

## Spiral 1 / Unit 2

Basic Boolean Algebra

Logic Functions

Decoders

Multiplexers

© Mark Redekopp

## Outcomes

- I know the difference between combinational and sequential logic and can name examples of each.
- I understand latency, throughput, and at least 1 technique to improve throughput
- I can identify when I need state vs. a purely combinational function
  - I can convert a simple word problem to a logic function (TT or canonical form) or state diagram
- I can use Karnaugh maps to synthesize combinational functions with several outputs
- I understand how a register with an enable functions & is built
- I can design a working state machine given a state diagram
- I can implement small logic functions with complex CMOS gates

## BOOLEAN ALGEBRA INTRO

## Boolean Algebra

- A set of theorems to help us manipulate logical expressions/equations
- Axioms = Basis / assumptions used
- Theorems = manipulations that we can use

## Axioms

- Axioms are the basis for Boolean Algebra
- Notice that these axioms are simply restating our definition of digital/binary logic
  - A1/A1' = Binary variables (only 2 values possible)
  - A2/A2' = NOT operation
  - A3,A4,A5 = AND operation
  - A3',A4',A5' = OR operation

(A1)	$X = 0 \text{ if } X \neq 1$	(A1')	$X = 1 \text{ if } X \neq 0$
(A2)	If $X = 0$ , then $X' = 1$	(A2')	If $X = 1$ , then $X' = 0$
(A3)	$0 \cdot 0 = 0$	(A3')	$1 + 1 = 1$
(A4)	$1 \cdot 1 = 1$	(A4')	$0 + 0 = 0$
(A5)	$1 \cdot 0 = 0 \cdot 1 = 0$	(A5')	$0 + 1 = 1 + 0 = 1$

## Duality

- Every truth statement can yields another truth statement
  - I exercise if I have *time* and *energy* (original statement)
  - I *don't* exercise if I *don't* have *time* or *don't* have *energy* (dual statement)
- To express the dual, swap...

1's  $\longleftrightarrow$  0's

$\bullet \longleftrightarrow +$

## Duality

- The “dual” of an expression is not equal to the original

$$\begin{array}{ccc} 1 + 0 & \neq & 0 \cdot 1 \\ \text{Original} & & \text{Dual} \\ \text{expression} & & \end{array}$$

- Taking the “dual” of both sides of an equation yields a new equation

$$\begin{array}{ccc} X + 1 = 1 & \longrightarrow & X \cdot 0 = 0 \\ \text{Original equation} & & \text{Dual} \end{array}$$

## Single Variable Theorems

- Provide some simplifications for expressions containing:
  - a single variable
  - a single variable and a constant bit
- Each theorem has a dual (another true statement)
- Each theorem can be proved by writing a truth table for both sides (i.e. proving the theorem holds for all possible values of X)

T1	$X + 0 = X$	T1'	$X \cdot 1 = X$
T2	$X + 1 = 1$	T2'	$X \cdot 0 = 0$
T3	$X + X = X$	T3'	$X \cdot X = X$
T4	$(X')' = X$		
T5	$X + X' = 1$	T5'	$X \cdot X' = 0$

## Single Variable Theorem (T1)

$$X+0 = X \quad (T1)$$

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

OR

Hold Y constant

$$X \cdot 1 = X \quad (T1')$$

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

AND

Whenever a variable is OR'ed with 0, the output will be the same as the variable...

**"0 OR Anything equals that anything"**

Whenever a variable is AND'ed with 1, the output will be the same as the variable...

**"1 AND Anything equals that anything"**

## Single Variable Theorem (T2)

$$X+1 = 1 \quad (T2)$$

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

OR

Hold Y constant

$$X \cdot 0 = 0 \quad (T2')$$

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

AND

Whenever a variable is OR'ed with 1, the output will be 1...

**"1 OR anything equals 1"**

Whenever a variable is AND'ed with 0, the output will be 0...

