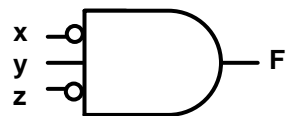


# Unit 9

## Fundamental Digital Building Blocks: Decoders & Multiplexers

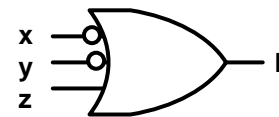
# Checkers / Decoders

- Recall
  - AND gates output '1' for only a single combination
  - OR gates output '0' for only a single combination
  - Inputs (inverted or non-inverted) determine which combination is checked for
  - We say that gate is "checking for" or "decoding" a specific combination



AND gate decoding  
 (checking for)  
 combination 010

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



OR gate decoding  
 (checking for)  
 combination 110

X	Y	Z	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

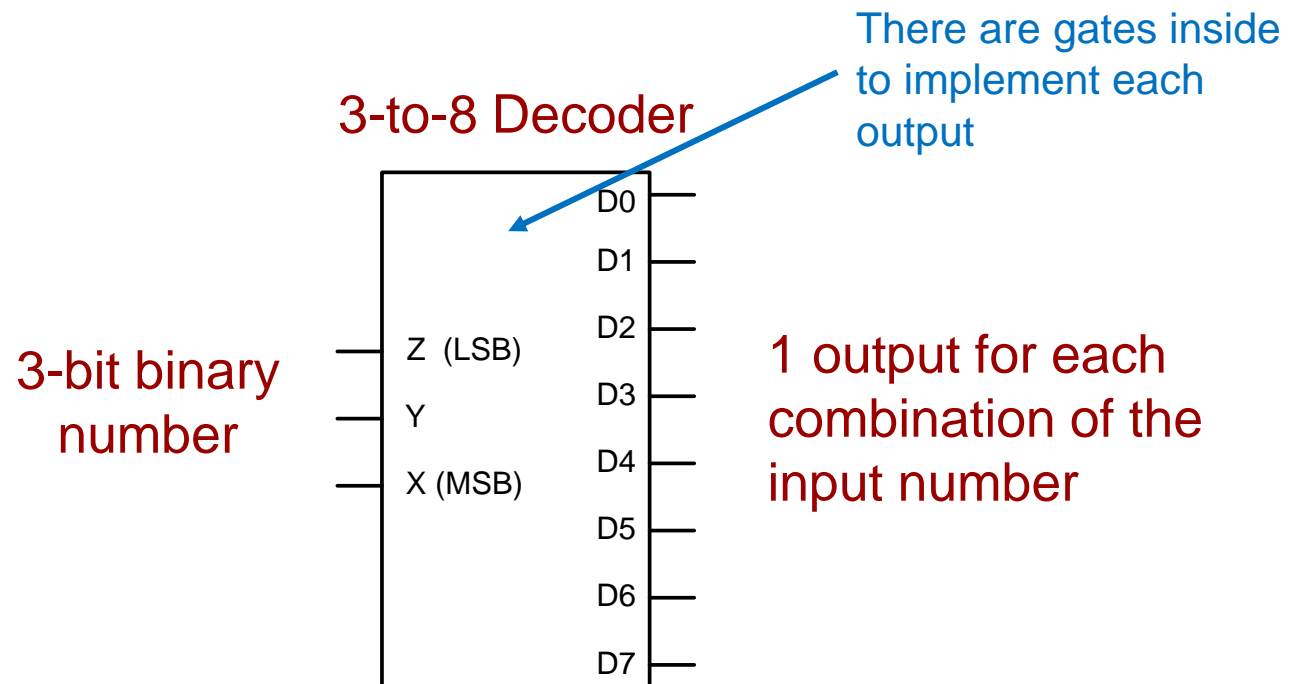
# Motivation

- Just like there are patterns and structures that occur commonly in nature, there are several common logic structures that occur over and over again in digital circuits
  - Decoders, Multiplexers, Adders, Registers
- In addition, we design hardware using a **hierarchical** approach
  - We design a **small component** using **basic logic gates** (e.g. a 1-bit mux)
  - We build a **large component** by interconnecting many copies of the **small component** + a few extra gates (e.g. a 32-bit mux)
  - We build **chips** by interconnecting many **large components** (e.g. a router)
  - Each components is truly made out of many gates but the design process is faster and easier by using hierarchy
- Let's look at a few common components
  - We'll start by describing the behavior of the component and then determine what gates are inside

# DECODERS

# Decoders

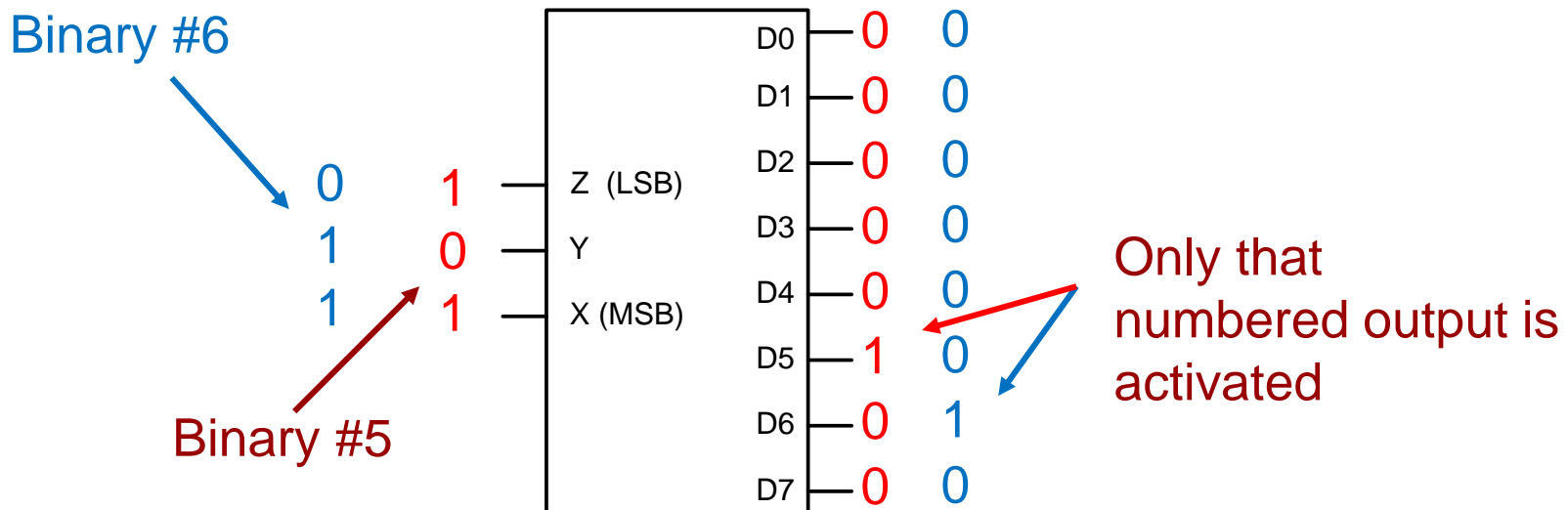
- A decoder is a building block that:
  - Takes in an n-bit binary number as input
  - Decodes that binary number and activates the corresponding output
  - Individual outputs for \_\_\_\_\_ input combinations



# Decoders

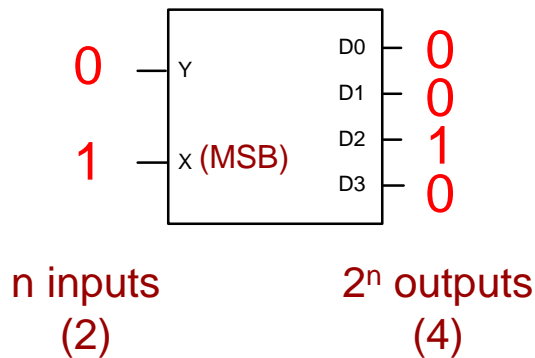
- A decoder is a building block that:
  - Takes a binary number as input
  - Decodes that binary number and activates the corresponding output
  - Put in 6=110, Output 6 activates ('1')
  - Put in 5=101, Output 5 activates ('1')

X	Y	Z	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>
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

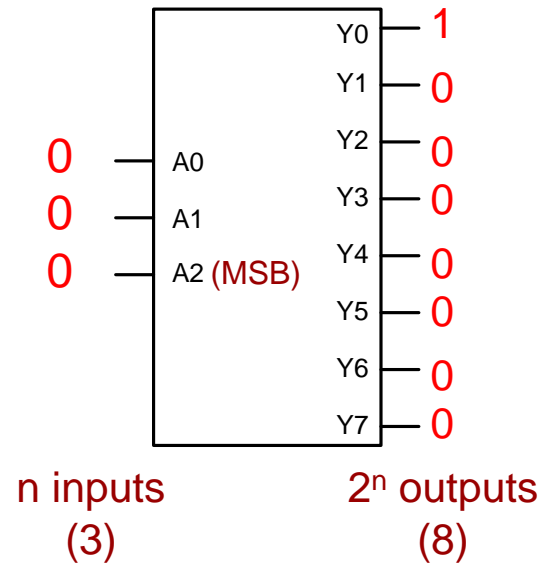


# Decoder Sizes

- A decoder w/ an **n-bit input** has  **$2^n$  outputs**
  - 1 output for every combination of the n-bit input



**2-to-4  
Decoder**

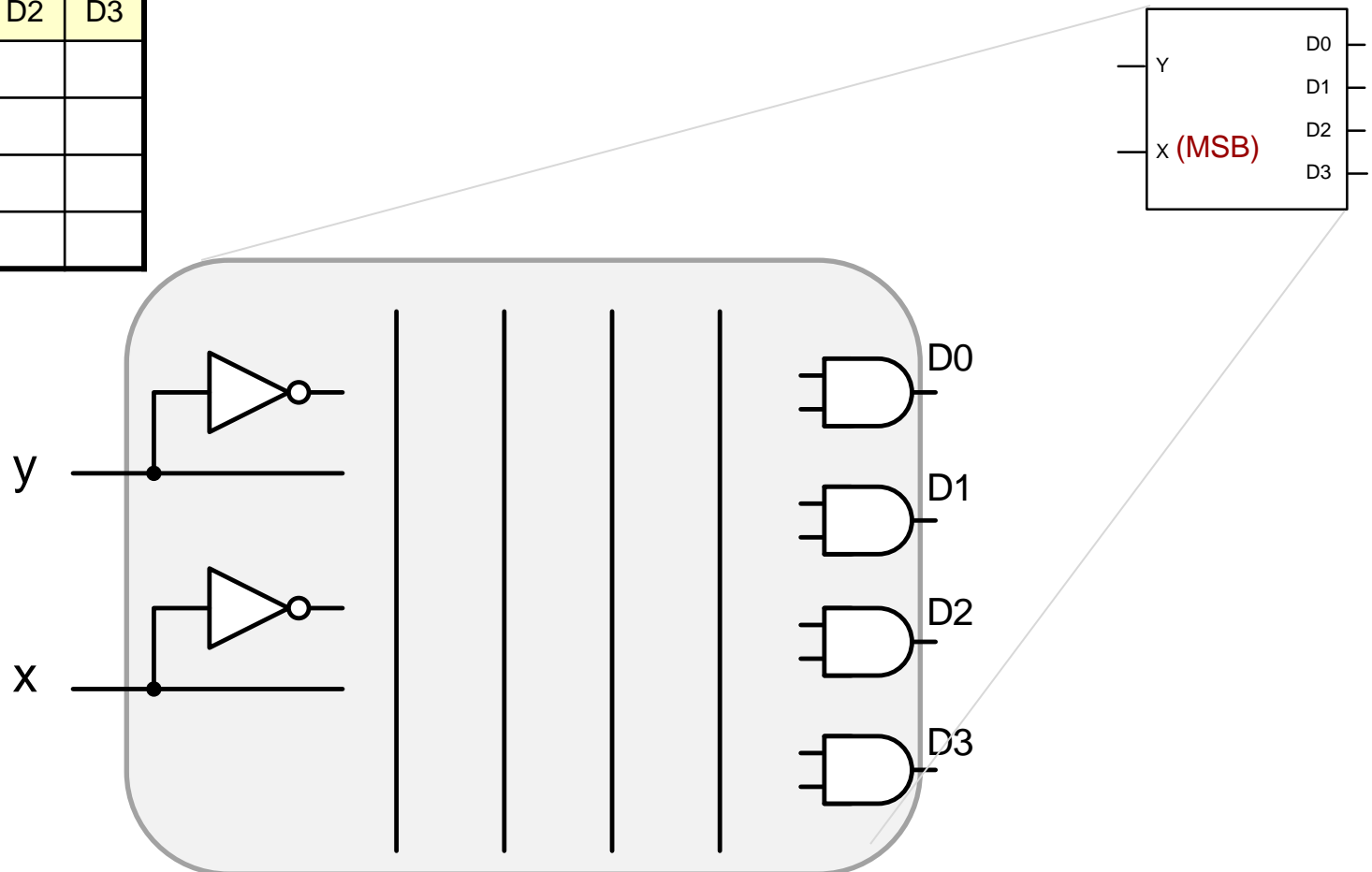


**3-to-8  
Decoder**

# Exercise

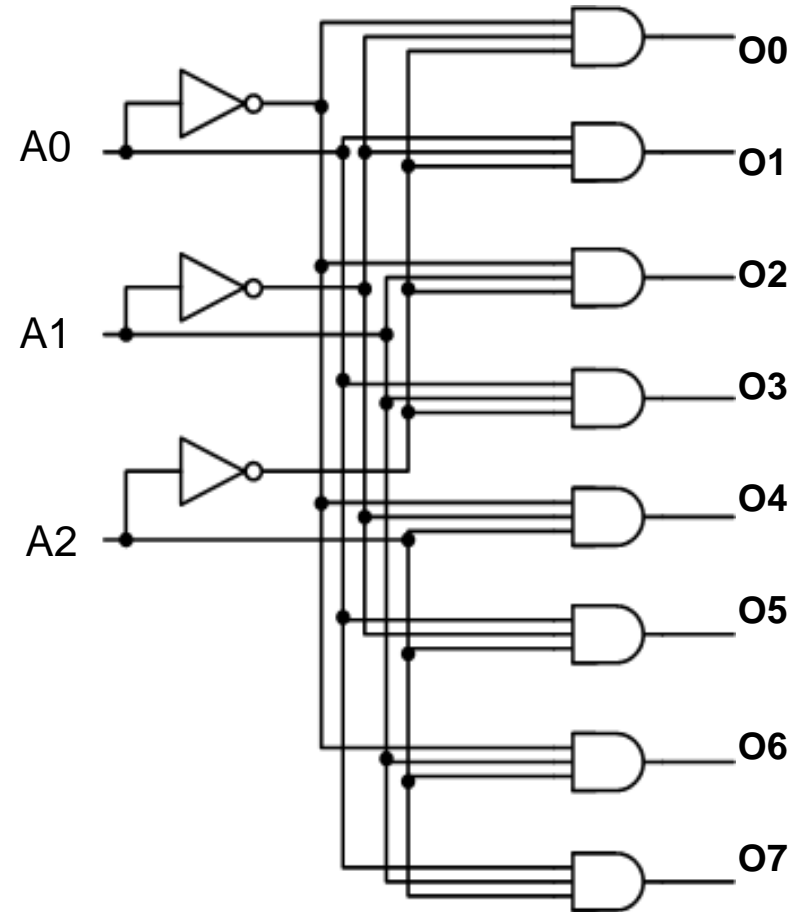
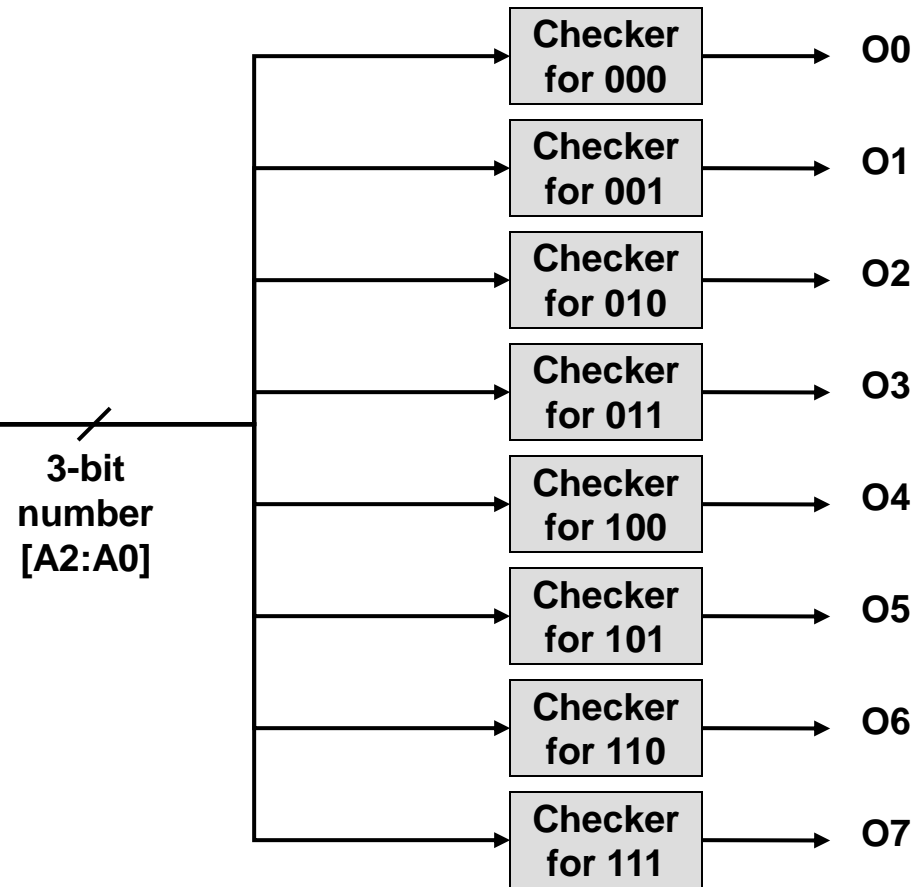
- Complete the design of a 2-to-4 decoder

