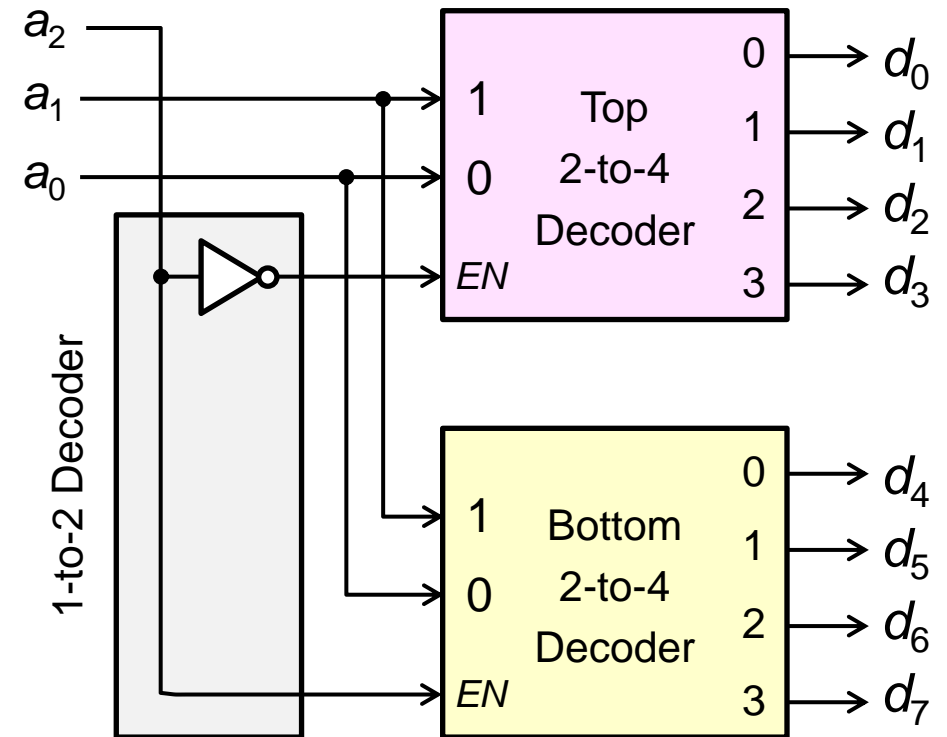


Building Larger Decoders

- ❖ Larger decoders can be built 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)

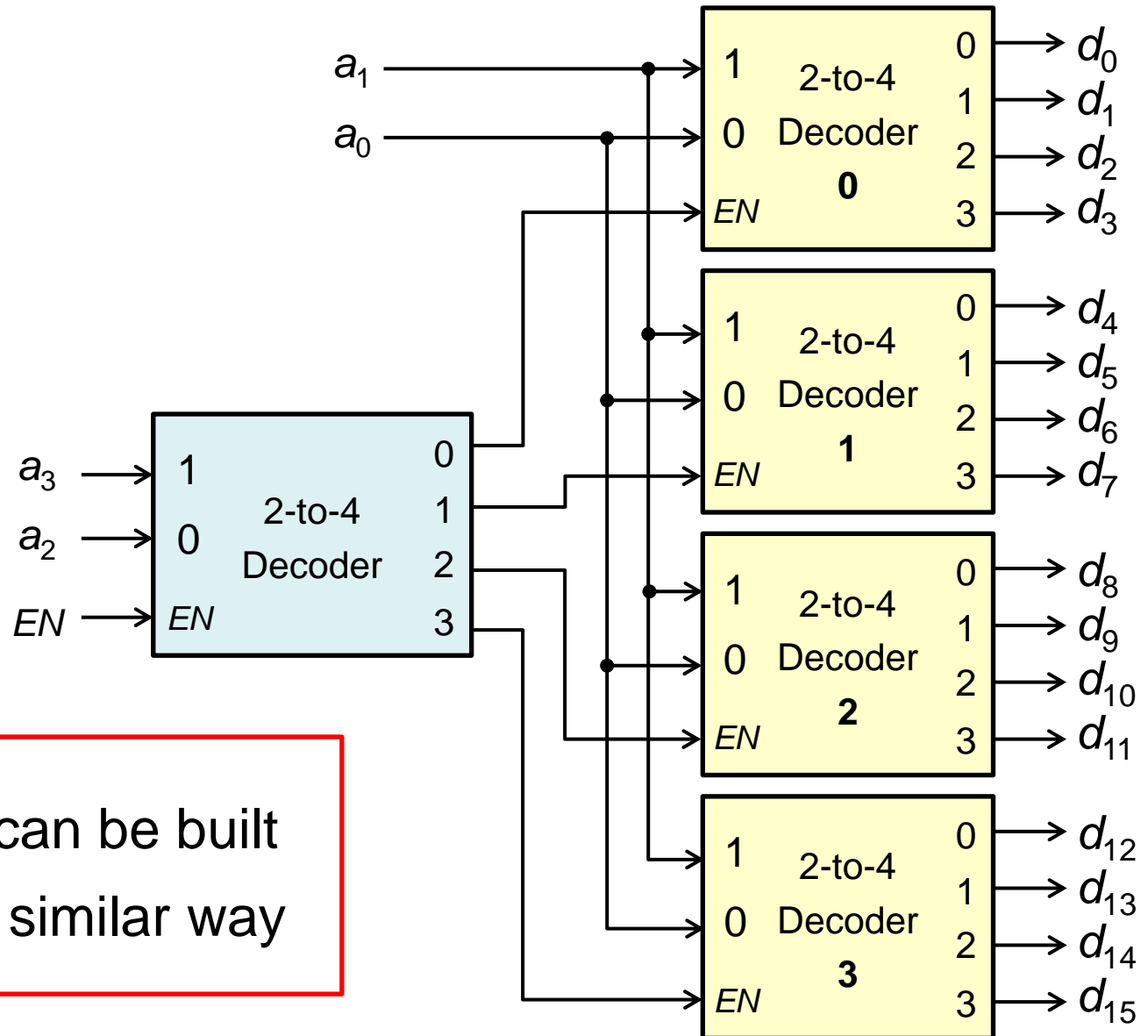
Inputs			Outputs							
a_2	a_1	a_0	d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



Building Larger Decoders

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

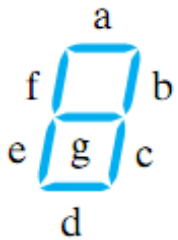
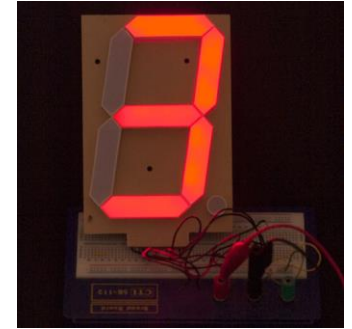
Larger decoders can be built hierarchically in a similar way



BCD to 7-Segment Decoder

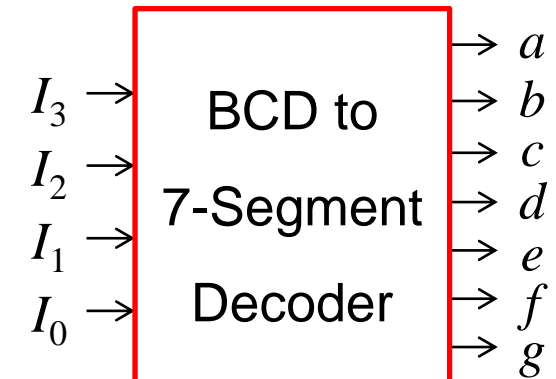
❖ Seven-Segment Display:

- ❖ 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



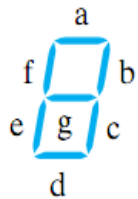
BCD to 7-Segment Decoder

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:

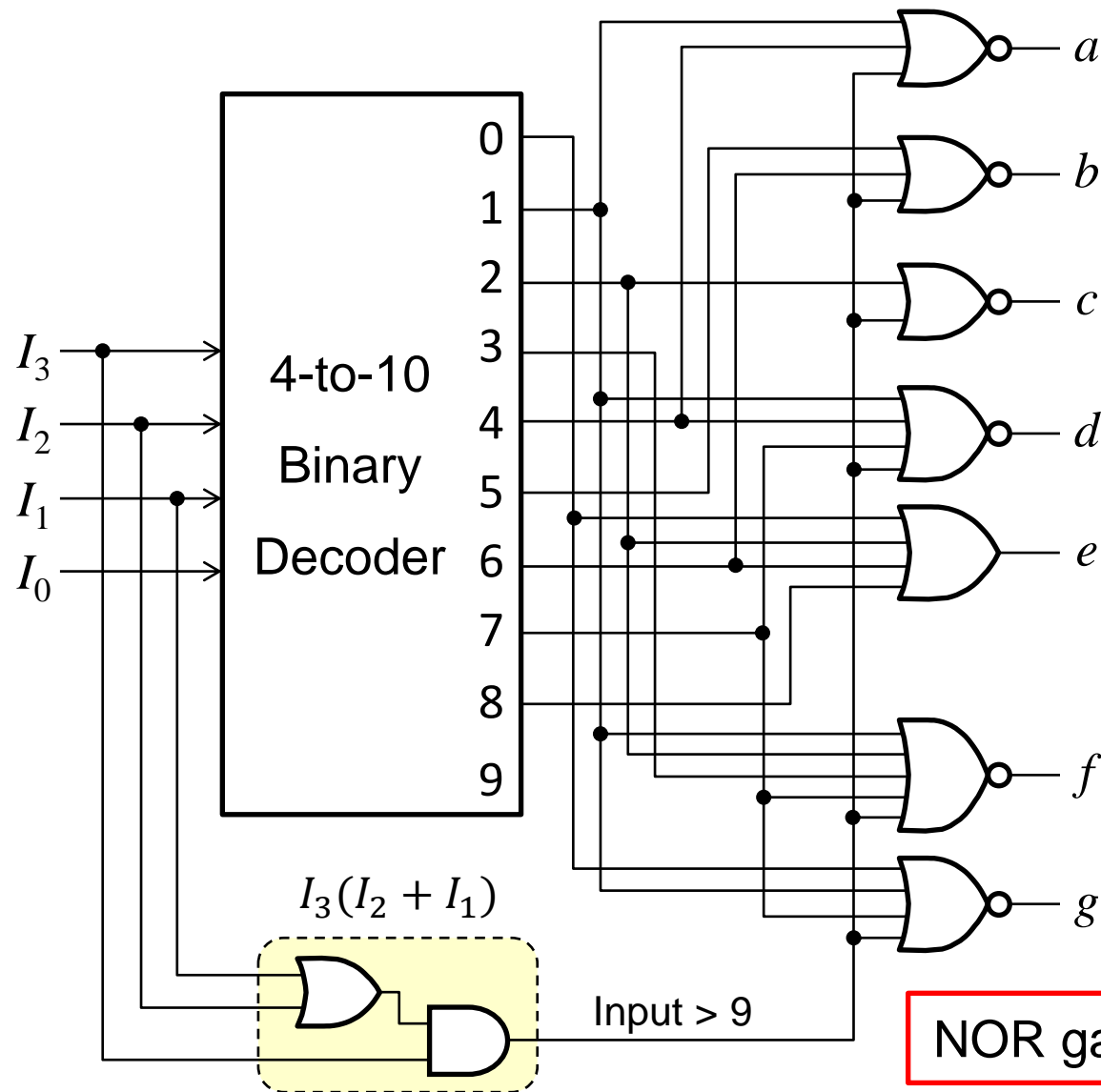
- ✧ A binary decoder
- ✧ Additional gates



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

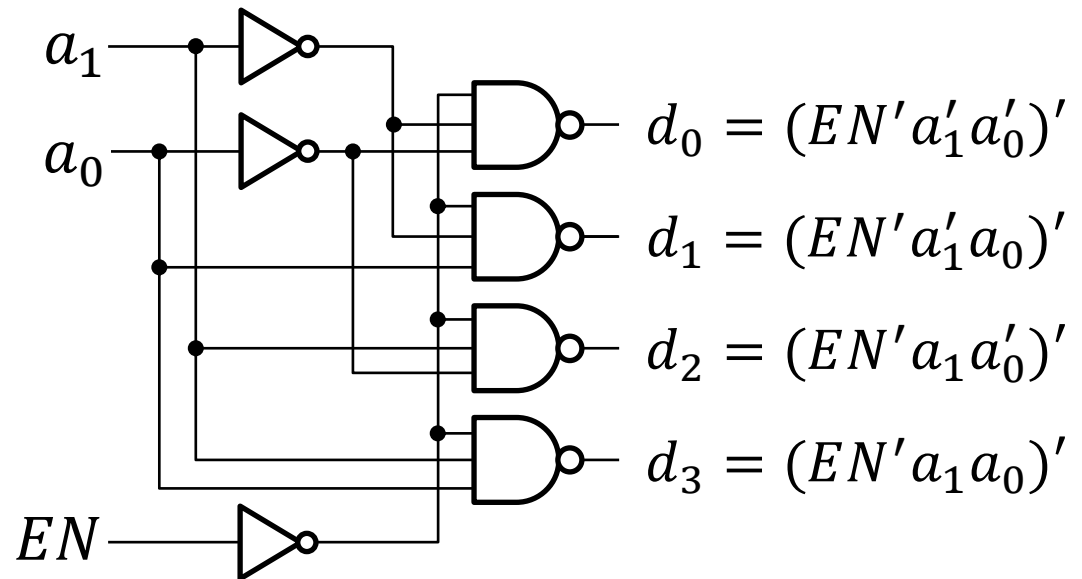
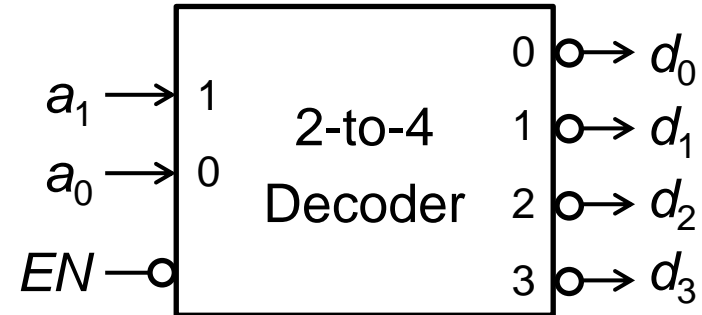
NOR gate is used for 0's

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

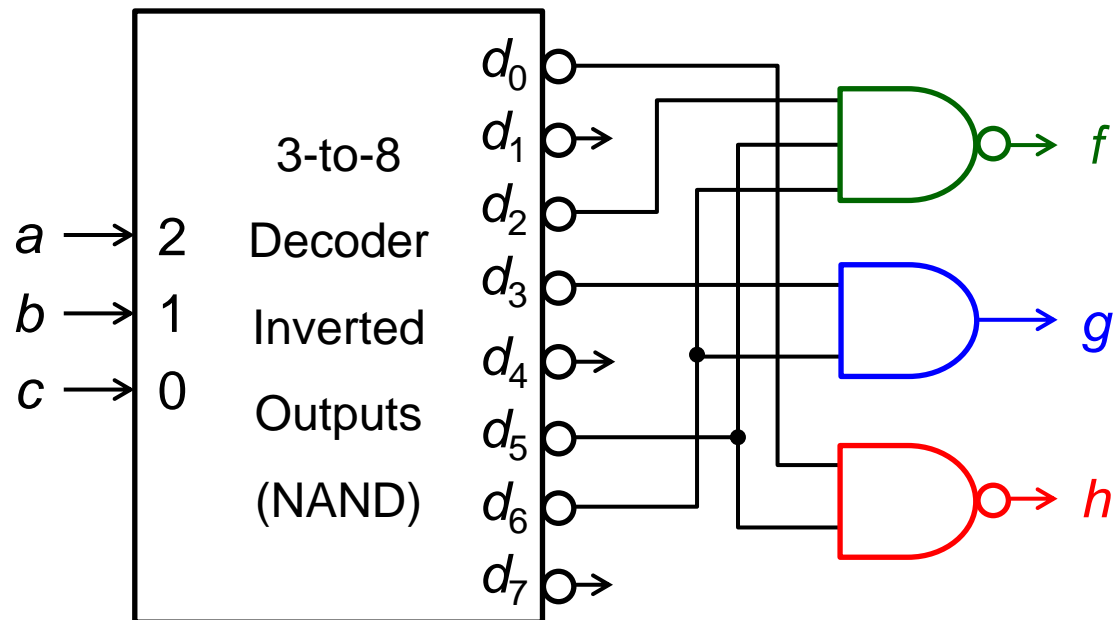
Some decoders are constructed with NAND gates. Their outputs are inverted. The Enable input is also active low (Enable if zero)



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



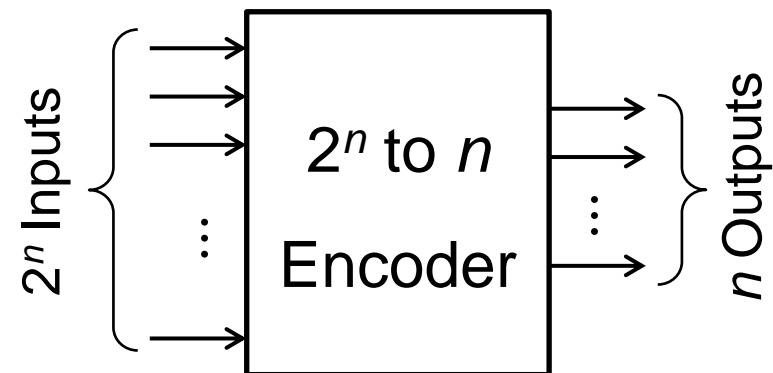
Next . . .

- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Demultiplexers
- ❖ Design Examples

Encoders

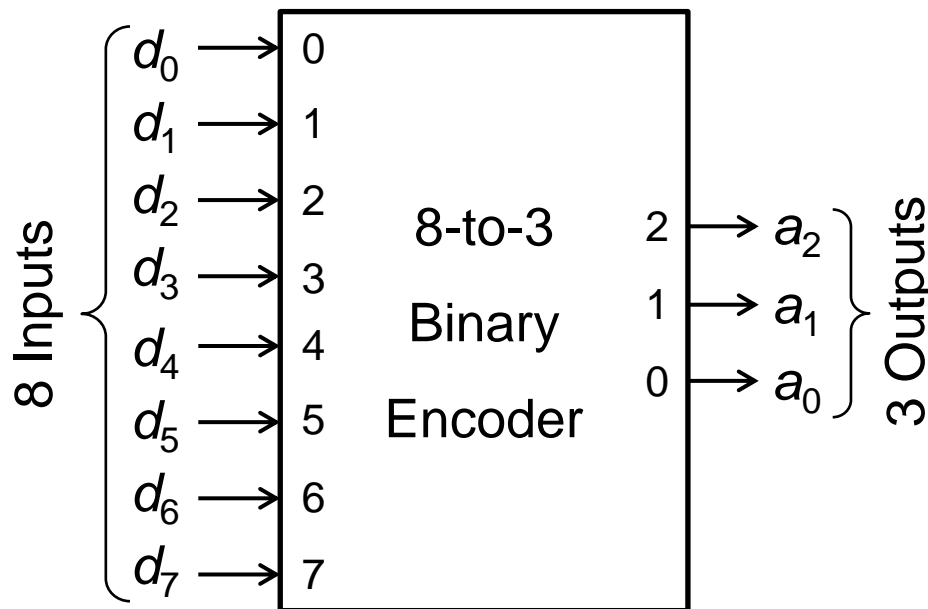
- ❖ An encoder performs the opposite operation of a decoder
- ❖ It converts a 2^n input to an n -bit output code
- ❖ The output indicates which input is active (logic **1**)
- ❖ Typically, **one** input should be **1** and all others must be **0**'s
- ❖ The conversion of input to output is called **encoding**

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



Example of an 8-to-3 Binary Encoder

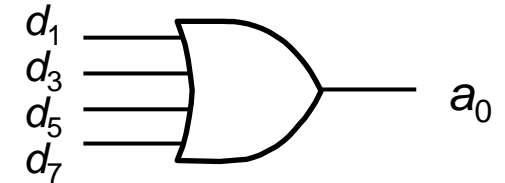
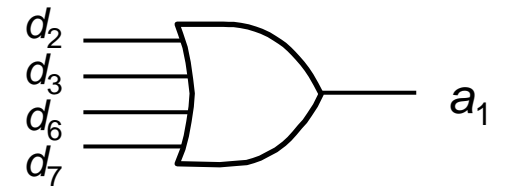
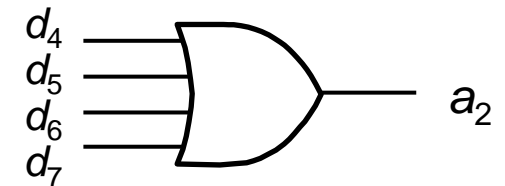
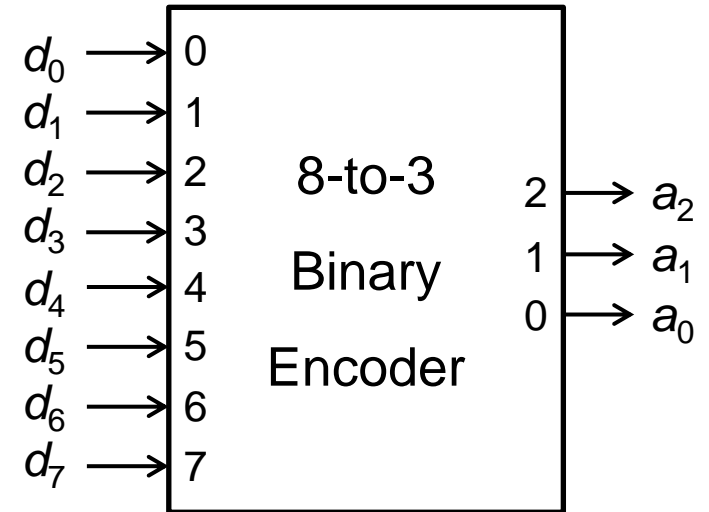
- ❖ 8 inputs, 3 outputs, only **one input** is **1**, all others are **0**'s
- ❖ Encoder generates the output binary code for the active input
- ❖ Output is **not specified** if more than one input is **1**



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 $a_2 a_1 a_0 = \mathbf{111}$ (**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?

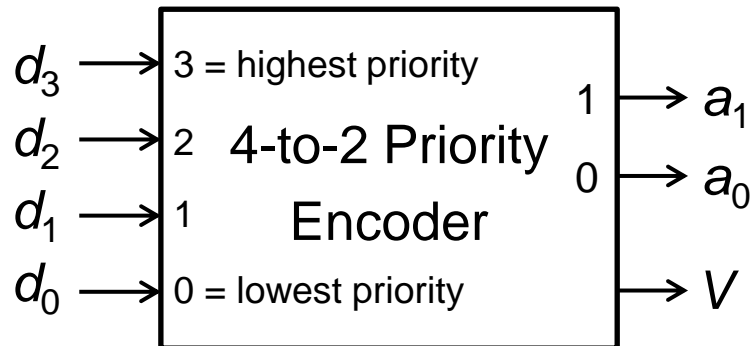
- ❖ Output $a_2 a_1 a_0 = 000$ if $d_0 = 1$ or all inputs are 0's

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

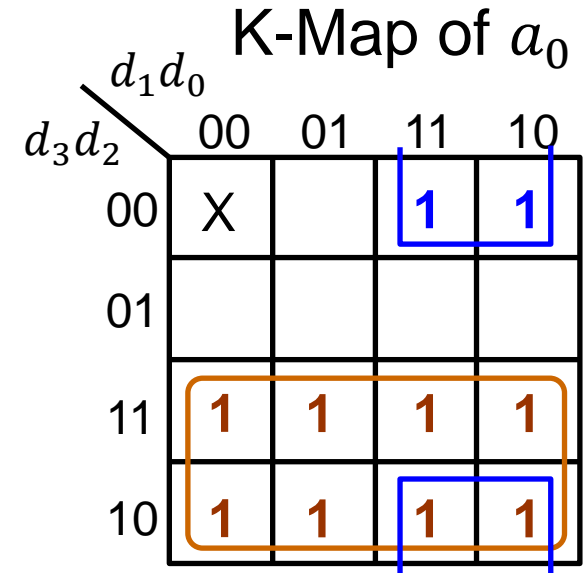
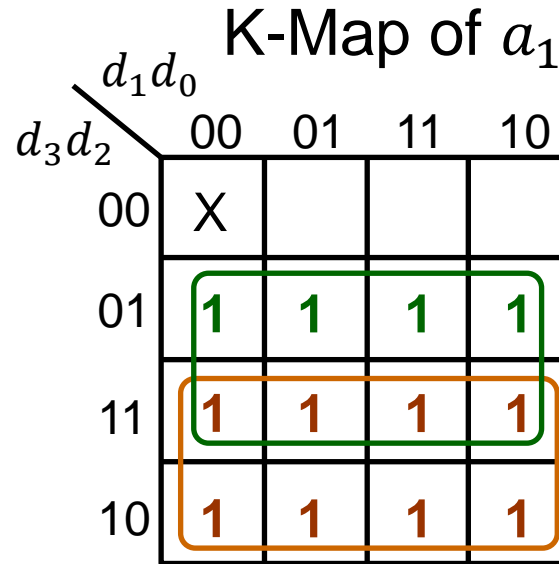


Condensed Truth Table
All 16 cases are listed

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

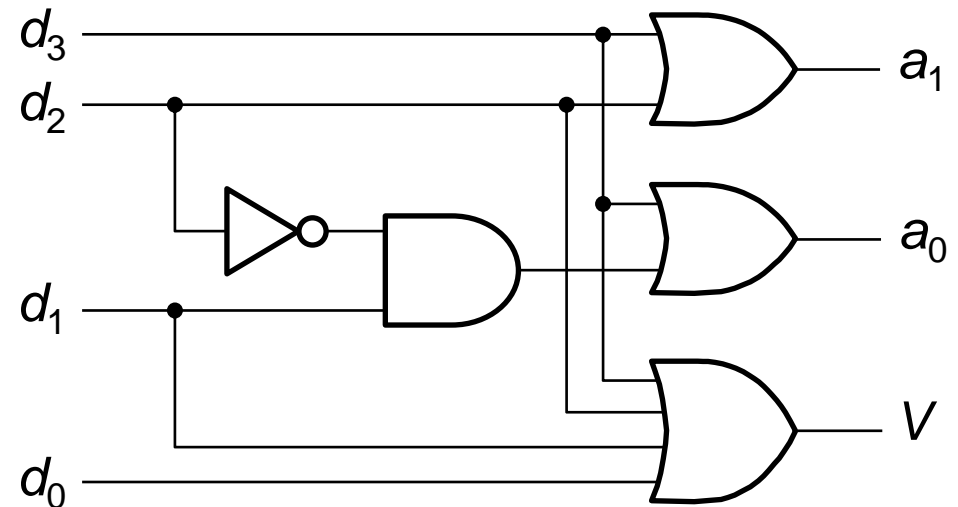


Output Expressions:

$$a_1 = d_3 + d_2$$

$$a_0 = d_3 + d_1 d_2'$$

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



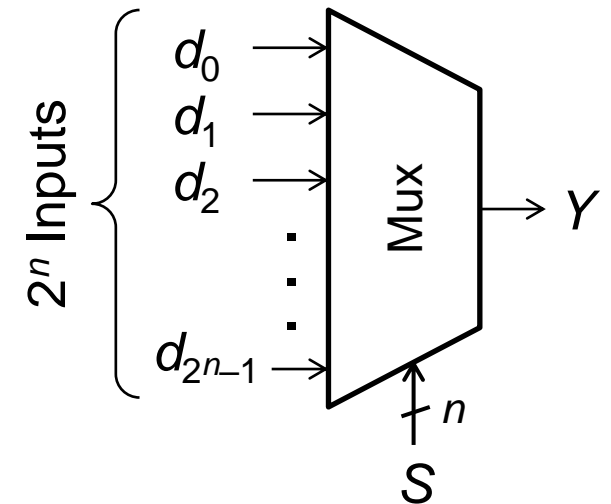
Next . . .

- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Demultiplexers
- ❖ 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

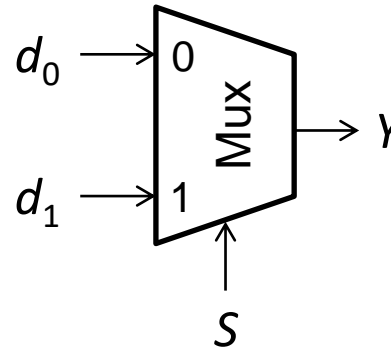
❖ 2-to-1 Multiplexer

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

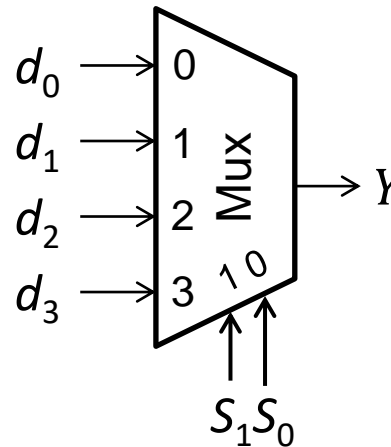
else if ($S_1 S_0 == 01$) $Y = d_1$;

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

else $Y = d_3$;

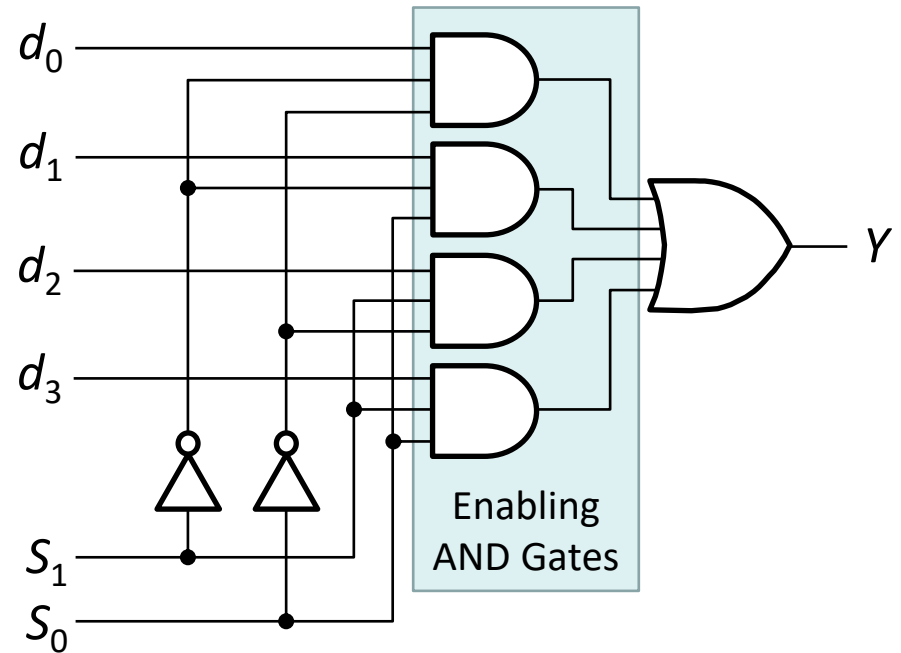
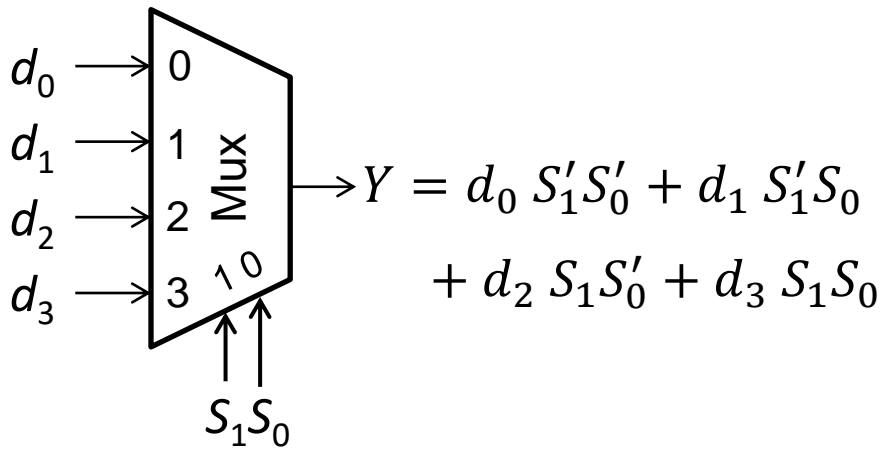
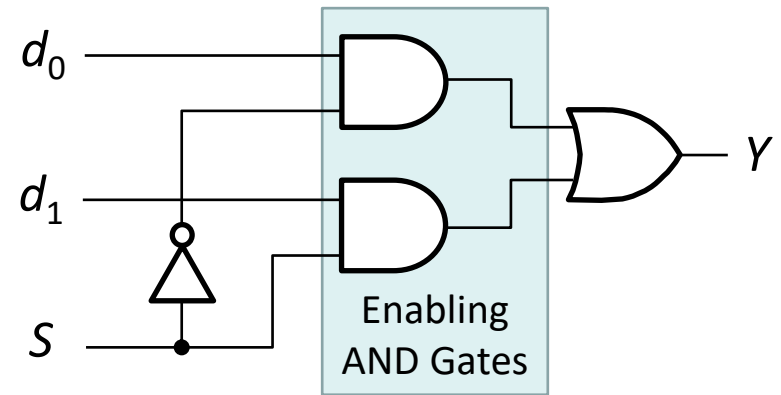
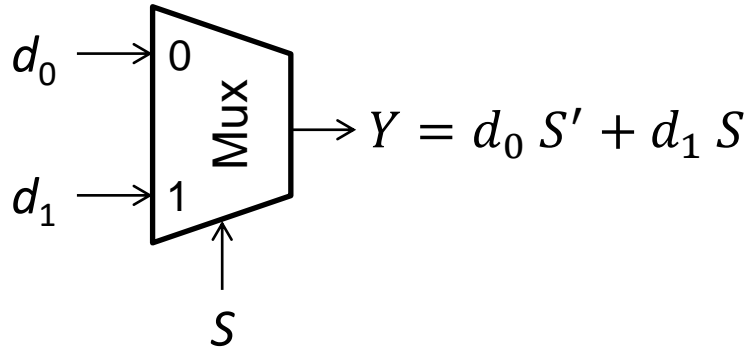
Logic expression:

$$Y = d_0 S_1' S_0' + d_1 S_1' S_0 + d_2 S_1 S_0' + d_3 S_1 S_0$$



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

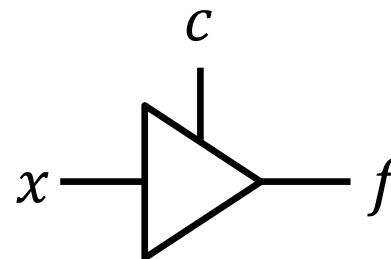
- ❖ Logic gates studied so far have two outputs: 0 and 1
- ❖ Three-State gate has three possible outputs: **0, 1, Z**
 - ✧ **Z** is the **Hi-Impedance** output
 - ✧ **Z** means that the output is **disconnected** from the input
 - ✧ Gate behaves as an **open switch** between input and output