**"0 AND anything equals 0"**

## Single Variable Theorem (T3)

$$X+X = X \quad (T3)$$

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

OR

$$X \cdot X = X \quad (T3')$$

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

AND

Whenever a variable is OR'ed with itself, the result is just the value of the variable

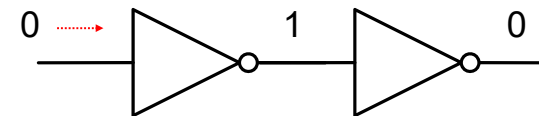
Whenever a variable is AND'ed with itself, the result is just the value of the variable

**This theorem can be used to reduce two identical terms into one OR to replicate one term into two.**

## Single Variable Theorem (T4)

$$(X')' = X \quad (T4)$$

$$\overline{\overline{X}} = X \quad (T4)$$



**Anything inverted twice yields its original value**

# Single Variable Theorem (T5)

$$X + \bar{X} = 1 \text{ (T5)}$$

$$X \cdot \bar{X} = 0 \text{ (T5')}$$

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

OR

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

AND

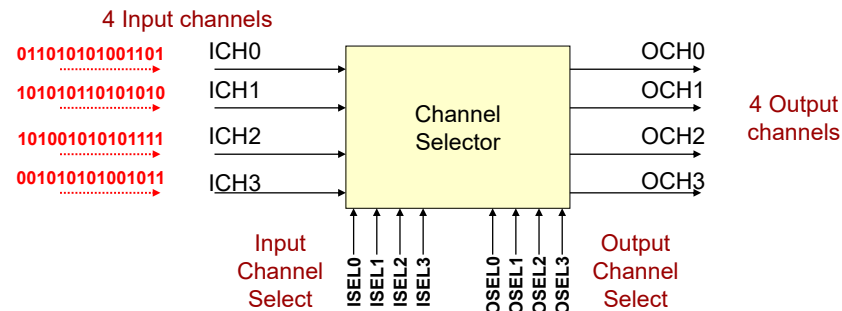
Whenever a variable is OR'ed with its complement, one value has to be 1 and thus the result is 1

Whenever a variable is AND'ed with its complement, one value has to be 0 and thus the result is 0

This theorem can be used to simplify variables into a constant or to expand a constant into a variable.

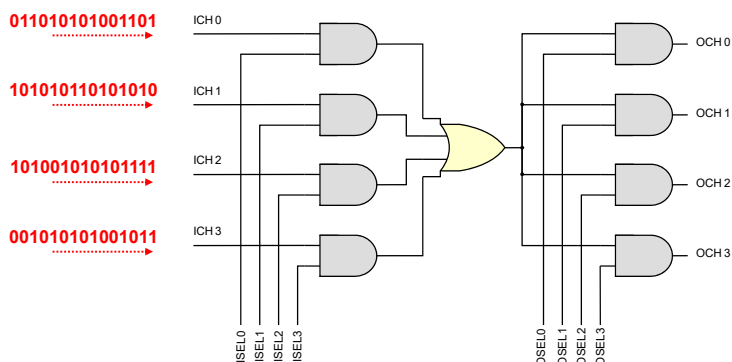
# Application: Channel Selector

- Given 4 input, digital music/sound channels and 4 output channels
- Given individual "select" inputs that select 1 input channel to be routed to 1 output channel



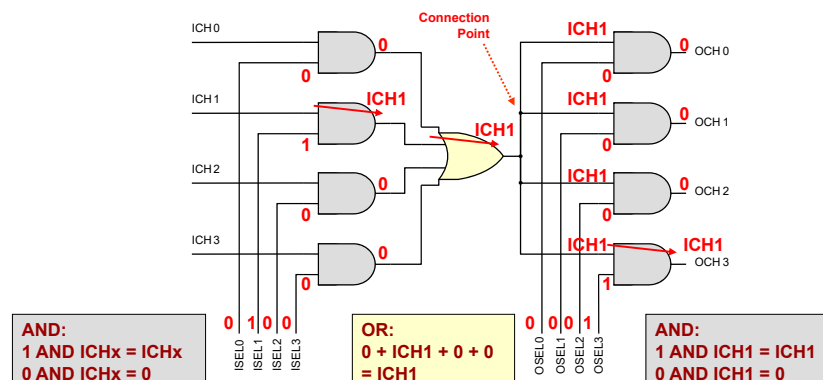
# Application: Steering Logic

- 4-input music channels (ICHx)
  - Select one input channel (use ISELx inputs)
  - Route to one output channel (use OSELx inputs)