X	Y	D0	D1	D2	D3
0	0				
0	1				
1	0				
1	1				



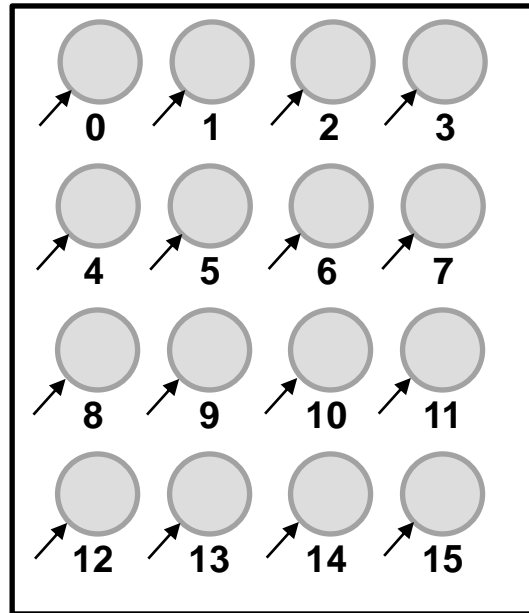
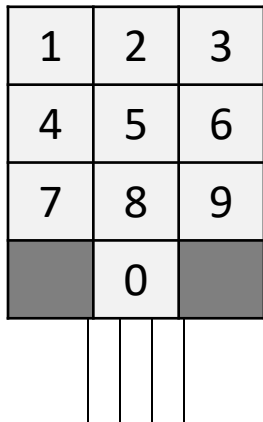


# Building Decoders



# Vending Machine Example

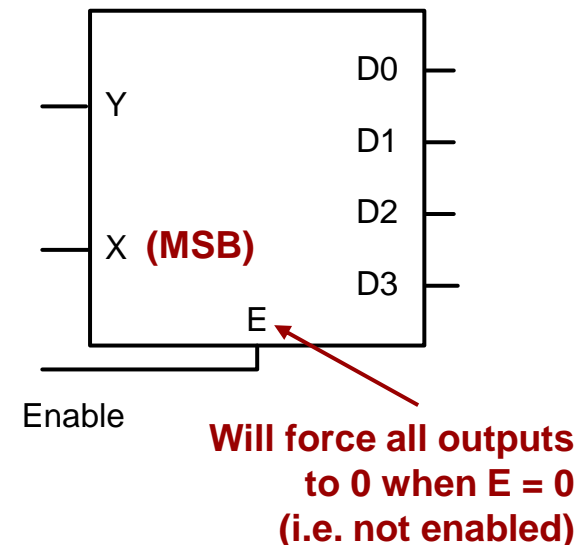
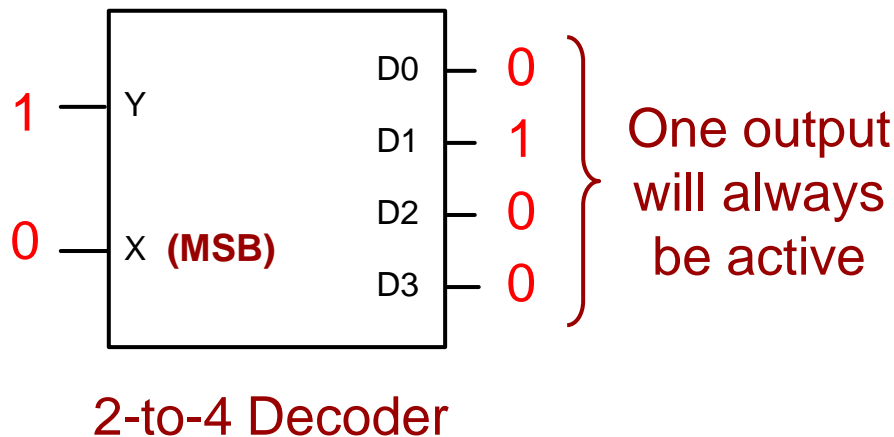
Assuming the keypad produces a 4-bit numeric output, add logic to produce the release signals for each of the 16 vending items.



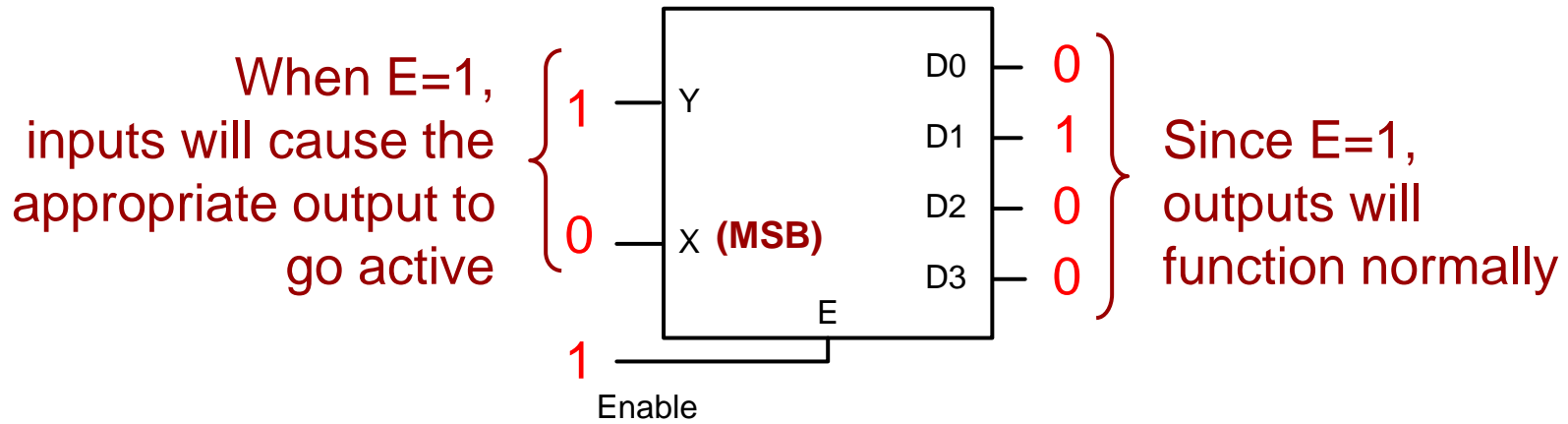
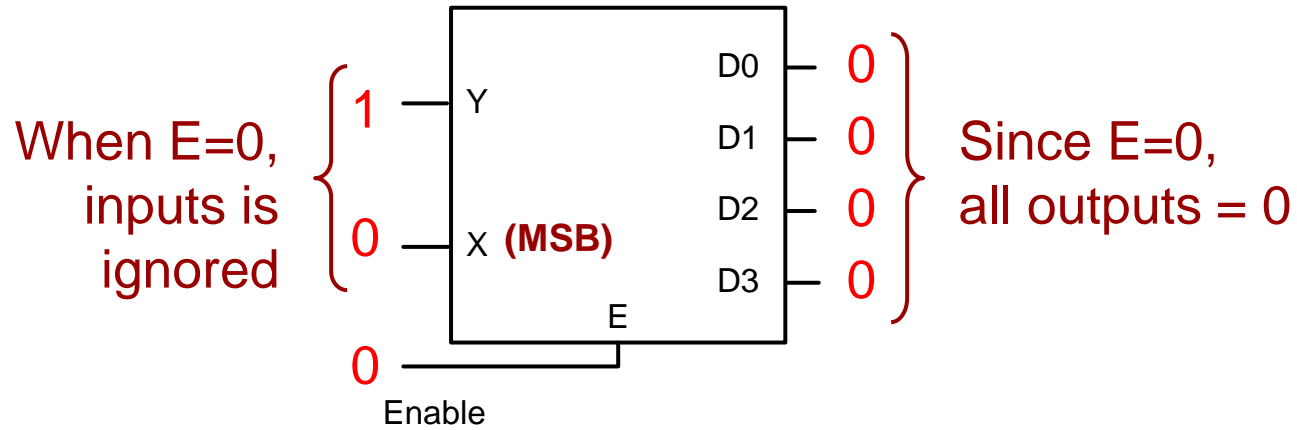
Consider any problems with this design.

# Enables

- In a normal decoder exactly one output is active at all times
- It may be undesirable to always have an active output
- We can add an extra input (called an enable) that can independently force all the outputs to their inactive values

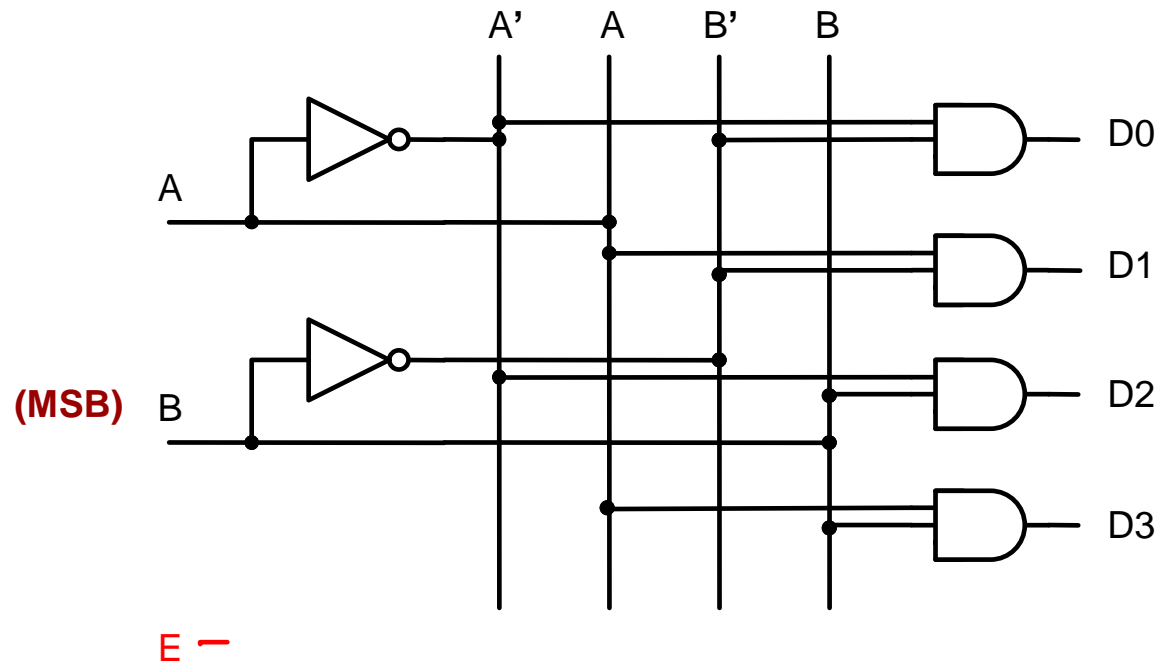


# Enables



# Implementing Enables

- Original 2-to-4 decoder

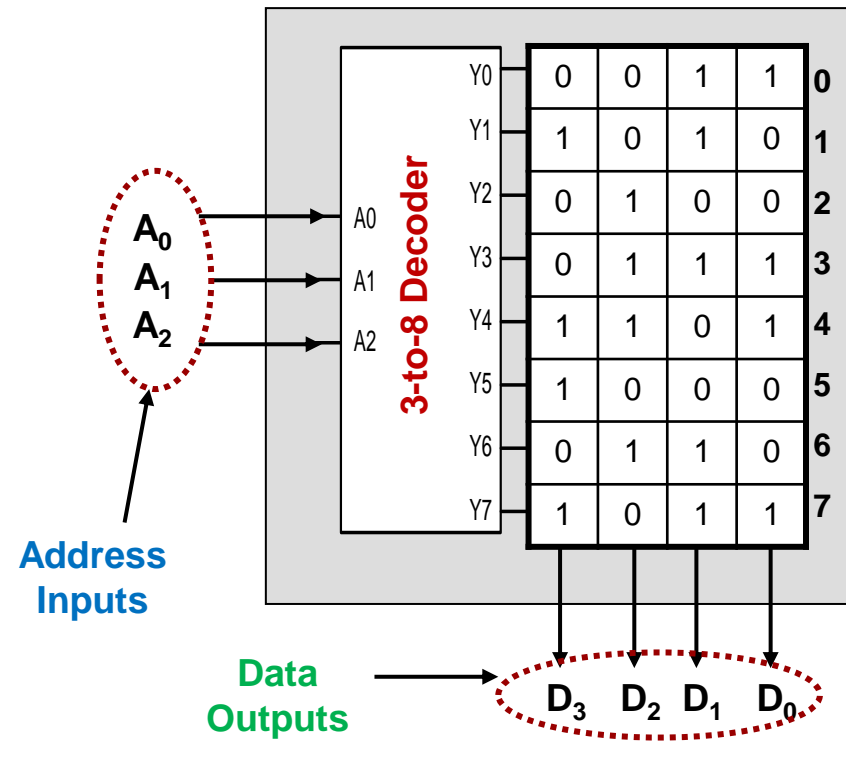
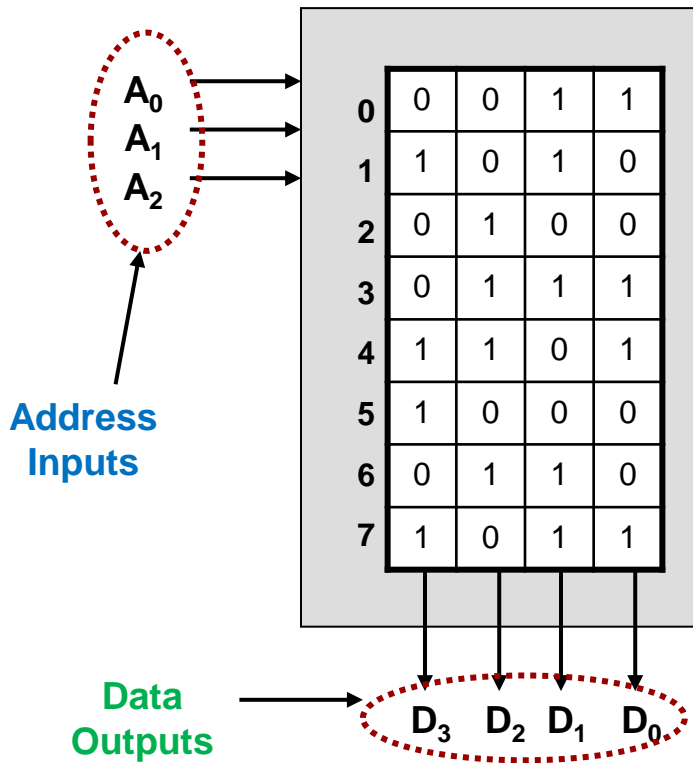