❖ 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**

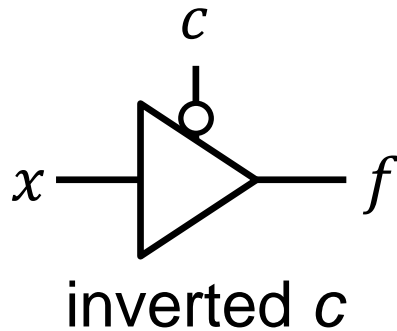


3-state gate

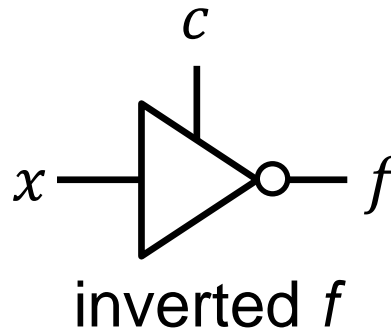
c	x	f
0	0	Z
0	1	Z
1	0	0
1	1	1

Variations of the 3-State Gate

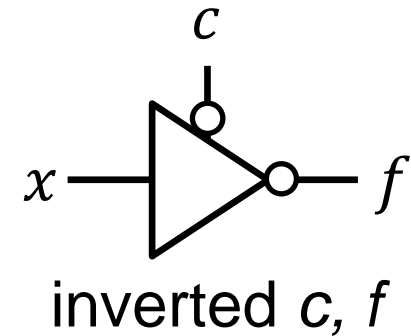
- ❖ Control input c and output f can be inverted
- ❖ A bubble is inserted at the input c or output f



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



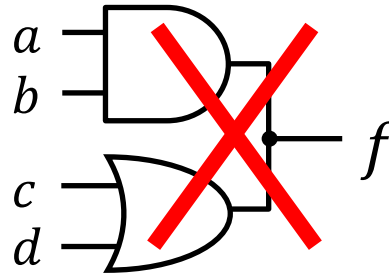
c	x	f
0	0	Z
0	1	Z
1	0	1
1	1	0



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

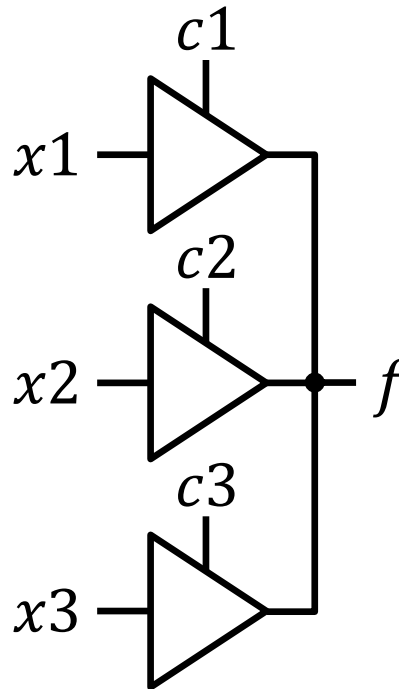
Wired Output

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



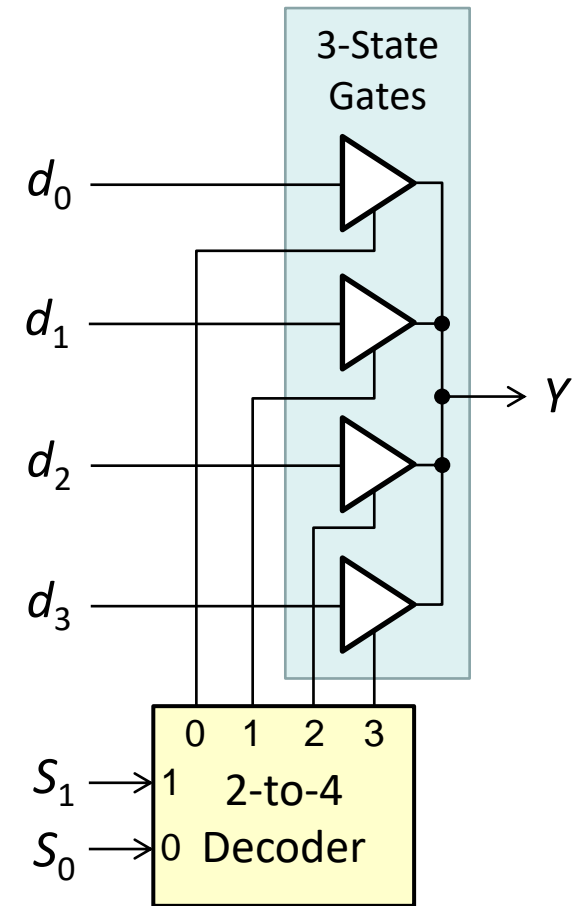
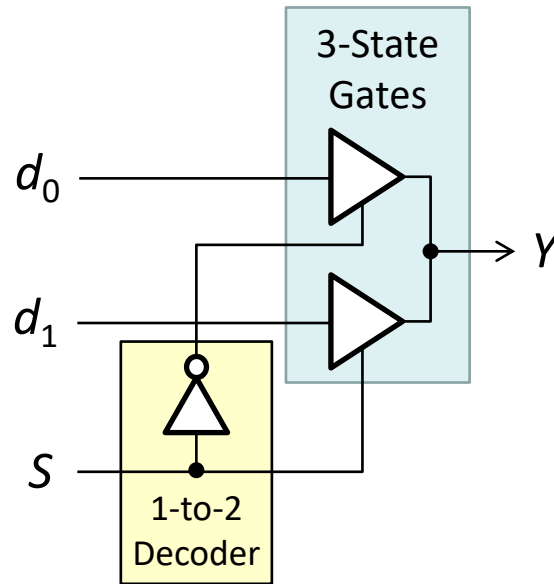
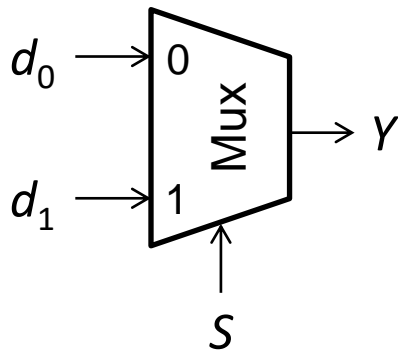
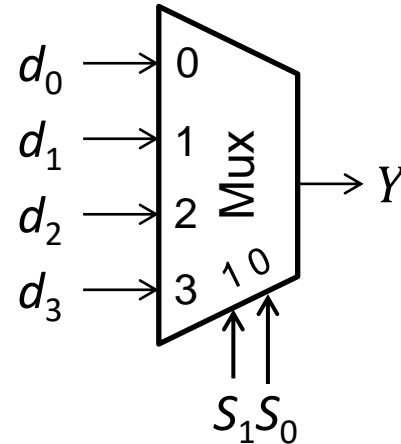
At most one 3-state gate can be enabled at a time
Otherwise, conflicting outputs will burn the circuit

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

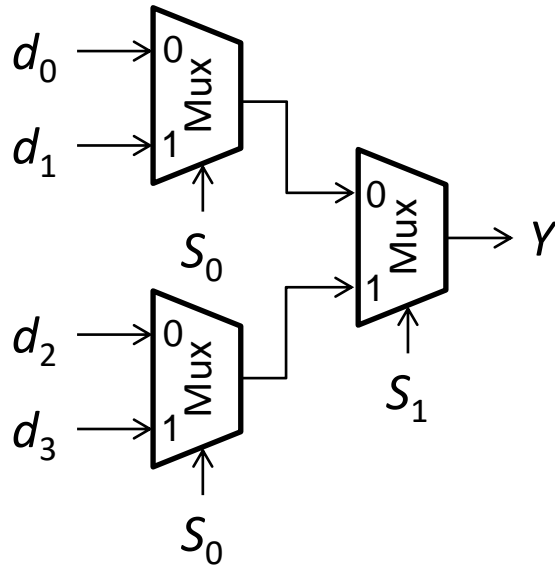
A Multiplexer can also be implemented using:

1. A decoder
2. Three-state gates

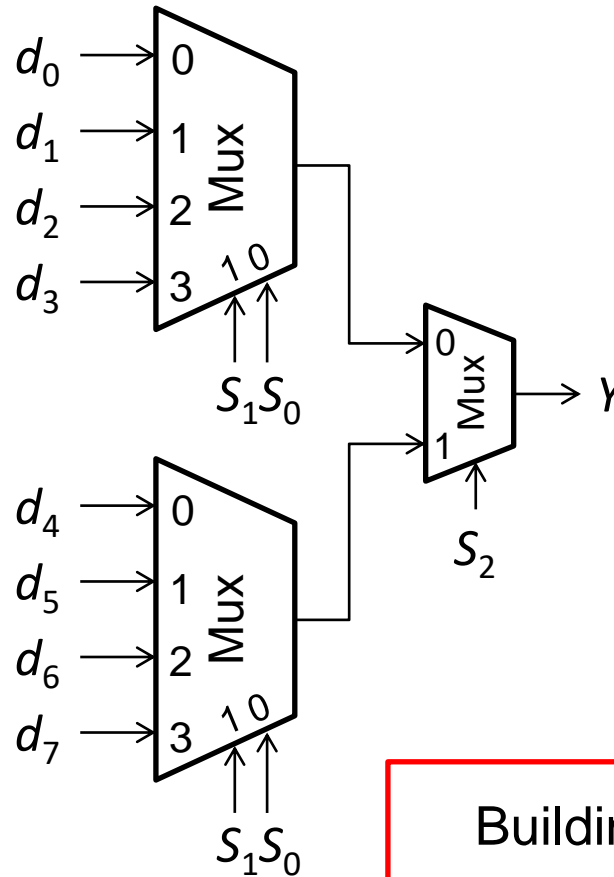


Building Larger Multiplexers

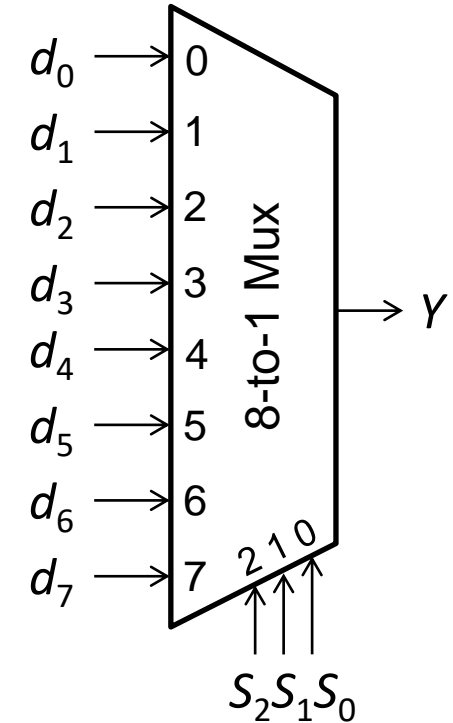
Larger multiplexers can be built hierarchically using smaller ones



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

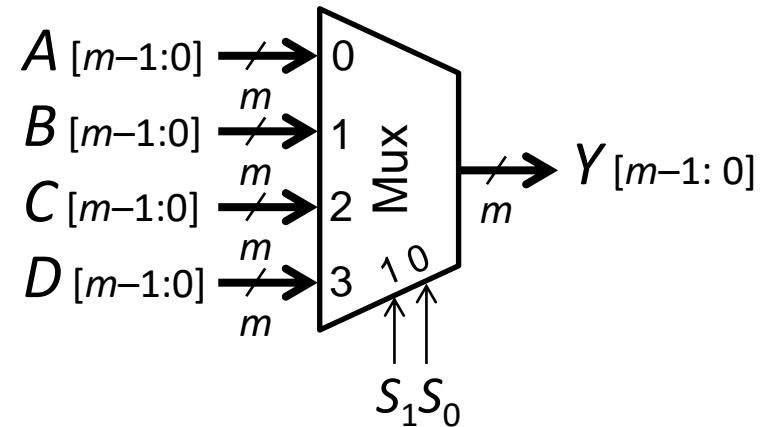
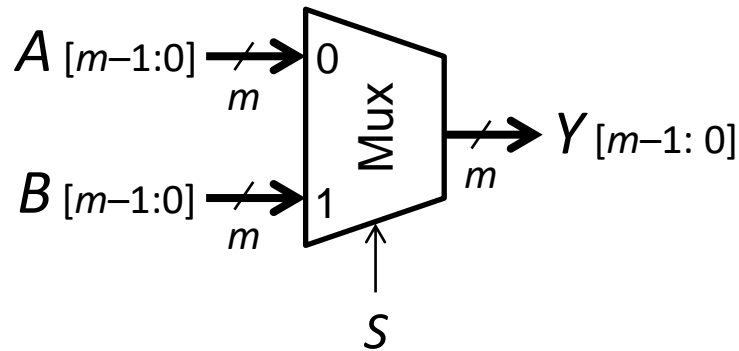


Building 8-to-1 Mux
using two 4-to-1 Muxes
and a 2-to-1 Mux



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

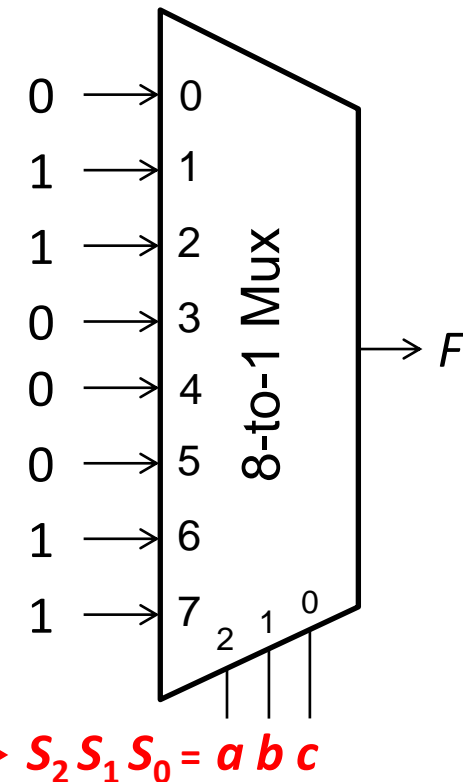
- ❖ A Multiplexer can be used to implement any logic function
- ❖ The function must be expressed using its minterms
- ❖ Example: Implement $F(a, b, c) = \Sigma(1, 2, 6, 7)$ using a Mux

❖ Solution:

The inputs are used as select lines to a Mux.

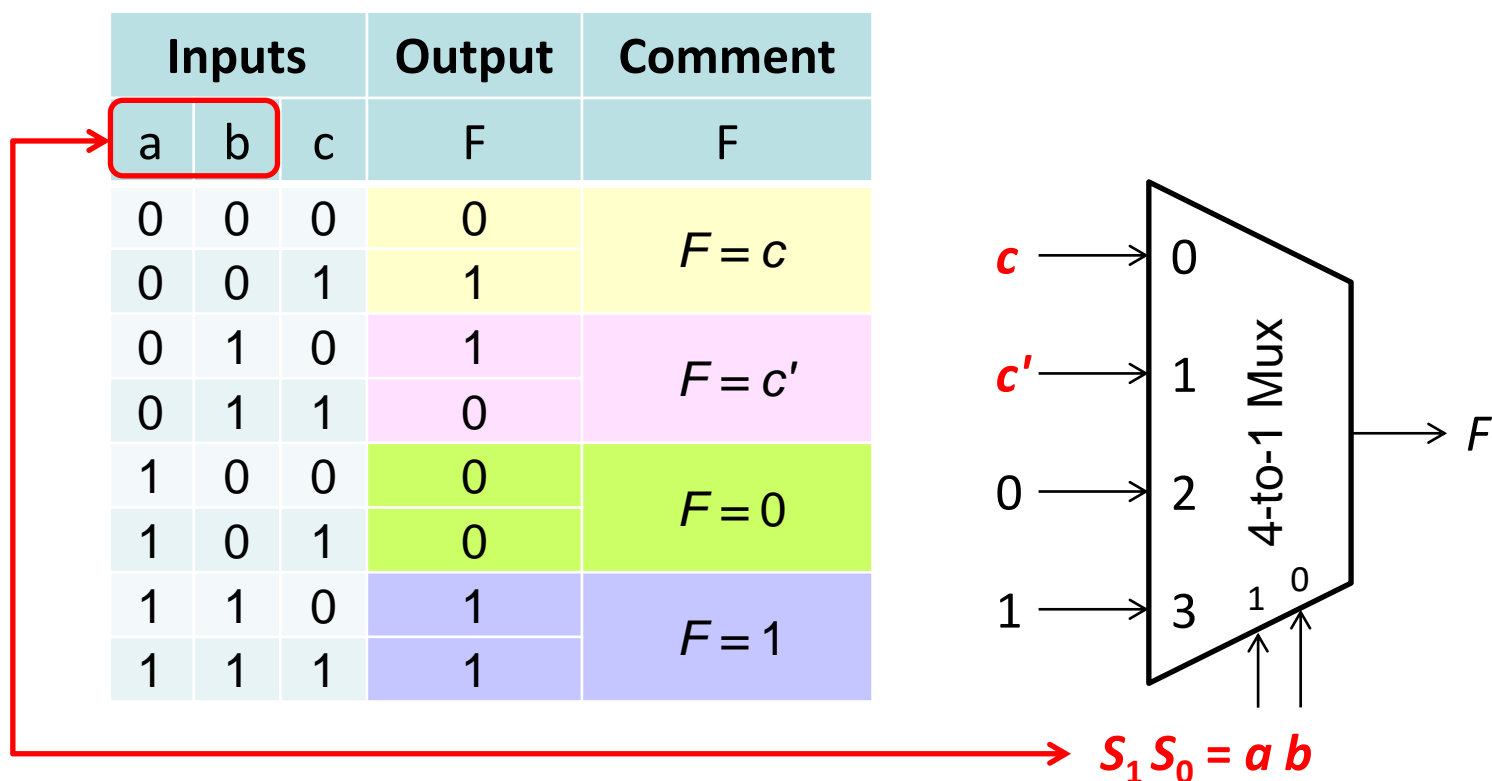
An 8-to-1 Mux is used because there are 3 variables