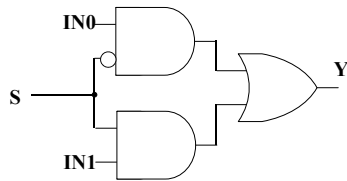
# Application: Steering Logic

- 1<sup>st</sup> Level of AND gates act as barriers only passing 1 channel
- OR gates combines 3 streams of 0's with the 1 channel that got passed (i.e. ICH1)
- 2<sup>nd</sup> Level of AND gates passes the channel to only the selected output



## Your Turn

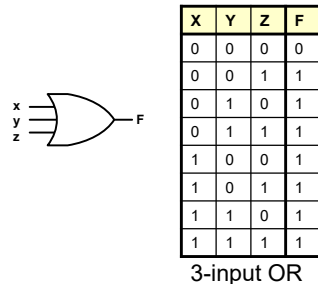
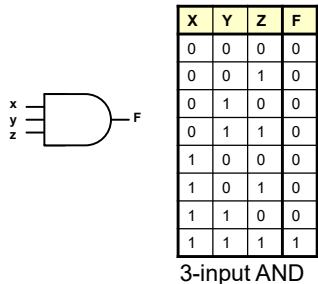
- Build a circuit that takes 3 inputs: S, IN0, IN1 and outputs a single bit Y.
- It's functions should be:
  - If S = 0, Y = IN0 (IN0 passes to Y)
  - If S = 1, Y = IN1 (IN1 passes to Y)



## CHECKERS / DECODERS

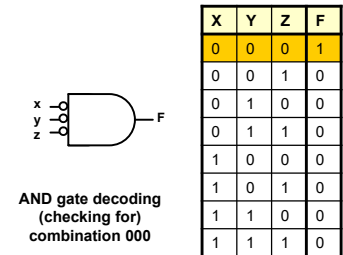
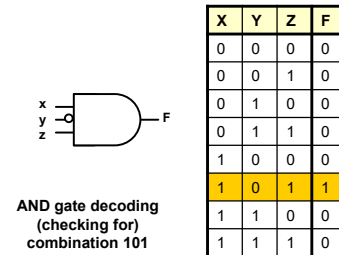
## Gates

- Gates can have more than 2 inputs but the functions stay the same
  - AND = output = 1 if ALL inputs are 1
    - Outputs 1 for only 1 input combination
  - OR = output = 1 if ANY input is 1
    - Outputs 0 for only 1 input combination



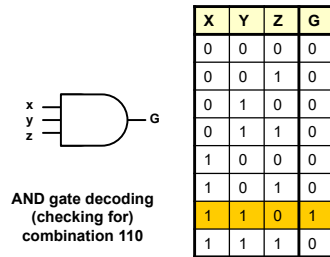
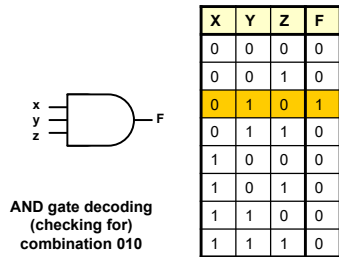
## Checkers / Decoders

- An AND gate only outputs '1' for 1 combination
  - That combination can be changed by adding inverters to the inputs
  - We can think of the AND gate as "checking" or "decoding" a specific combination and outputting a '1' when it matches.



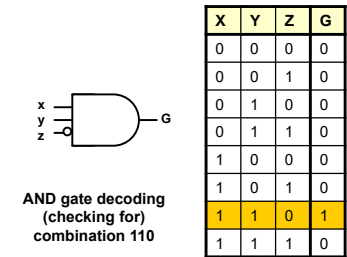
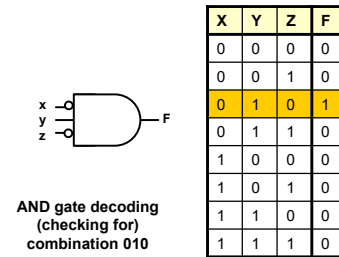
## Checkers / Decoders

- Place inverters at the input of the AND gates such that
  - F produces '1' only for input combination  $\{x,y,z\} = \{010\}$
  - G produces '1' only for input combination  $\{x,y,z\} = \{110\}$