When E=0, force all outputs = 0

When E=1, outputs operate as they did originally

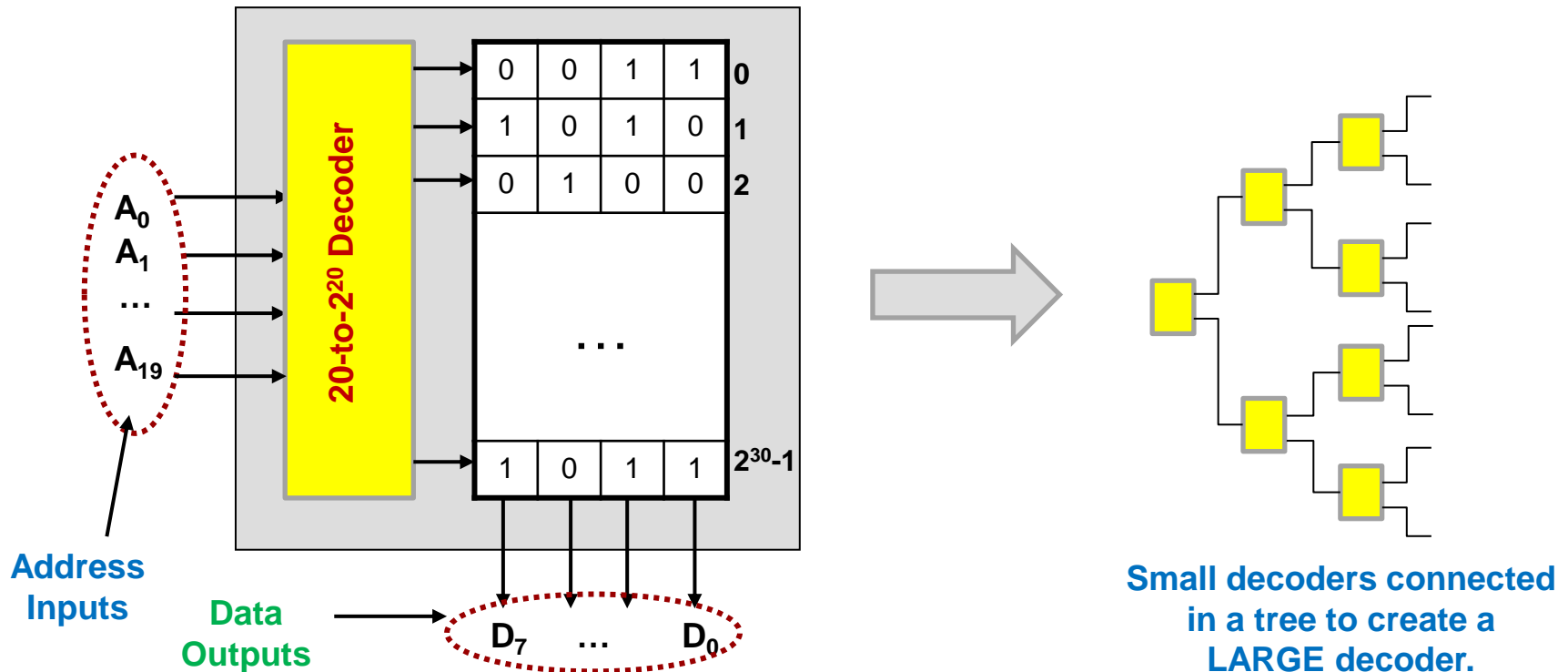
# Another Application of Decoders: Memories

- All memory (RAMs, ROMs) use decoders to select the desired data given an address (each location/byte corresponds to one address combination)
- If you have a 1 MB ( $2^{20}$  bytes) RAM, there is a 20-to- $2^{20}$  decoder present in that device



# Building Large Decoders

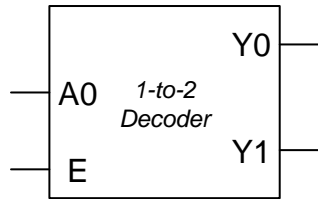
- If you have 1 MB ( $2^{20}$  bytes) RAM, there is a 20-to- $2^{20}$  decoder present in that device
- How can we create such large decoders?
  - Through hierarchy (building-block methodology)..usually of linear chains or tree-based structures



# Larger Decoder Exercise 1

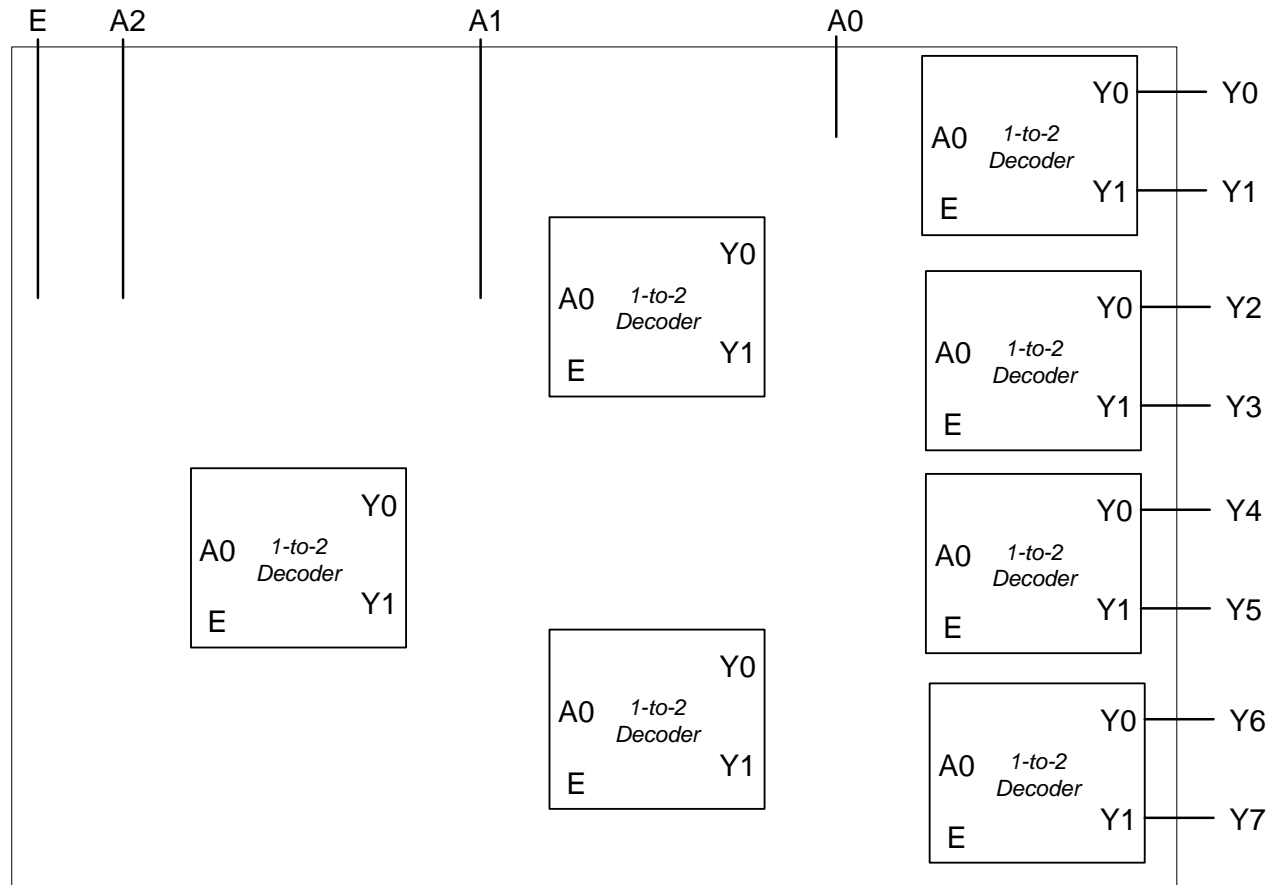
- Build a 3-to-8 decoder from 1-to-2 decoders

## 1-to-2 Decoder Operation



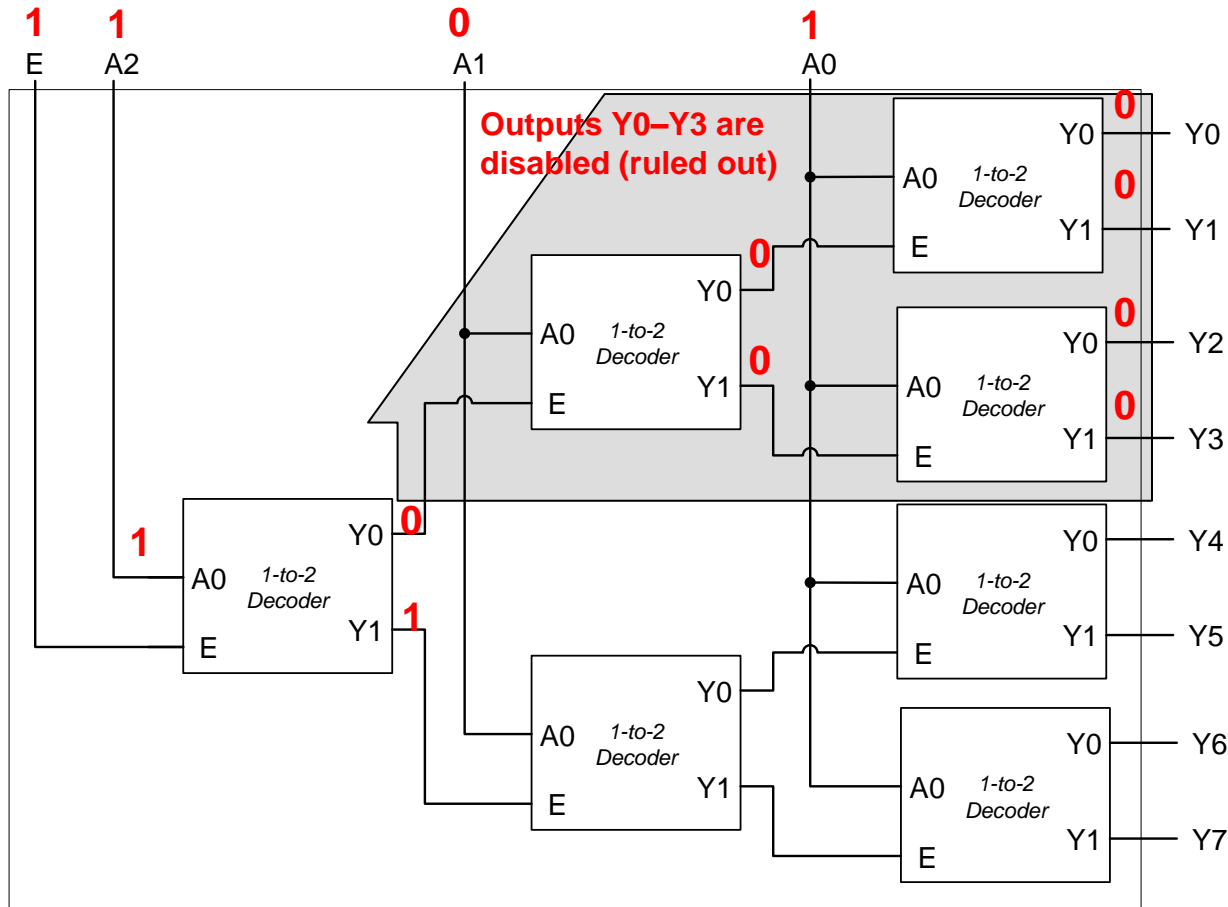
E	A0	Y0	Y1
0	X		
1	0		
1	1		

X = not relevant  
(same result for all possible values of A0)





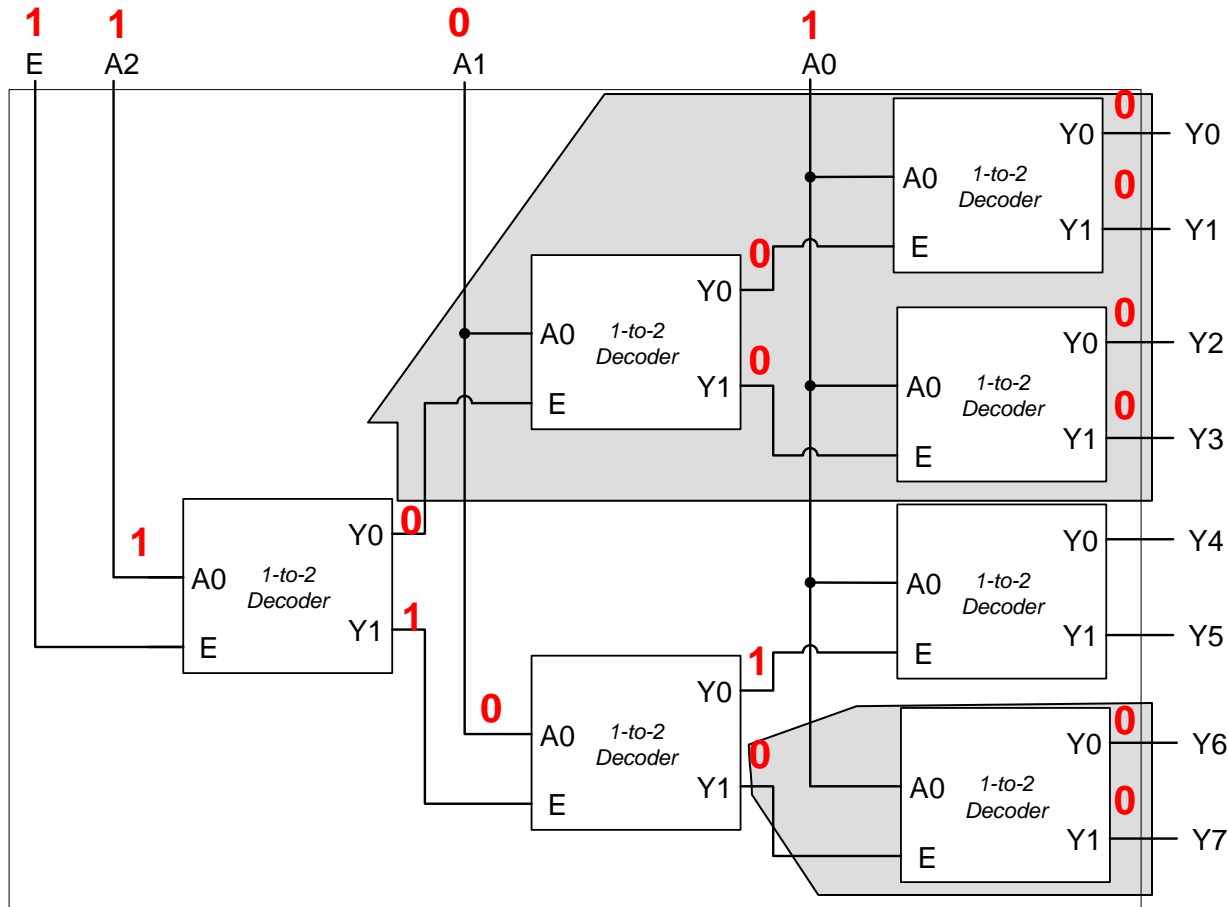
# Larger Decoder Exercise 1a



$A_2$	$A_1$	$A_0$	Active Output
0	0	0	$Y_0$
0	0	1	$Y_1$
0	1	0	$Y_2$
0	1	1	$Y_3$
1	0	0	$Y_4$
	0	1	$Y_5$
	1	0	$Y_6$
	1	1	$Y_7$