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



Better Solution with a Smaller Multiplexer

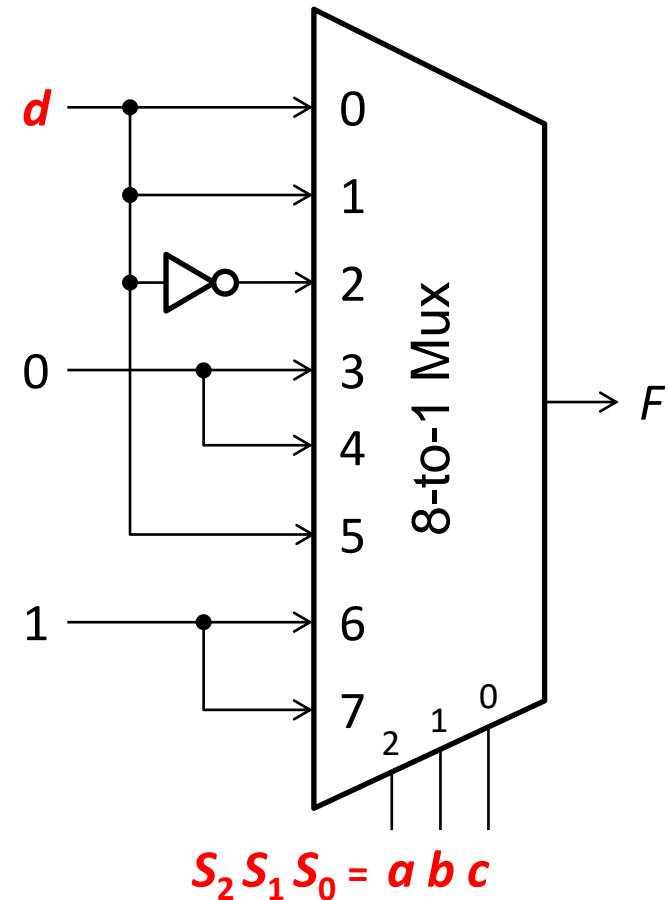
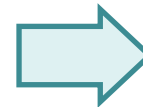
- ❖ Re-implement $F(a, b, c) = \Sigma(1, 2, 6, 7)$ using a 4-to-1 Mux
- ❖ We will use the two select lines for variables a and b
- ❖ Variable c and its complement are used as inputs to the Mux



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
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>F</i>	<i>F</i>
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	



Shannon's Expansion

- ❖ Shannon's Expansion Theorem:

$$f(x_1, \dots, x_i, \dots, x_n) = x_i' \cdot f_{x_i'} + x_i \cdot f_{x_i}$$

$$f_{x_i'} = f(x_1, \dots, x_i = 0, \dots, x_n) \text{ and } f_{x_i} = f(x_1, \dots, x_i = 1, \dots, x_n)$$

- ❖ Example: $f(a, b, c) = ab + ac + bc$

$$f_{a'} = f(0, b, c) = 0 + 0 + bc = bc$$

$$f_a = f(1, b, c) = b + c + bc = b + c \quad (\text{absorption})$$

$$\text{Therefore, } f(a, b, c) = a'(bc) + a(b + c)$$

- ❖ Shannon's Expansion can be applied recursively:

$$f(a, b, c) = a'(b'(0) + b(c)) + a(b'(c) + b(1))$$

$$f(a, b, c) = a'b'(0) + a'b(c) + ab'(c) + ab(1)$$

Shannon's Expansion (cont'd)

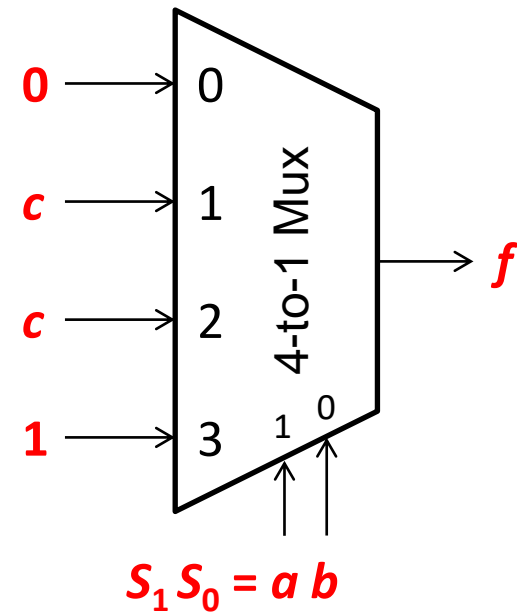
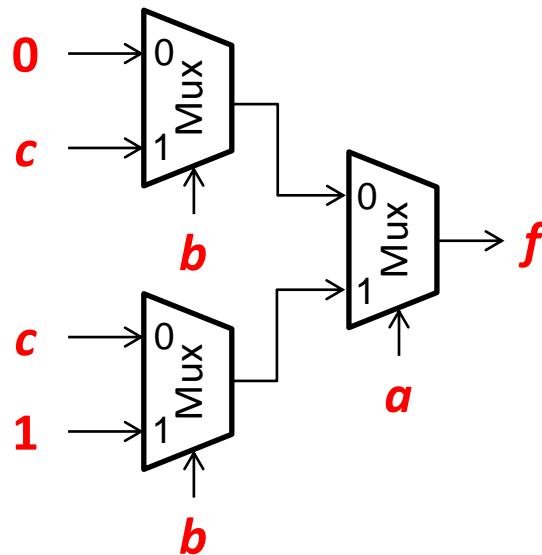
- ❖ Used to implement functions with multiplexers
- ❖ Example: $f(a, b, c) = ab + ac + bc = a'(bc) + a(b + c)$

$$f(a, b, c) = a'(b'(0) + b(c)) + a(b'(c) + b(1))$$

$$f(a, b, c) = a'b'(0) + a'b(c) + ab'(c) + ab(1)$$

4×1 mux

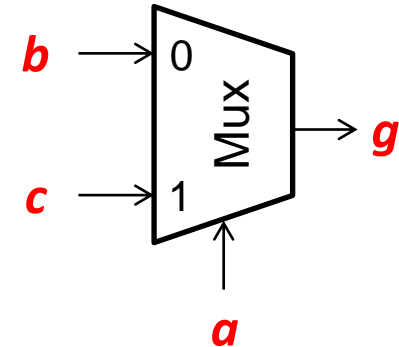
Three 2×1 muxes



Shannon's Expansion (cont'd)

❖ $g(a, b, c) = a'b + a c$

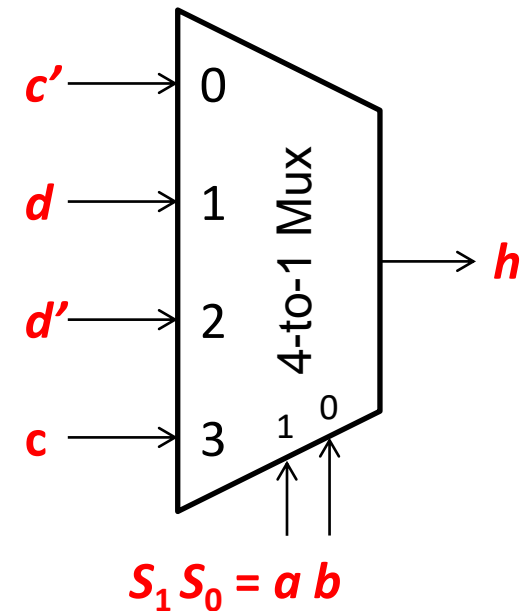
Can be implemented using a 2×1 mux



The truth table method uses a 4×1 mux

❖ $h(a, b, c, d) = a'b'c' + a'bd + ab'd' + abc$

Can be implemented using a 4×1 mux

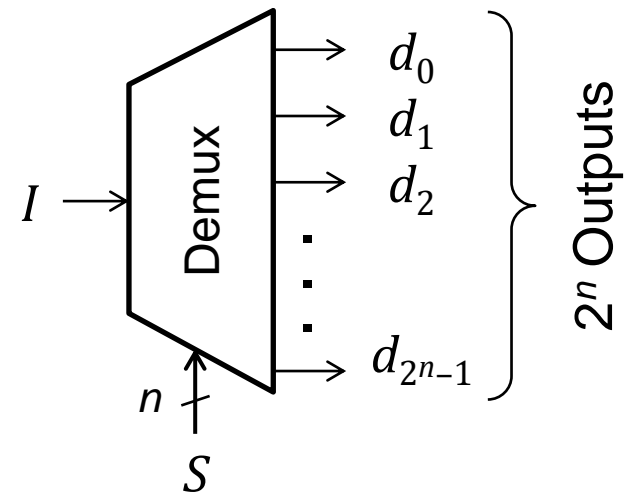


The truth table method uses an 8×1 mux

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

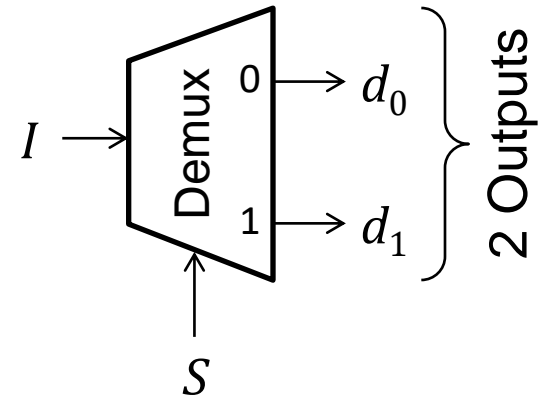
❖ 1-to-2 Demultiplexer

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

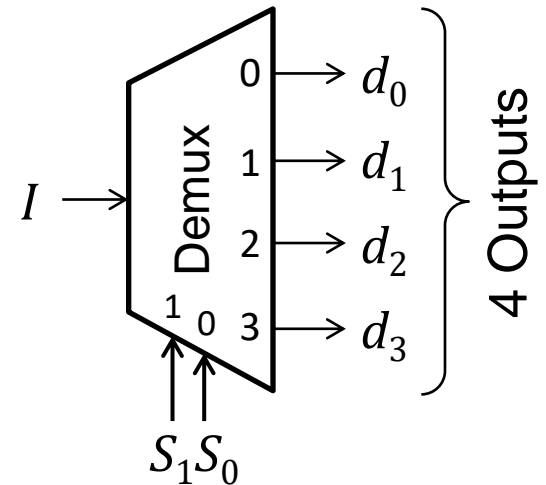
else if ($S_1 S_0 == 01$) { $d_1 = I$; $d_0 = d_2 = d_3 = 0$; }