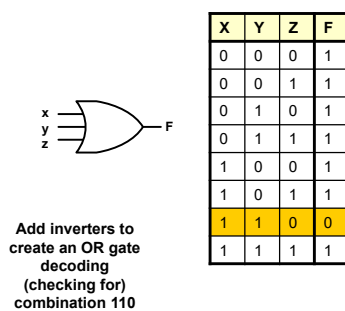
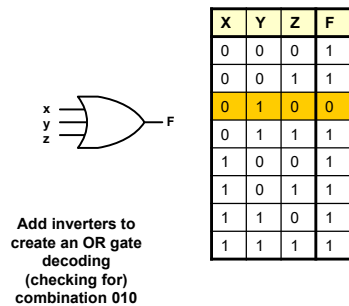
## Checkers / Decoders

- Place inverters at the input of the AND gates such that
  - F produces '1' only for input combination  $\{x,y,z\} = \{010\}$
  - G produces '1' only for input combination  $\{x,y,z\} = \{110\}$



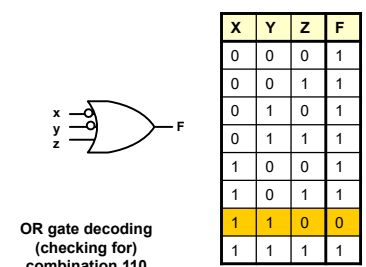
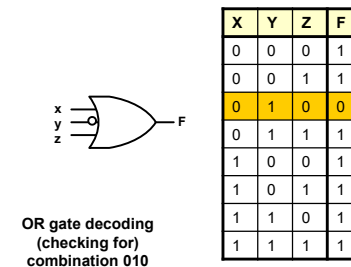
## Checkers / Decoders

- An OR gate only outputs '0' for 1 combination
  - That combination can be changed by adding inverters to the inputs



## Checkers / Decoders

- An OR gate only outputs '0' for 1 combination
  - That combination can be changed by adding inverters to the inputs
  - We can think of the OR gate as "checking" or "decoding" a specific combination and outputting a '0' when it matches.



## Decoder Exercise

- Design an instruction decoder that uses opcode[5:0] and func[5:0] as inputs and produces a separate output {ADD, SRL, SUB, etc.} for each instruction type that will produce '1' when that instruction is loaded

R-Type	opcode	rs	rt	rd	shamt	func
	000000	01000	10001	00101	00111	100000
	ADD	\$8	\$17	\$5	00000	
	000000	01000	10001	00101	00111	000010
	SRL	\$8	\$17	\$5	7	
000000	01000	10001	00101	00111	100010	
SUB	\$8	\$17	\$5	7		
I-Type	opcode	rs	rt	immediate		
	001000	11000	00101	0000	0000	0000 0001
	ADDI	\$24	\$5	20		
	100011	00011	00101	1111	1111	1111 1000
	LW	\$3	\$5	-8		
	101011	00011	00101	1111	1111	1111 1000
	SW	\$3	\$5	-8		
	000100	00011	00101	1111	1111	1111 1000
BEQ	\$3	\$5	-8			
J-Type	opcode	Jump address				
	000010	Jump address				
	J	26-bits				

## Decoder Exercise

ADD

SRL

SUB

R-Type	opcode	rs	rt	rd	shamt	func
	000000	01000	10001	00101	00111	100000
	ADD	\$8	\$17	\$5	00000	
	000000	01000	10001	00101	00111	000010
	SRL	\$8	\$17	\$5	7	
000000	01000	10001	00101	00111	100010	
SUB	\$8	\$17	\$5	7		
I-Type	opcode	rs	rt	immediate		
	001000	11000	00101	0000	0000	0000 0001
	ADDI	\$24	\$5	20		
	100011	00011	00101	1111	1111	1111 1000
	LW	\$3	\$5	-8		
	101011	00011	00101	1111	1111	1111 1000
	SW	\$3	\$5	-8		
	000100	00011	00101	1111	1111	1111 1000
BEQ	\$3	\$5	-8			
J-Type	opcode	Jump address				
	000010	Jump address				
	J	26-bits				