Decode the MSB...possible combos = 4-7

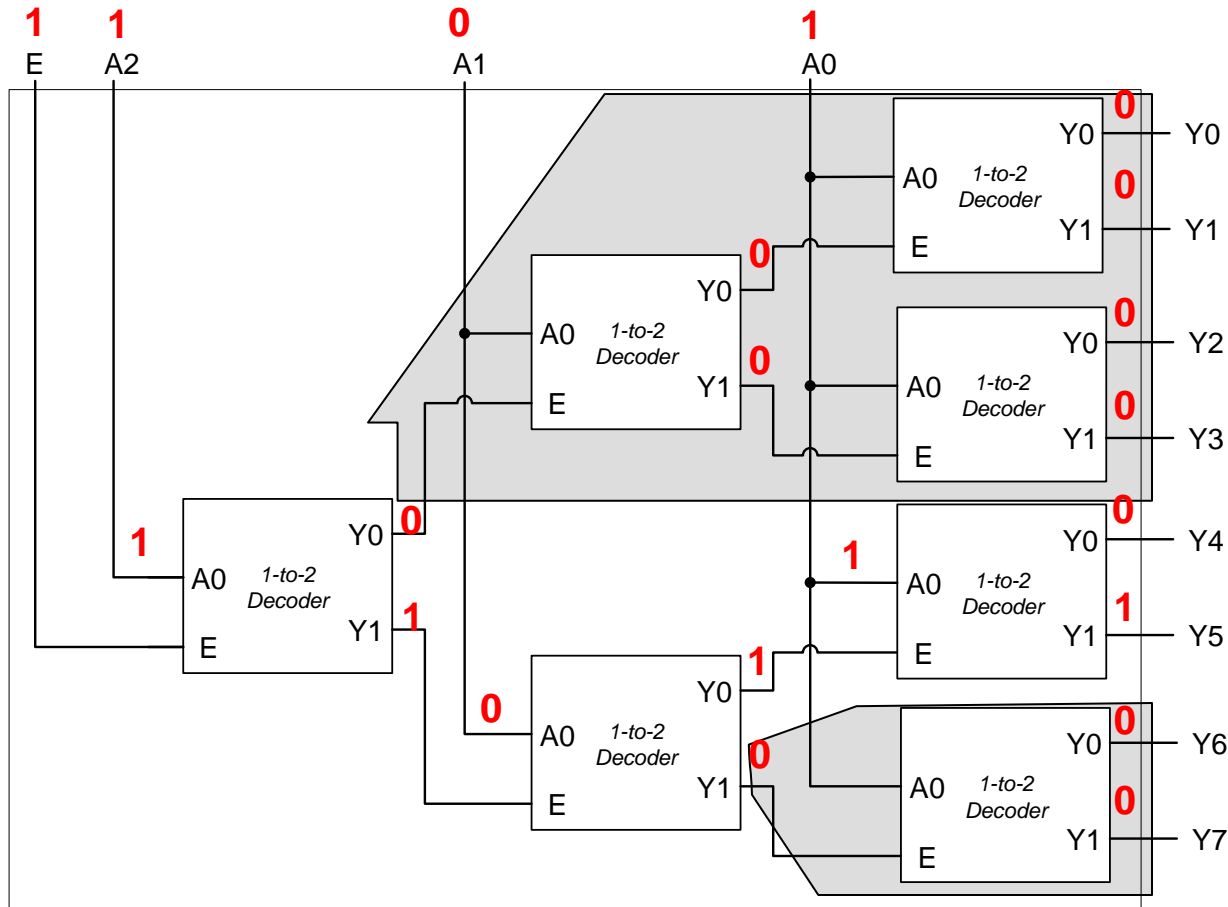
# Larger Decoder Exercise 1b



Decode the  $A_1$  ... possible combos = 5-6

$A_2$	$A_1$	$A_0$	Active Output
0	0	0	$Y_0$
0	0	1	$Y_1$
0	1	0	$Y_2$
0	1	1	$Y_3$
<b>1</b>	<b>0</b>	0	$Y_4$
		1	$Y_5$
	1	0	$Y_6$
	1	1	$Y_7$

# Larger Decoder Exercise 1c



Decode the  $A_0$  ... possible combo = 5

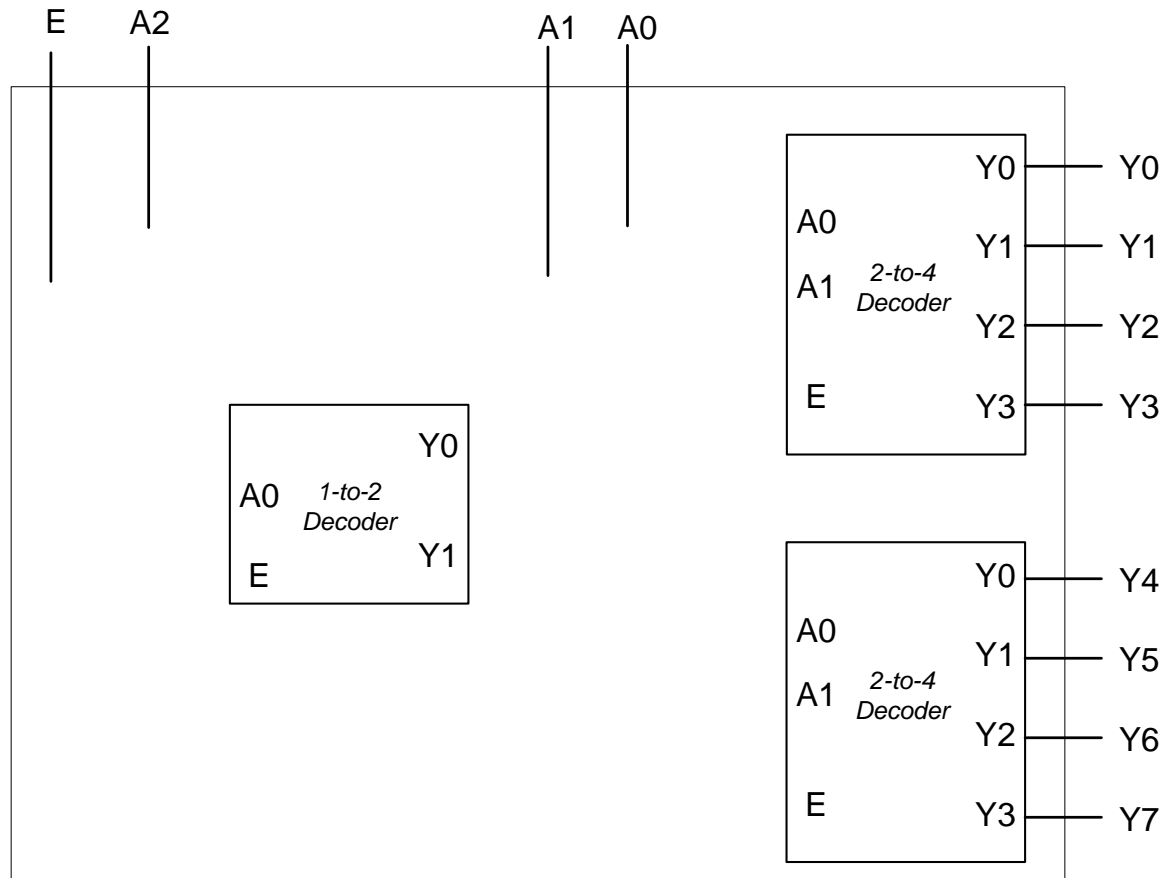
$A_2$	$A_1$	$A_0$	Active Output
0	0	0	$Y_0$
0	0	1	$Y_1$
0	1	0	$Y_2$
0	1	1	$Y_3$
<b>1</b>	<b>0</b>	0	$Y_4$
		<b>1</b>	<b><math>Y_5</math></b>
	1	0	$Y_6$
	1	1	$Y_7$

# General Tree Decoder Approach

- Step 1: Outputs of one stage should connect to the \_\_\_\_\_ of the next stage
- Step 2: All decoders in a stage (level) should decode the same \_\_\_\_\_
  - Usually, the MSB is connected to the first stage and LSB to the last stage

# Larger Decoder Exercise 2

- Different size decoders can be utilized
  - Build a 3-to-8 decoder using 1-to-2 and 2-to-4 decoders

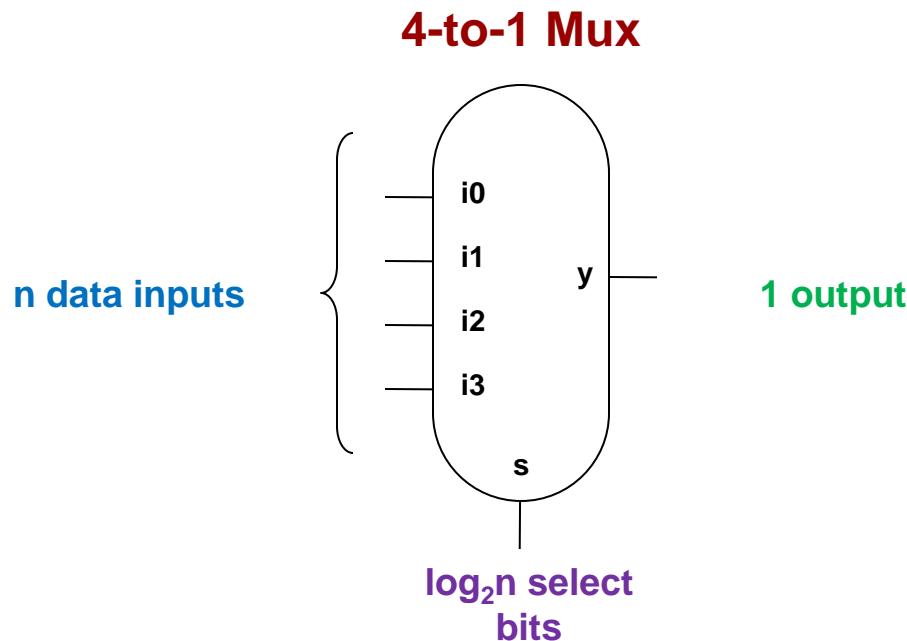


The if..else of digital hardware

# MULTIPLEXERS

# Multiplexers

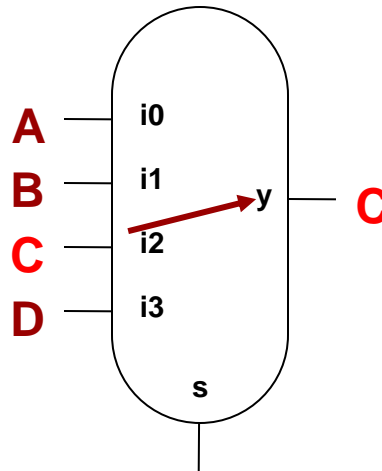
- Multiplexers are one of the most common digital circuits
- Anatomy:  $n$  data inputs,  $\log_2 n$  select bits, 1 output
- A multiplexer (“mux” for short) selects one data input and passes it to the output



$S_1$	$S_0$	Y
0	0	$i_0$
0	1	$i_1$
1	0	$i_2$
1	1	$i_3$

# Multiplexers

4-to-1 Mux



② Thus, input 2 = C is selected and passed to the output

① Select bits =  $10_2 = 2_{10}$ .

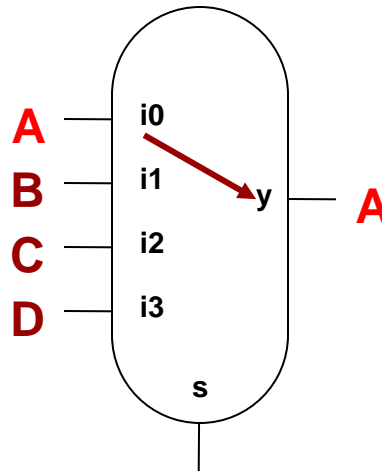
$S_1$	$S_0$	Y
0	0	i0
0	1	i1
1	0	i2
1	1	i3

As long as the select bits are  $10_2 = 2$ , whatever bit value appears on input 2 is copied to the output, same as if we had just wired input 2 directly to the output.



# Multiplexers

## 4-to-1 Mux



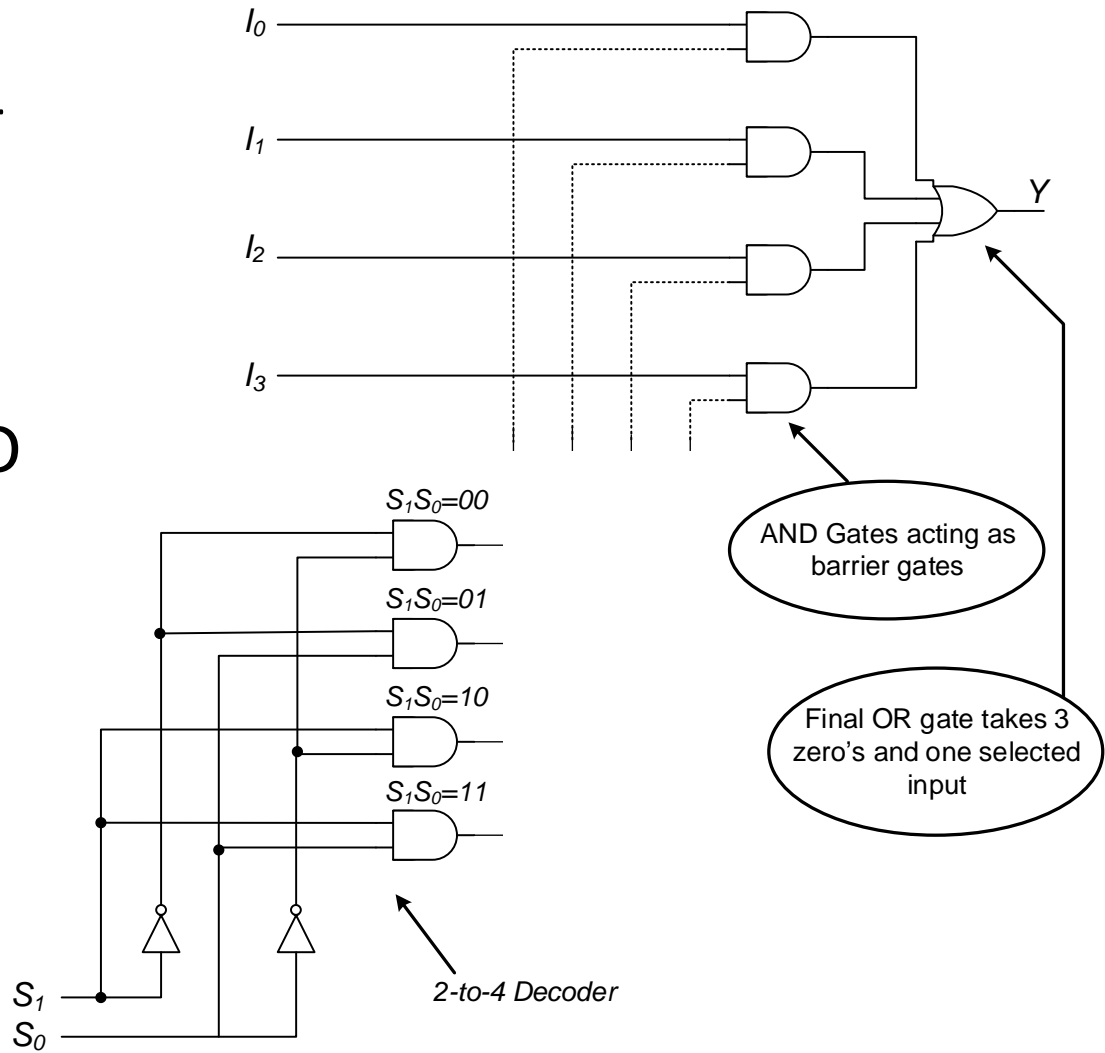
2 Thus, input 0 = A is selected and passed to the output