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

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

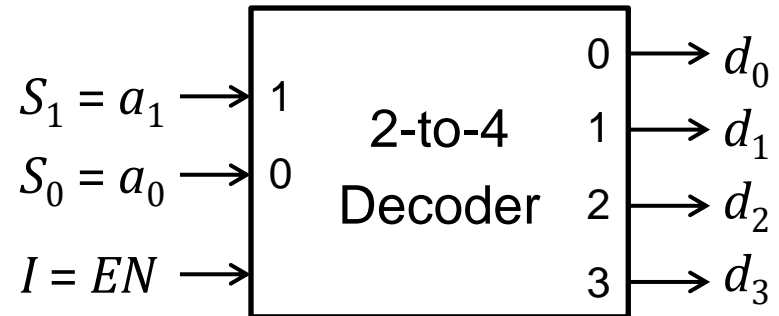
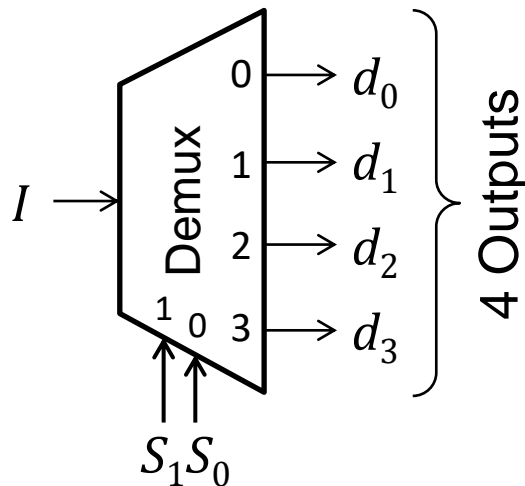
Output expressions:

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



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 . . .

- ❖ Decoders
- ❖ Encoders
- ❖ Multiplexers
- ❖ Demultiplexers
- ❖ 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

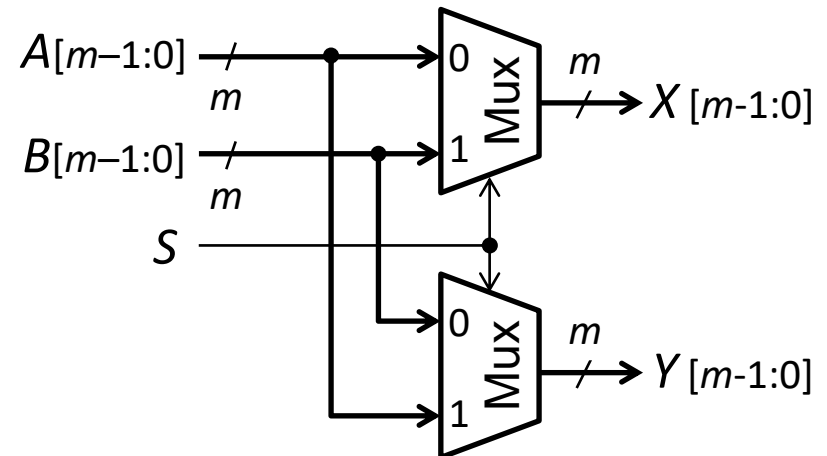
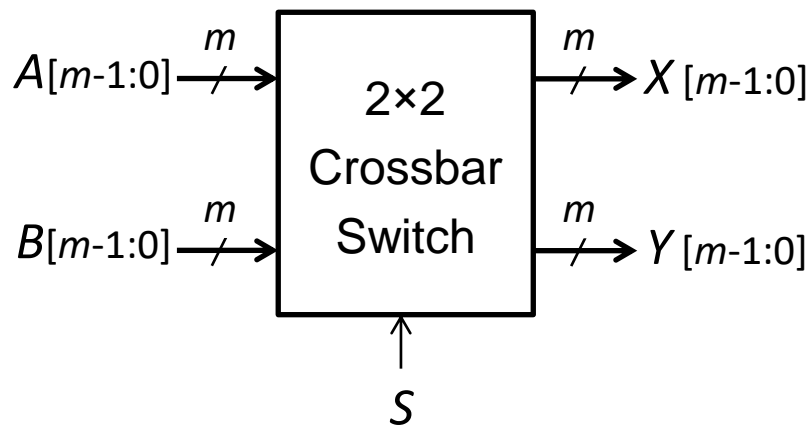
Two m -bit outputs: X and Y

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

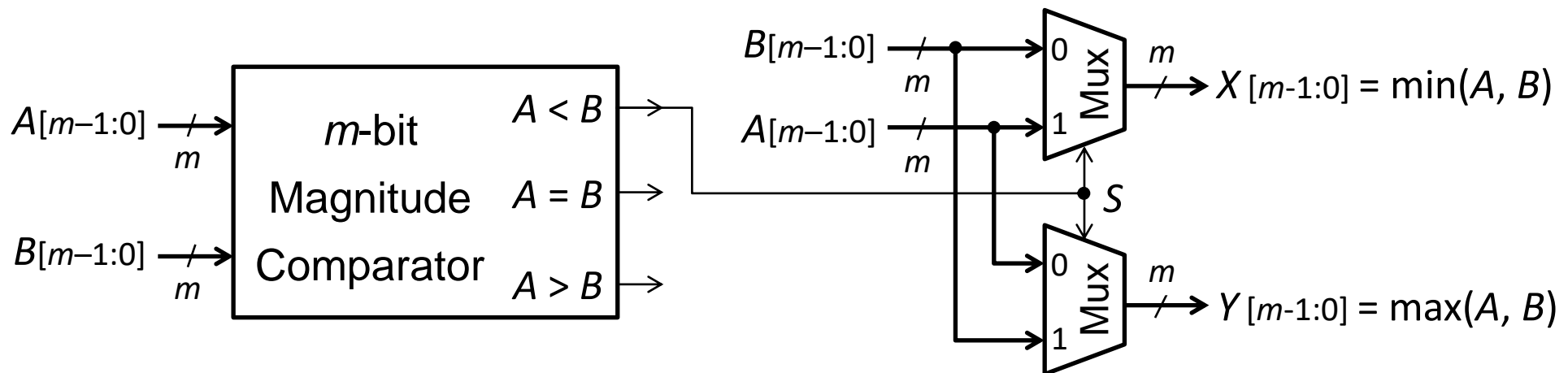
- ❖ Design a circuit that sorts two m -bit unsigned integers A and B

Inputs: Two m -bit unsigned integers A and B

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

- ❖ **Solution:**

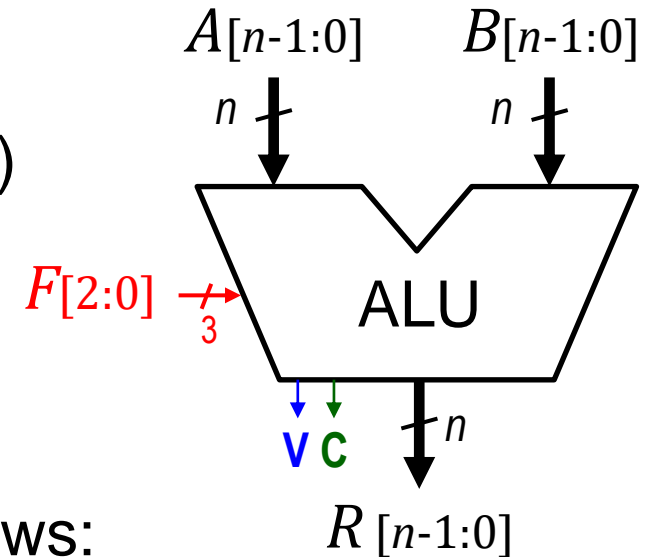
We will use a magnitude comparator to compare A with B , and 2×2 crossbar switch implemented using two 2-input multiplexers



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