## Decoder Exercise

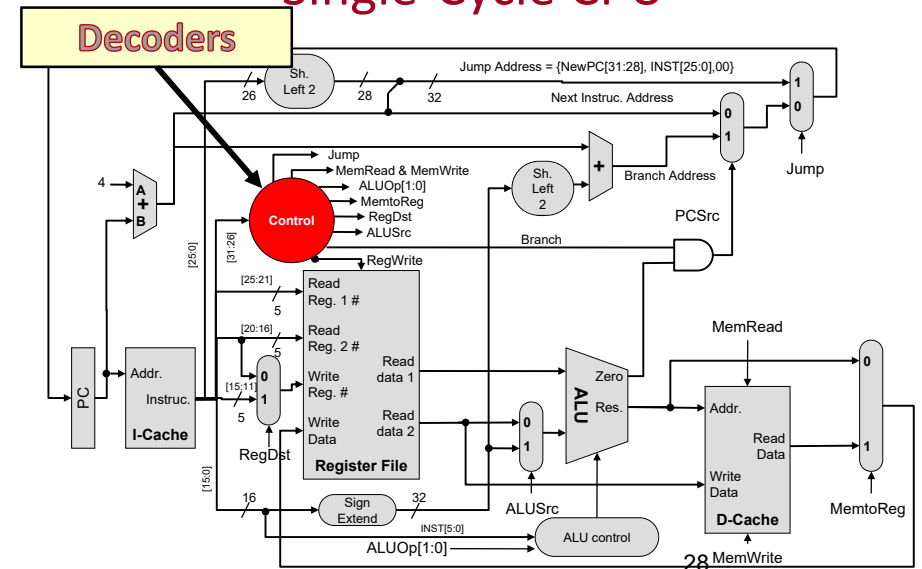
ADDI

LW

SW

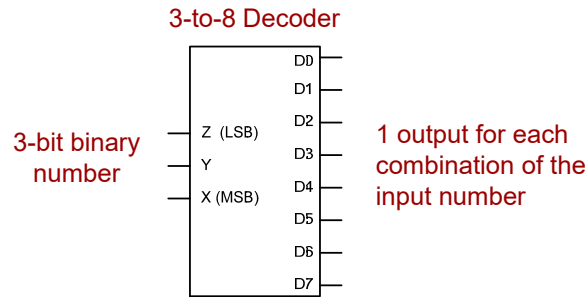
R-Type	opcode	rs	rt	rd	shamt	func
	000000	01000	10001	00101	00111	100000
	ADD	\$8	\$17	\$5	00000	
	000000	01000	10001	00101	00111	000010
	SRL	\$8	\$17	\$5	7	
000000	01000	10001	00101	00111	100010	
SUB	\$8	\$17	\$5	7		
I-Type	opcode	rs	rt	immediate		
	001000	11000	00101	0000	0000	0000 0001
	ADDI	\$24	\$5	20		
	100011	00011	00101	1111	1111	1111 1000
	LW	\$3	\$5	-8		
	101011	00011	00101	1111	1111	1111 1000
	SW	\$3	\$5	-8		
	000100	00011	00101	1111	1111	1111 1000
BEQ	\$3	\$5	-8			
J-Type	opcode	Jump address				
	000010	Jump address				
	J	26-bits				

## Single-Cycle CPU



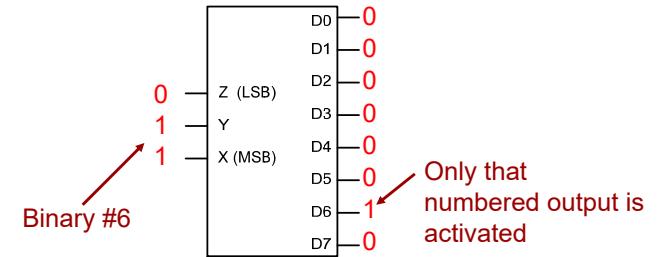
# Full Decoders

- A full 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 EVERY (MOST) input combination (i.e.  $2^n$  outputs)