1 Select bits =  $00_2 = 0_{10}$ .

$S_1$	$S_0$	Y
0	0	i0
0	1	i1
1	0	i2
1	1	i3

# Exercise: Build a 4-to-1 mux

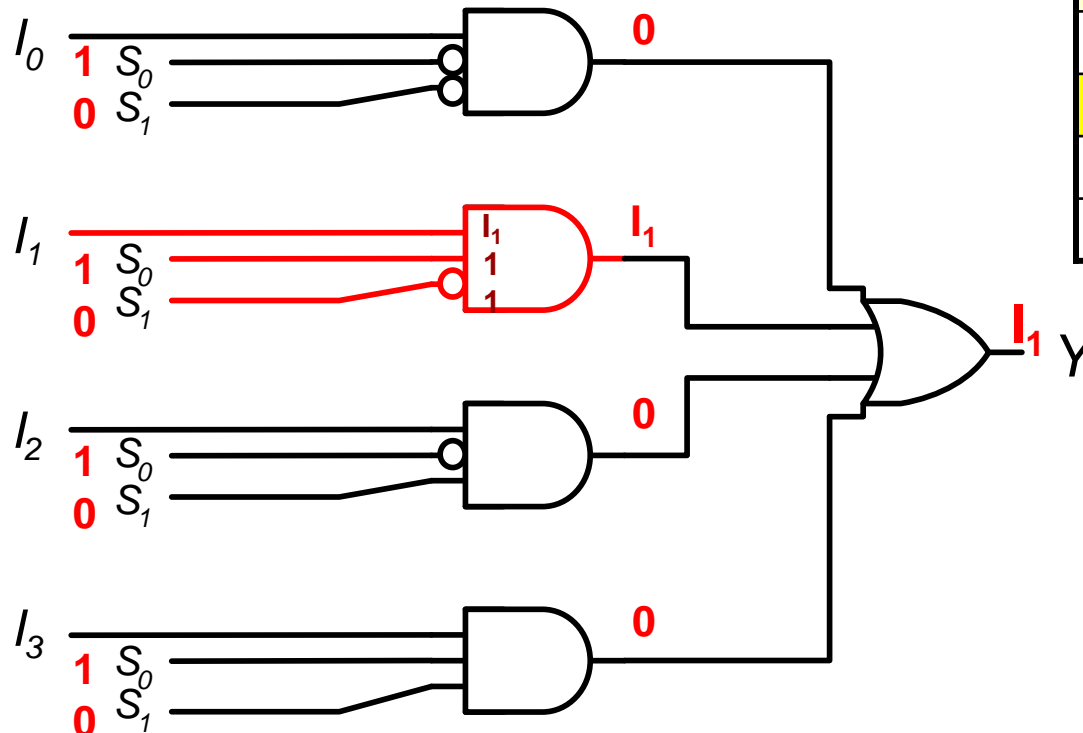
- Complete the 4-to-1 mux to the right by drawing wires between the 2-to-4 decode and the AND gates



# Building a Mux

- To build a mux
  - Decode the select bits and include the corresponding data input.
  - Finally OR all the first level outputs together.

$S_1 S_0 = 01_2$

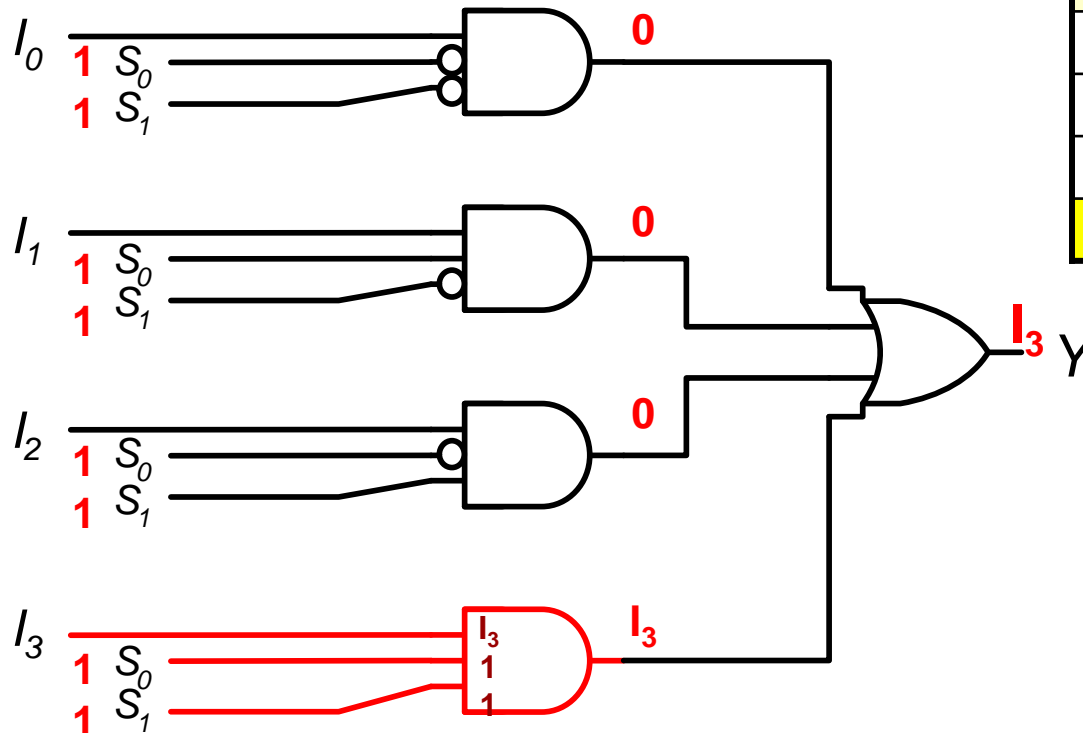


$S_1$	$S_0$	Y
0	0	$i_0$
0	1	$i_1$
1	0	$i_2$
1	1	$i_3$

# Building a Mux

- To build a mux
  - Decode the select bits and include the corresponding data input.
  - Finally OR all the first level outputs together.

$S_1 S_0 = 11_2$

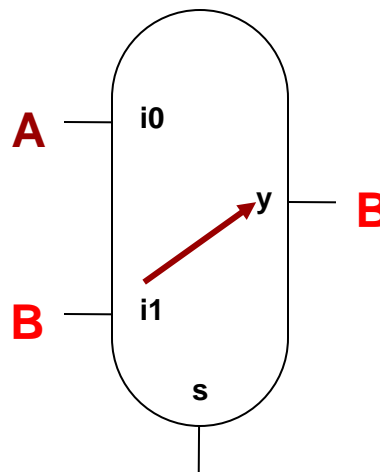


$S_1$	$S_0$	Y
0	0	$i_0$
0	1	$i_1$
1	0	$i_2$
1	1	$i_3$

# 2-to-1 Multiplexers

- We can have design and build muxes with any number of inputs (2-to-1, 5-to-1, 16-to-1, etc.)

## 2-to-1 Mux



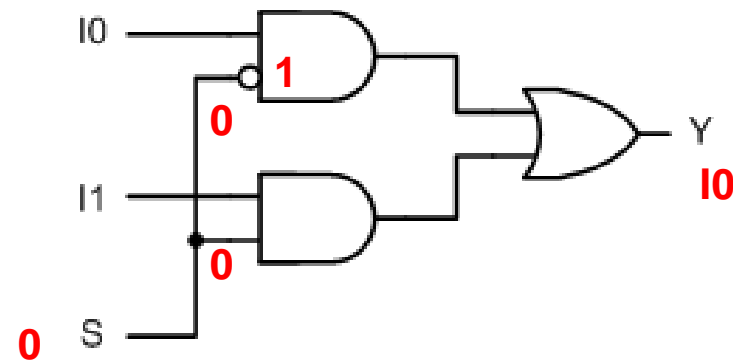
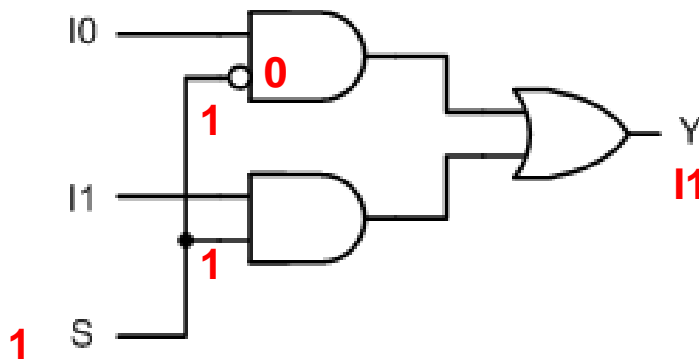
② Thus, input 1 = B is selected and passed to the output

① Select bits =  $1_2 = 1_{10}$ .

S	Y
0	i0
1	i1

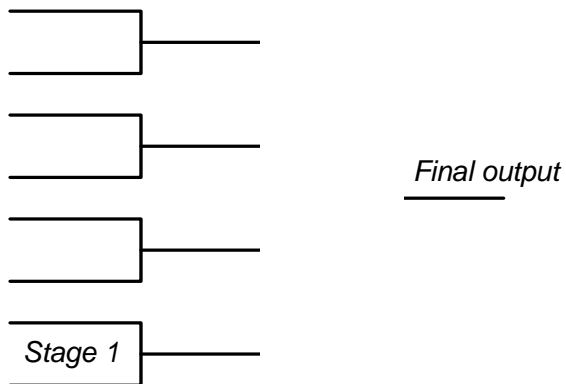
# Building a 2-to-1 Mux

- To build a mux
  - Decode the select bits and include the corresponding data input.
  - Finally OR all the first level outputs together.



# Building Large Muxes

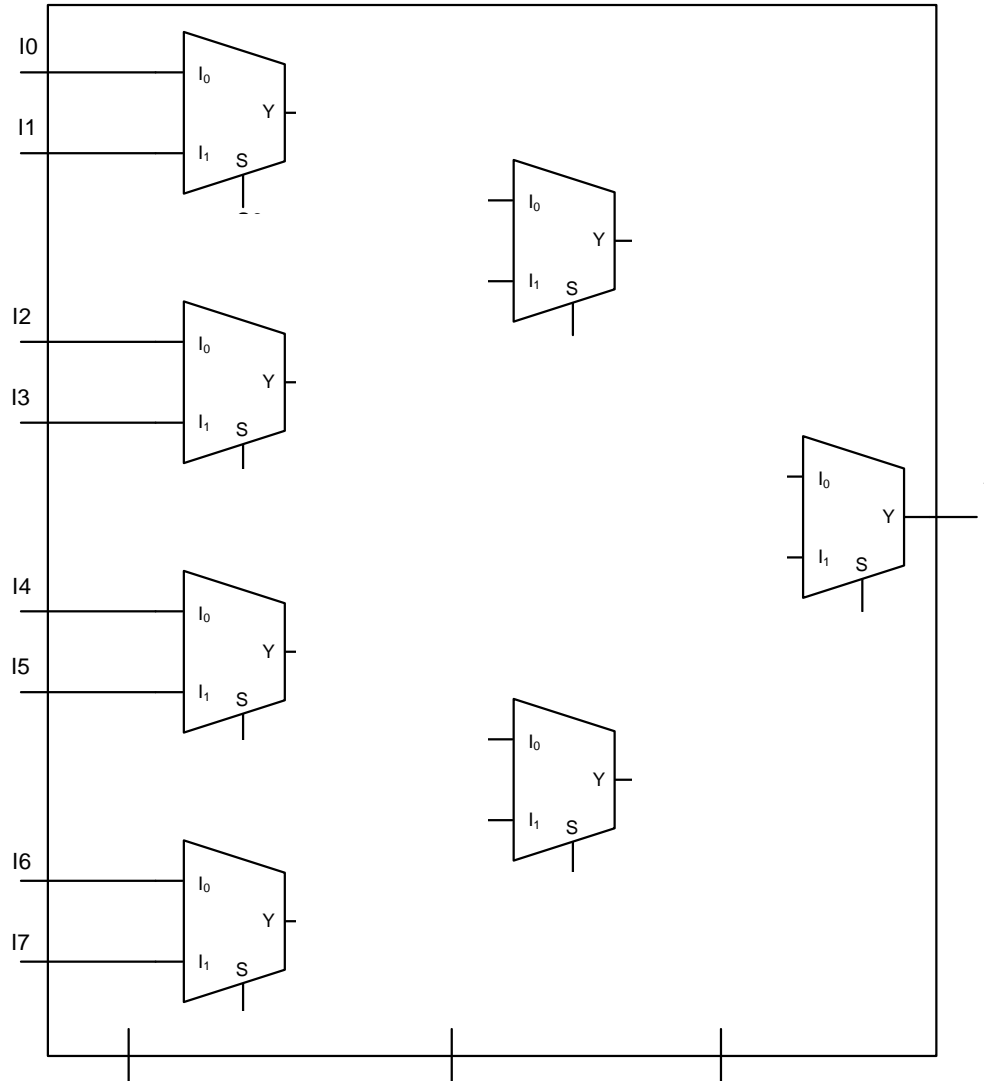
- When we build large muxes, the number of inputs to the gates grows too large to build them directly
- Instead, we will build larger muxes from smaller muxes
- Similar to a tournament of sports teams
  - Many teams enter and then are narrowed down to 1 winner
  - In each round winners play \_\_\_\_\_



Railroad Switch Station

# Design an 8-to-1 mux with 2-to-1

## Muxes

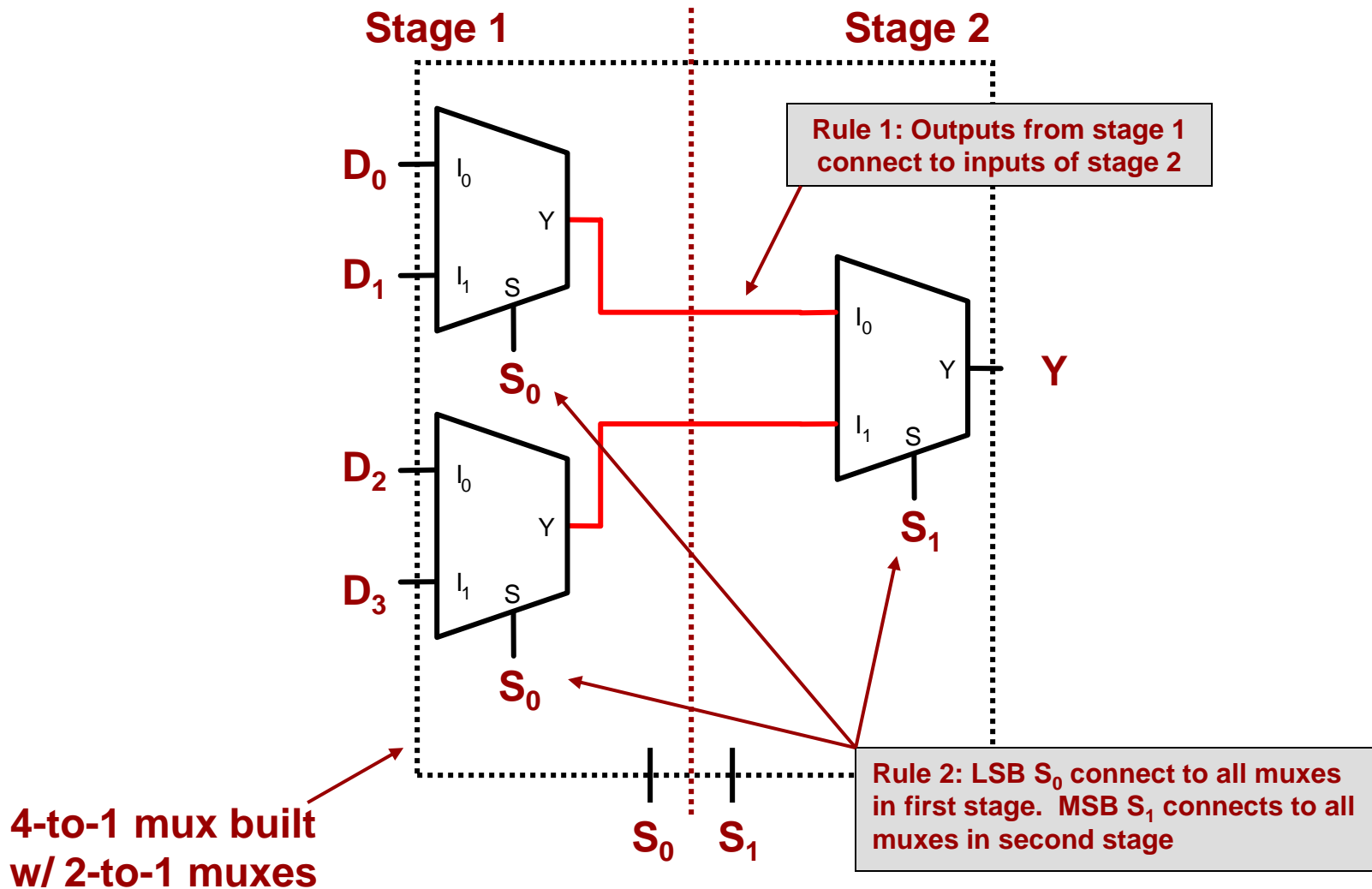




# Cascading Muxes

- Use several small muxes to build large ones
- Rules
  1. Arrange the muxes in stages (based on necessary number of inputs in 1<sup>st</sup> stage)
  2. Outputs of one stage feed to inputs of the next until only 1 final output
  3. All muxes in a stage connect to the same group of select bits
    - Usually, LSB connects to first stage
    - MSB connect to last stage

# Building a 4-to-1 Mux

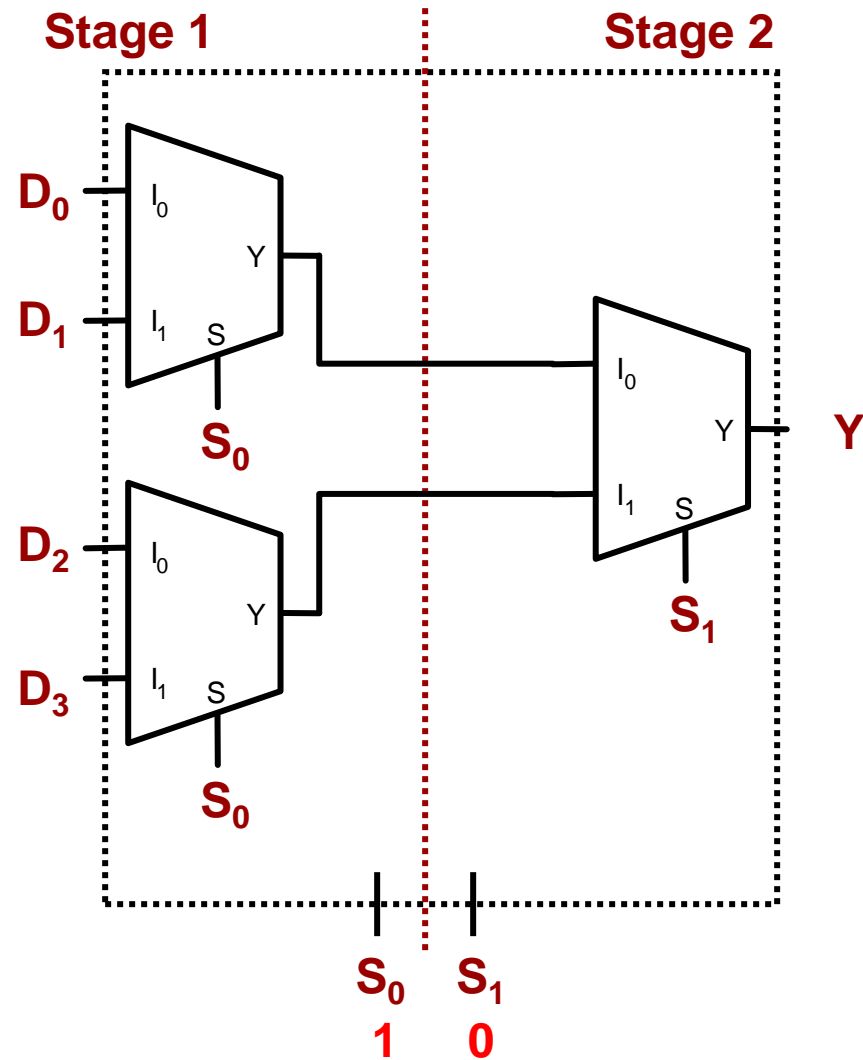


# Building a 4-to-1 Mux

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

Walk through an example:

$$S_1 S_0 = 01$$

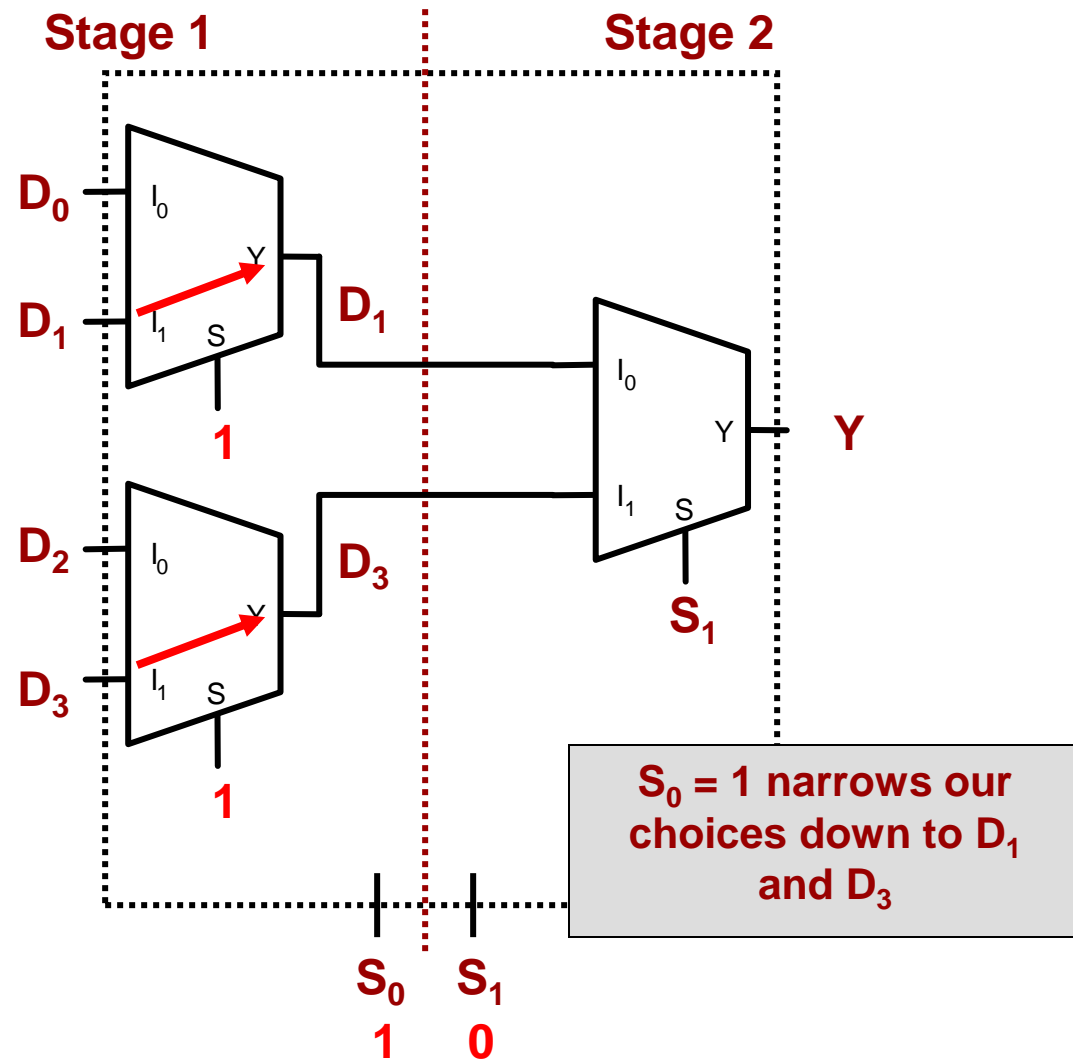


# Building a 4-to-1 Mux

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

Walk through an example:

$$S_1 S_0 = 01$$

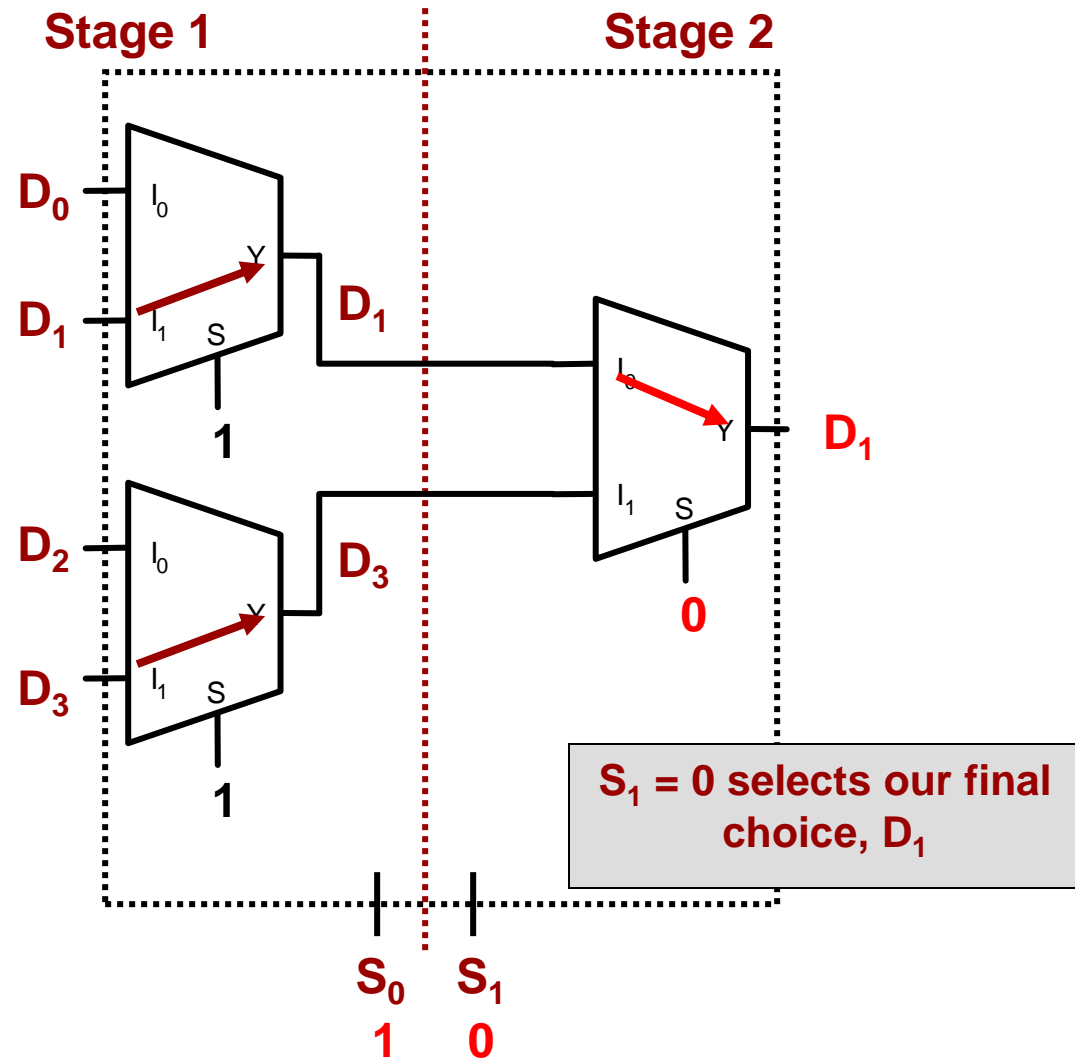


# Building a 4-to-1 Mux

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

Walk through an example:

$S_1 S_0 = 01$



# Device vs. System Labels

- When using hierarchy (i.e. building blocks) to design a circuit be sure to show both device and system labels
  - Device Labels: Signal names used \_\_\_\_\_ the block
    - \_\_\_\_\_ names the **designer/manufacture of the block** uses to indicate which input/output is which to the outside user (Names may \_\_\_\_\_; read the manual)
  - System labels: Signal names used \_\_\_\_\_ the block
    - \_\_\_\_\_ signals from the circuit being built
    - Can have the same name as the device label if such a signal name exists at the outside level

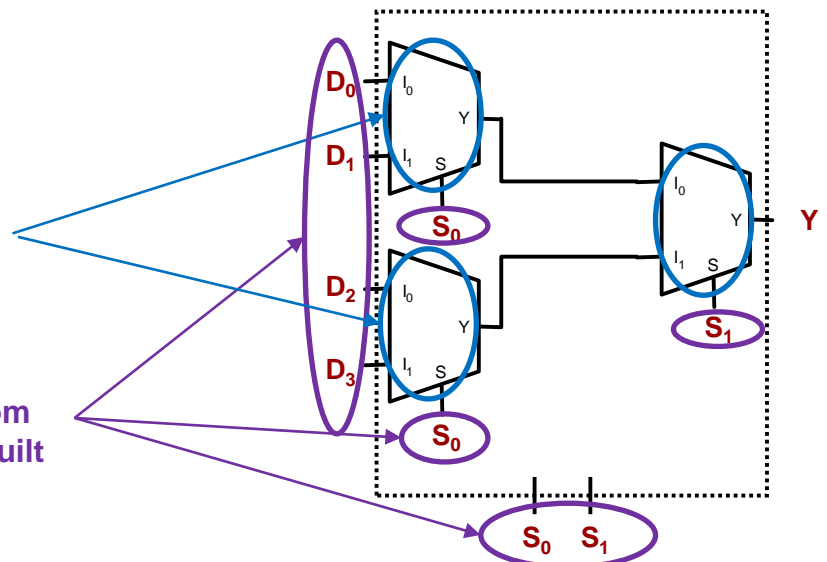
Analogy: **Formal** and **Actual** parameters in software function calls

- a and b are like device labels and indicate the names used inside a block.
- x and y are like system labels and represent the actual values to be used.

```
int div(int i0, int i1)
{ int t = i0/i1;
  return t; }
int main()
{
  int d0=10, d1=2;
  int s = div(d0,d1);
}
```

**Device Labels:**  
Indicate which input/output is which inside the block.

**System Labels:**  
Actual signals from the circuit being built



# Exercise

- Sketch how you could build a 16-to-1 mux with 4-to-1 muxes? 8-to-1 and 2-to-1 muxes?