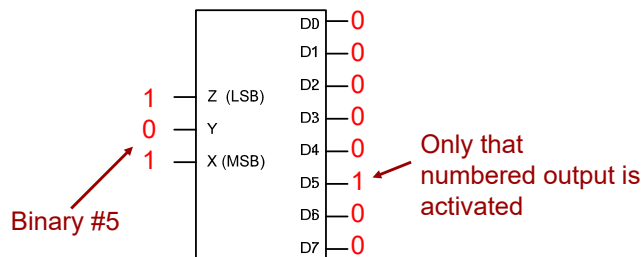
# Decoders

- 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')



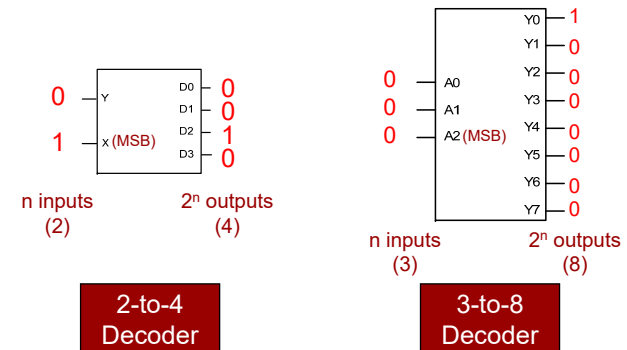
# Decoders

- 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')



# Decoder Sizes

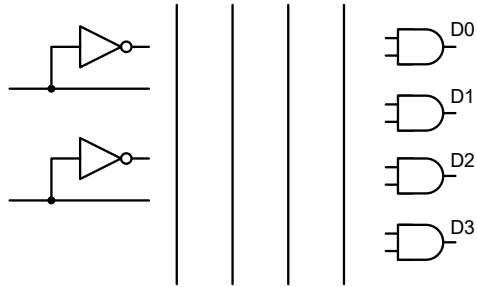
- A decoder w/ an **n-bit input** has  **$2^n$  outputs**
  - 1 output for every combination of the n-bit input



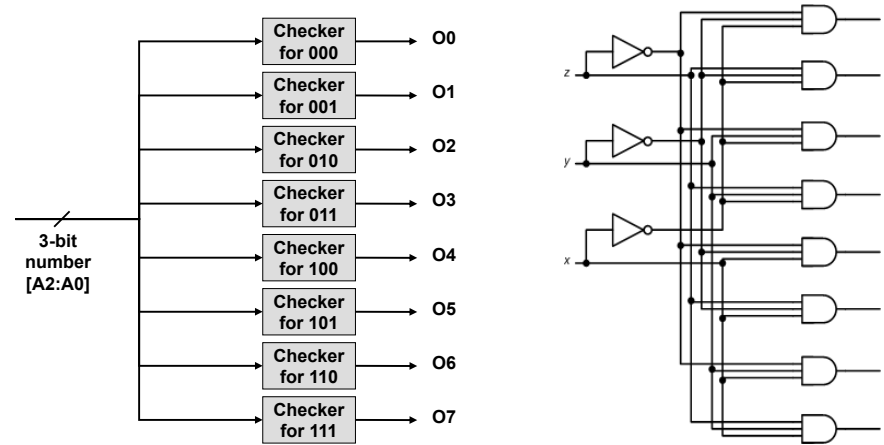


## Exercise

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

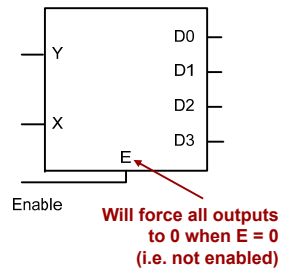
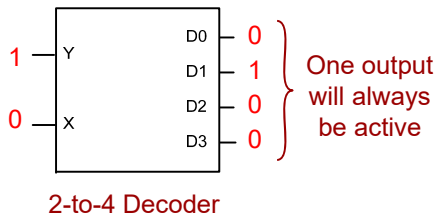