# Exercise

- Create a 3-to-1 mux using 2-to-1 muxes
  - Inputs:  $I_0, I_1, I_2$  and select bits  $S_1, S_0$
  - Output:  $Y$

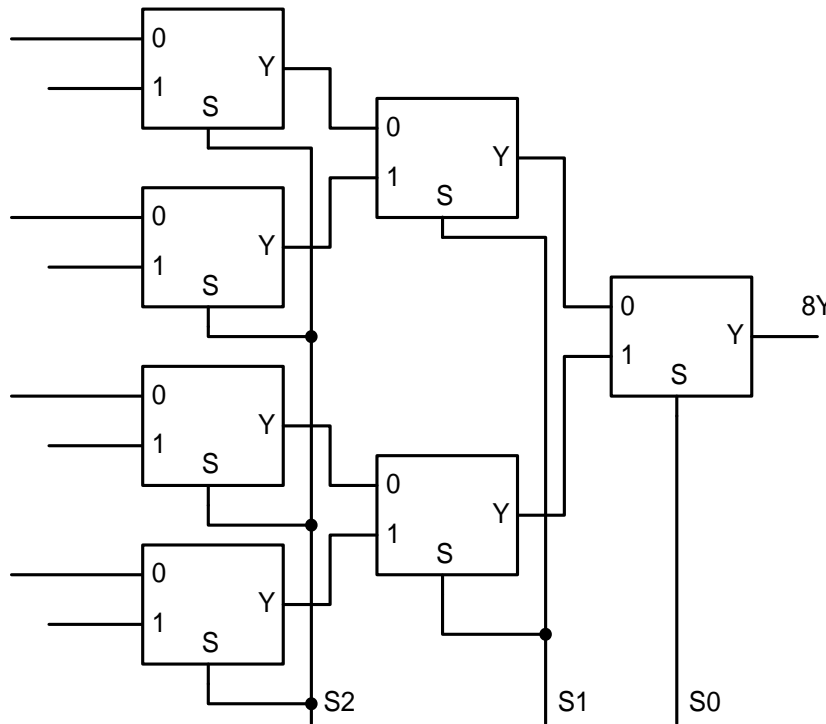
$S_1$	$S_0$	$Y$
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$





# Select-bit Ordering

- If we connect the select bits as shown to build an 8-to-1 mux, show how to label the inputs ( $i_0$ - $i_7$ ) so that the correct input is passed based on the binary value of  $S_2:S_0$



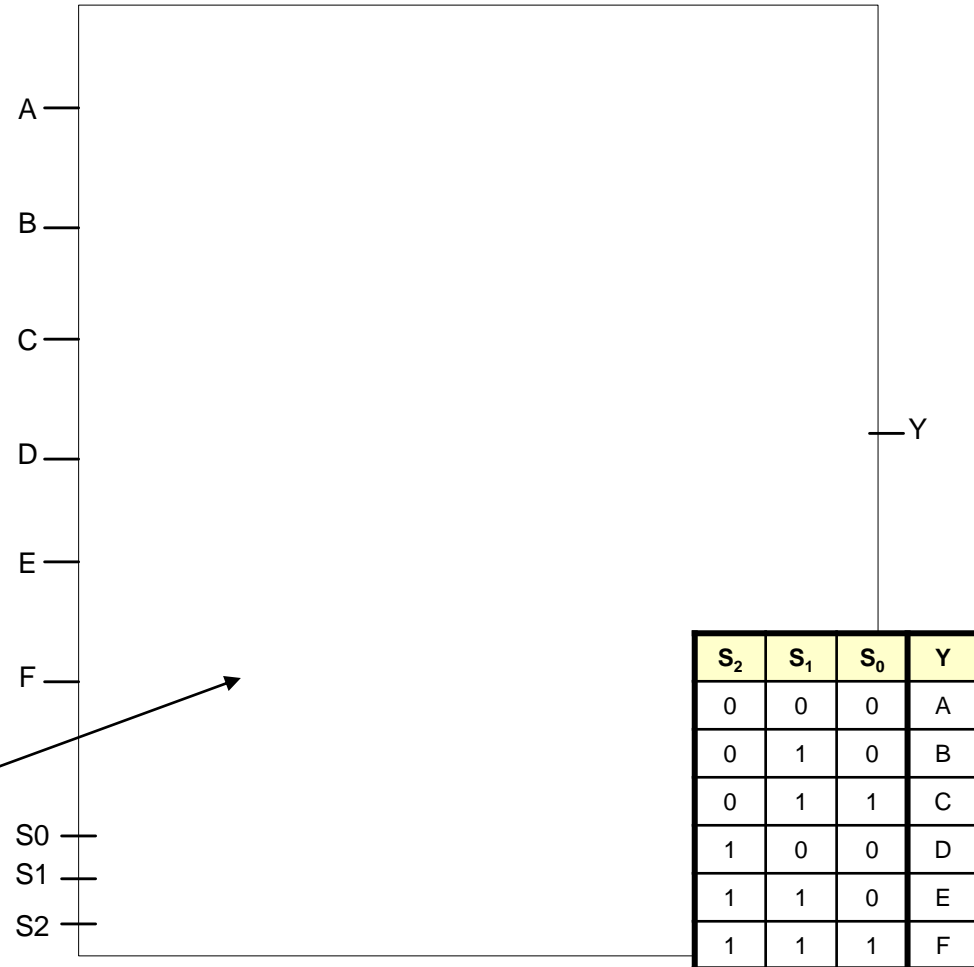
Selects			OUT Y
$S_2$	$S_1$	$S_0$	
0	0	0	
1			
0	1		
1			
0	0	1	
1			
0	1		
1			

# Alternate Select Bit Ordering Example

- Given 6 inputs: **A-F**, design a **6-to-1** mux from 4- and 2-to-1 muxes that uses the following select bit combinations

S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	Y
0	0	0	A
0	1	0	B
0	1	1	C
1	0	0	D
1	1	0	E
1	1	1	F

S <sub>2</sub>	S <sub>1</sub>	S <sub>0</sub>	Y
0	0	0	A
0	1	0	B
0	1	1	C
1	0	0	D
1	1	0	E
1	1	1	F



**Tip 1:** Whatever inputs you connect to a 4-to-1 mux, must correspond to 2 select bits that take on all combinations: 00, 01, 10, 11

**Tip 2:** For later stages, the select bit you connect must differentiate all potential options on 1 input from all the options on another (e.g. S1 differentiates A,D from B,C,E,F)

Another way to multiplex

# TRI-STATE GATES

# Typical Logic Gate

- Gates can output two values: 0 & 1
  - Logic '1' ( $V_{dd} = 3V$  or  $5V$ ), or Logic '0' (GND)
  - But they are ALWAYS outputting something!!!
- Analogy: a sink faucet
  - 2 possibilities: Hot ('1') or Cold ('0')
- In a real circuit, inputs cause *EITHER* a pathway from output to VDD *OR* VSS

Hot Water = Logic 1

Cold Water = Logic 0

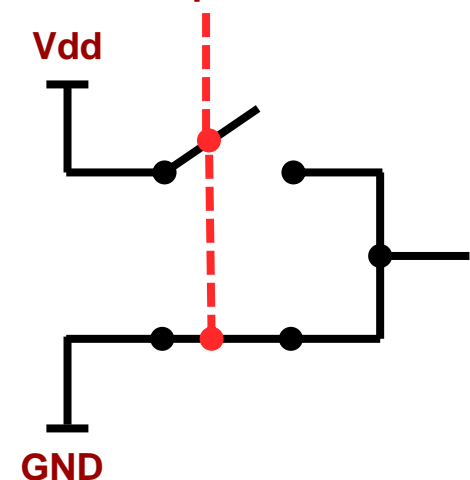
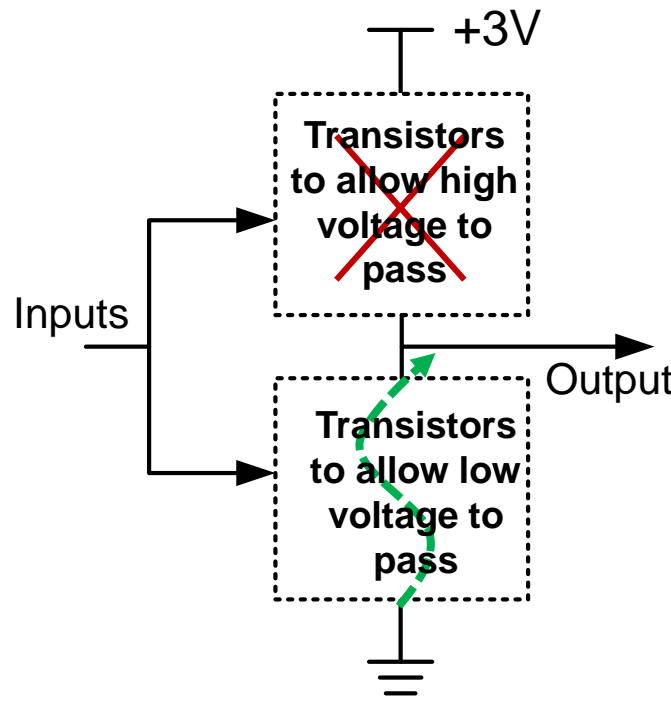
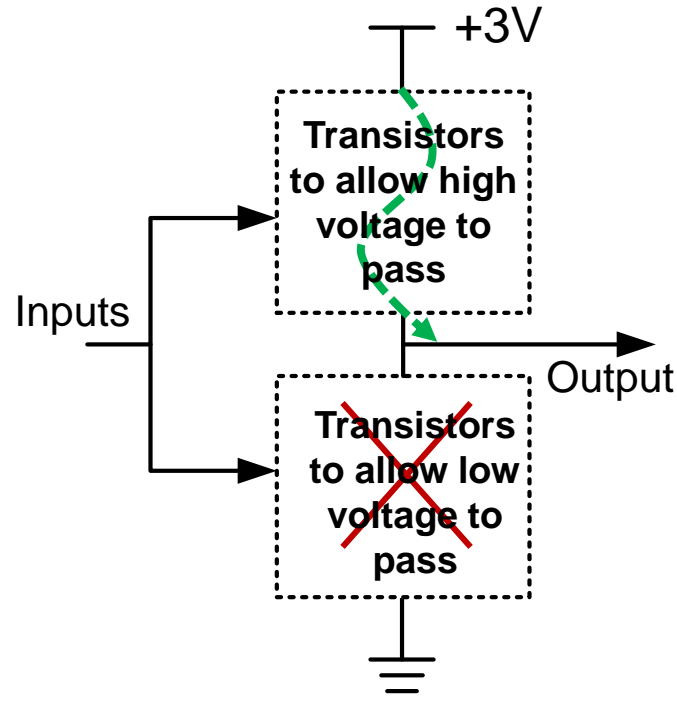
(Strapped together so always one type of water coming out)



Inputs

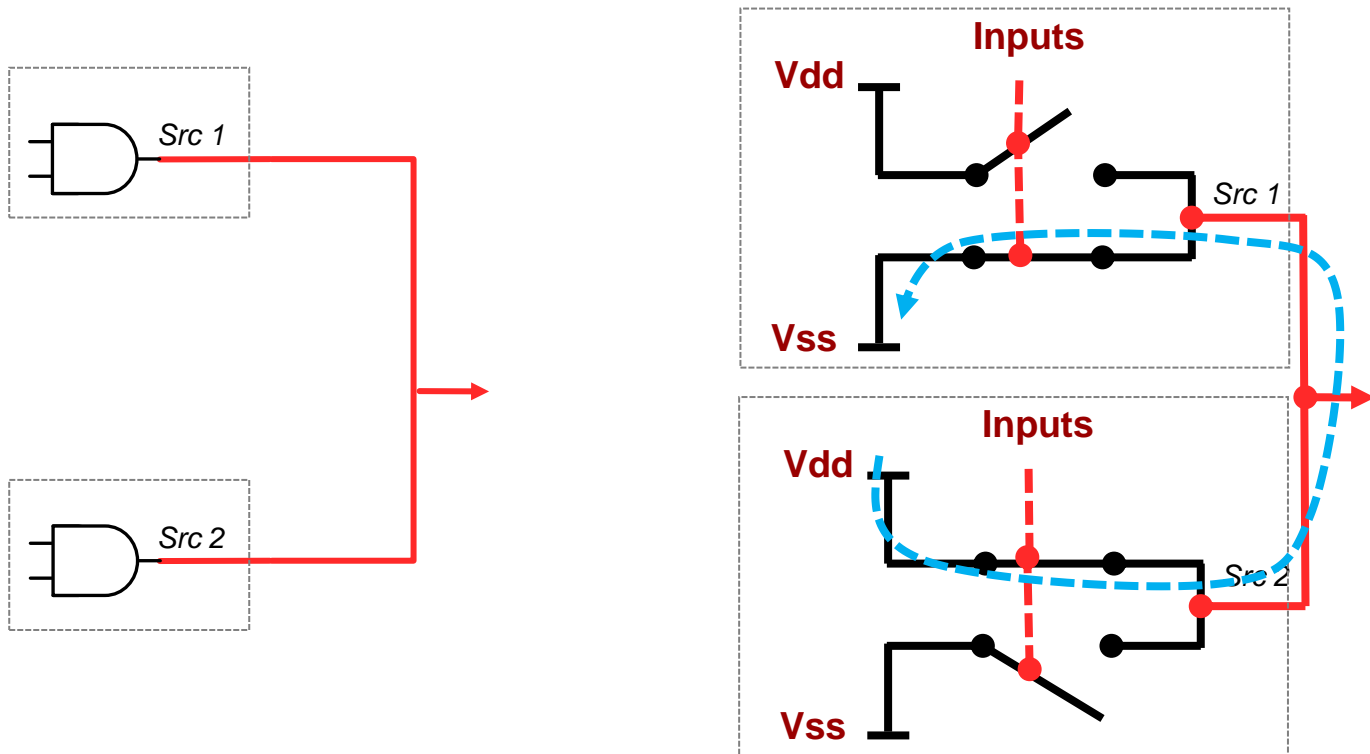
V<sub>dd</sub>

GND



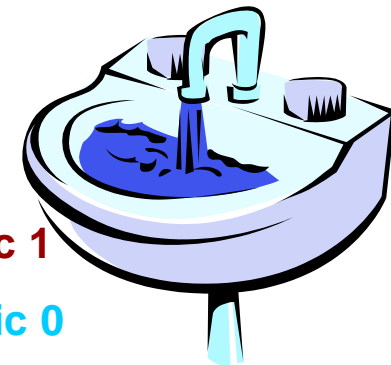
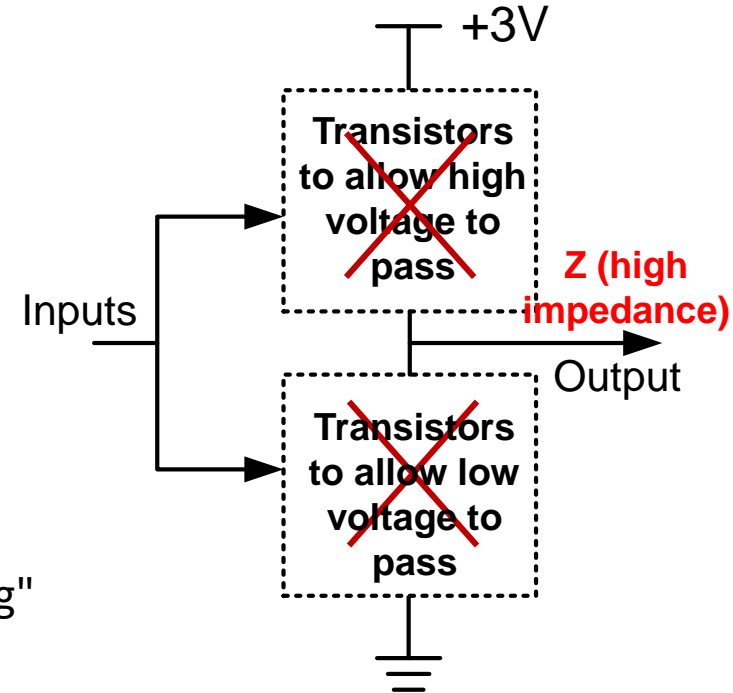
# Output Connections

- Can we connect the output of two logic gates together?
- Yes! Possible Wired-OR (static, low-resistance pathway from Vdd to GND)
- We call this situation “ Wired-OR ”



# Tri-State Buffers

- Normal digital gates can output two values: 0 & 1
  1. Logic 0 = 0 volts
  2. Logic 1 = 5 volts
- Tristate buffers can output a third value:
  3. \_\_\_\_\_ = \_\_\_\_\_ = "Floating"  
(no connection to any voltage source...infinite resistance)
- Analogy: a sink faucet
  - 3 possibilities:
    - 1.) Hot water,
    - 2.) Cold water,
    - 3.) \_\_\_\_\_ water



Hot Water = Logic 1

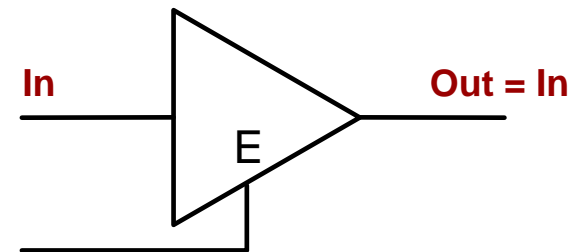
Cold Water = Logic 0

\_\_\_ Water = Z (High-Impedance)

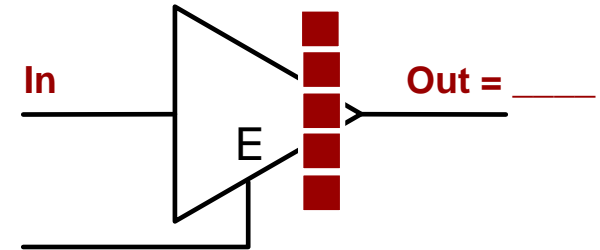
# Tri-State Buffers

- Tri-state buffers have an extra enable input
- When disabled, output is said to be at high impedance (a.k.a. Z)
  - High Impedance is equivalent to no connection (i.e. floating output) or an infinite resistance
  - It's like a brick wall between the output and any connection to source
- When enabled, normal buffer

## Tri-State Buffer



Enable=1

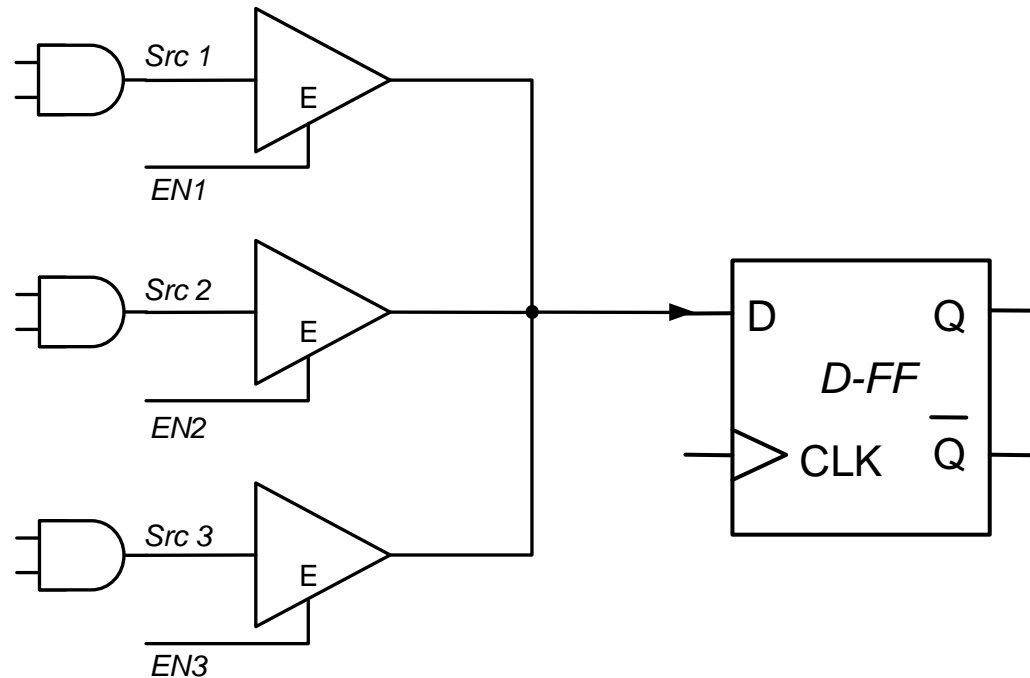


Enable=0

En	In	Out
0	-	Z
1	0	0
1	1	1

# Tri-State Buffers

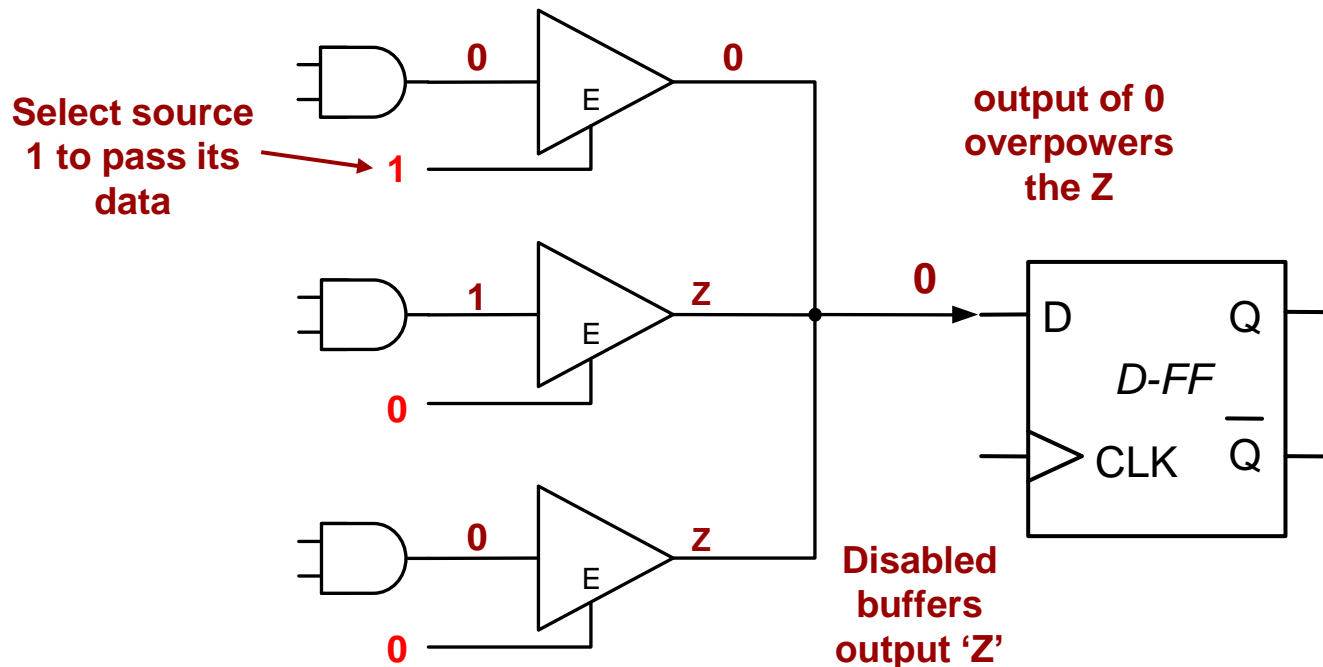
- We use tri-state buffers to \_\_\_\_\_ one output amongst several sources
- Rule: Only \_\_\_\_\_ at a time





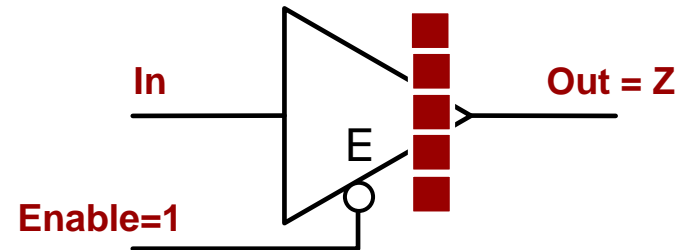
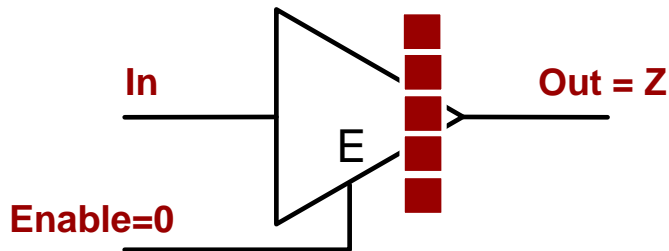
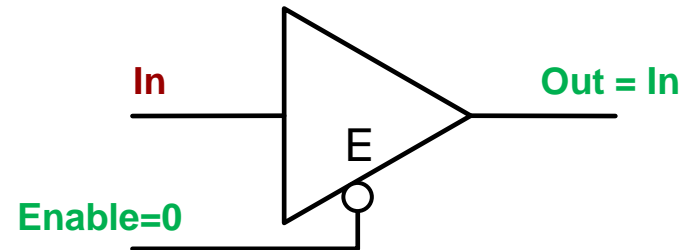
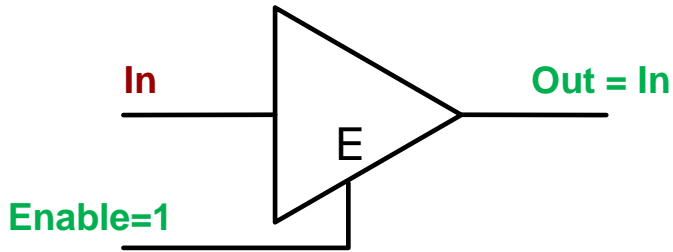
# Tri-State Buffers

- We use tri-state buffers to share one output amongst several sources
- Rule: Only 1 buffer enabled at a time
- When 1 buffer enabled, its output overpowers the Z's (no connection) from the other gates



# Enable Polarity

- **Side note:** Some tri-states are design to pass the input (be enabled) when the enable is 0 (rather than 1)
  - A inversion bubble is shown at the enable input to indicate the "\_\_\_\_\_" polarity needed to enable the tristate



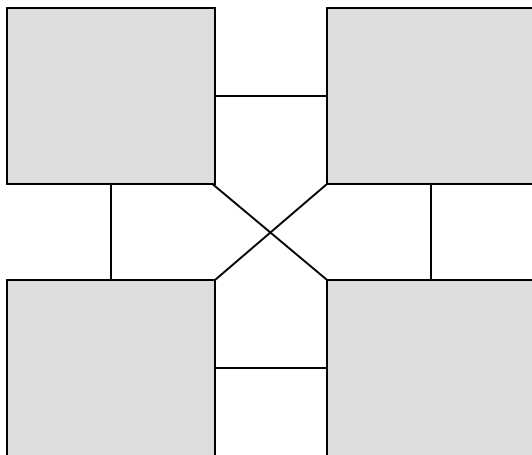
En	In	Out
0	-	Z
1	0	0
1	1	1

En	In	Out
1	-	Z
0	0	0
0	1	1

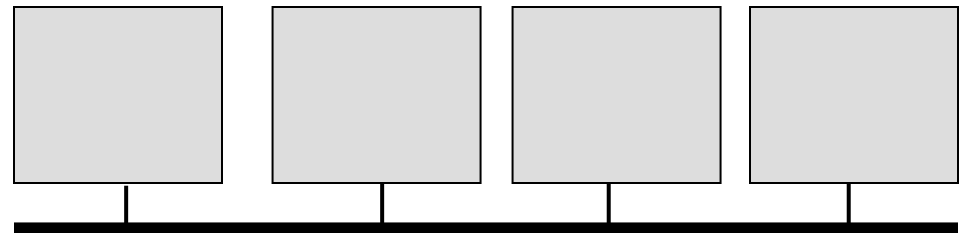
# Communication Connections

- Multiple entities need to communicate
- We could use
  - Point-to-point connections
  - A \_\_\_\_\_

**Separate point to point connections**

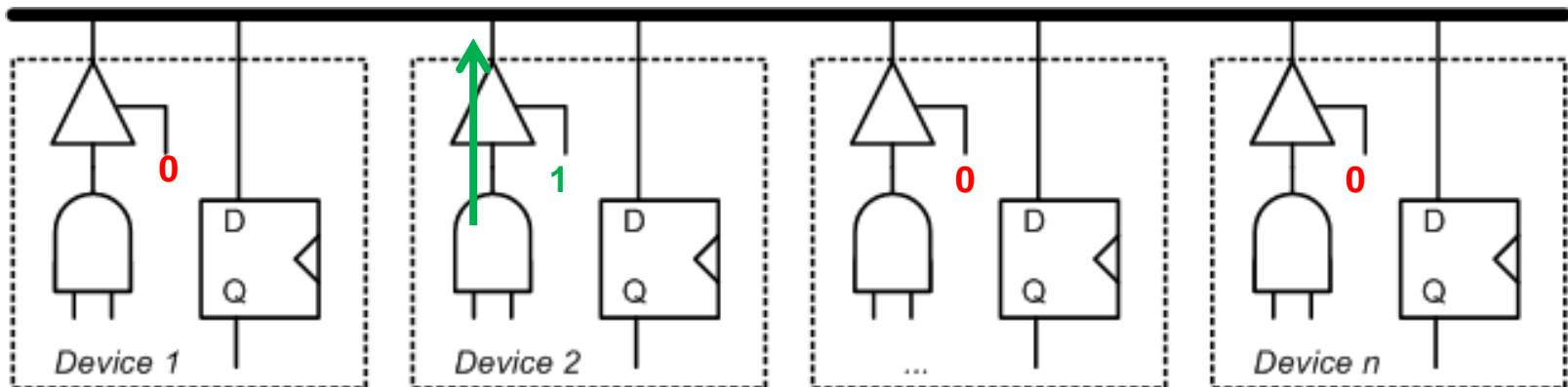


**Shared Bus**



# Bidirectional Bus

- 1 transmitter (otherwise bus contention)
- N receivers
- Each device can send (though 1 at a time) or receive



# Tri-State Gates

- Advantage: don't have to know in advance how many devices will be connected together
  - Tri-State gates give us the option of connecting together the outputs of many devices without requiring a circuit to multiplex many signals into one
- Just have to make sure only one is enabled (output active) at any one time.