## Building Decoders

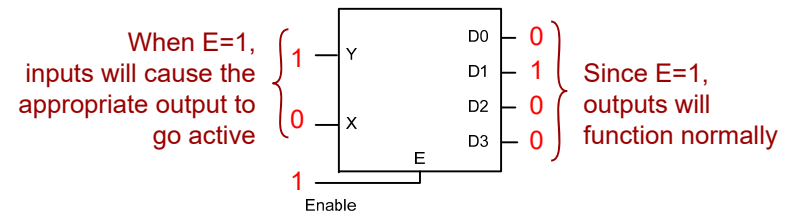
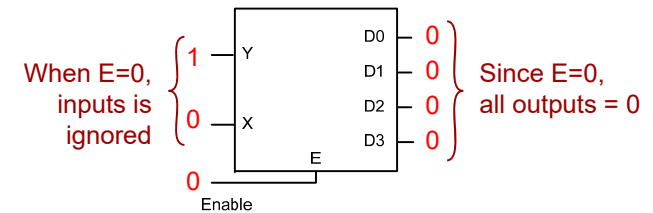


## Enables

- Exactly one output is active at all times
- It may be undesirable to always have an active output
- Add an extra input (called an enable) that can independently force all the outputs to their inactive values

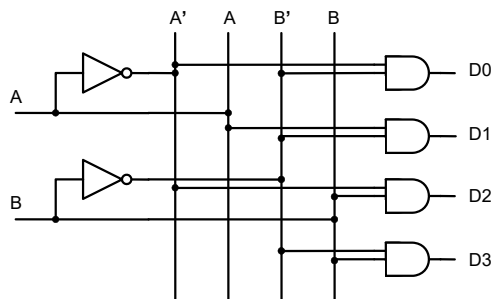


## Enables



## Implementing Enables

- Original 2-to-4 decoder



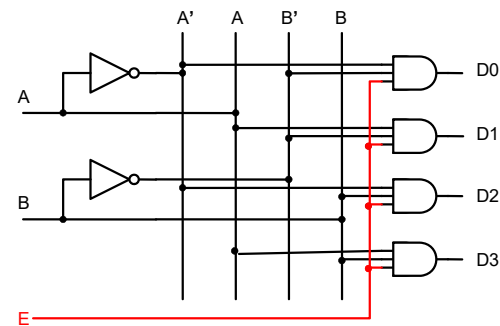
E —

When E=0, force all outputs = 0

When E=1, outputs operate as they did originally

## Enables

- Enables can be implemented by connecting it to each AND gate of the decoder

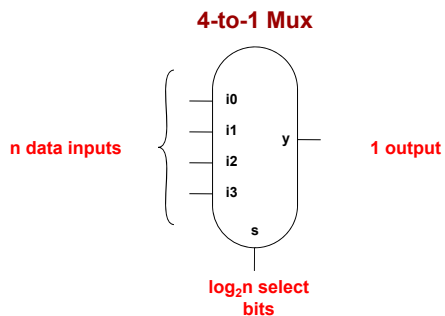


When E=0, 0 AND anything = 0

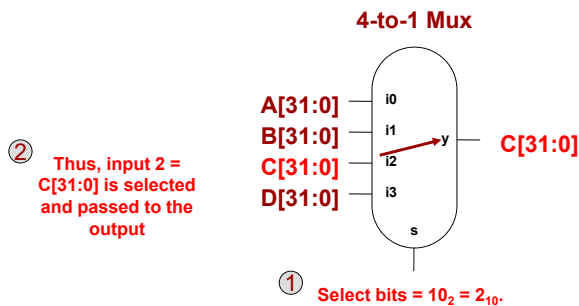
When E=1, 1 AND anything = that anything, which was the normal decoding logic

## Multiplexers

- Along with adders, multiplexers are most used building block
- 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

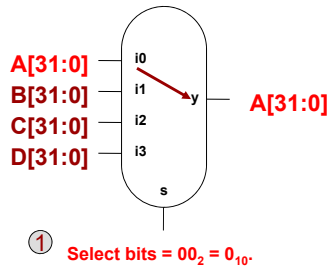


## Multiplexers



# Multiplexers

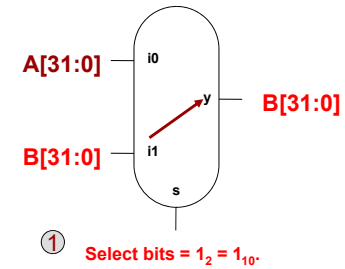
4-to-1 Mux



② Thus, input 0 = A[31:0] is selected and passed to the output

# Multiplexers

2-to-1 Mux

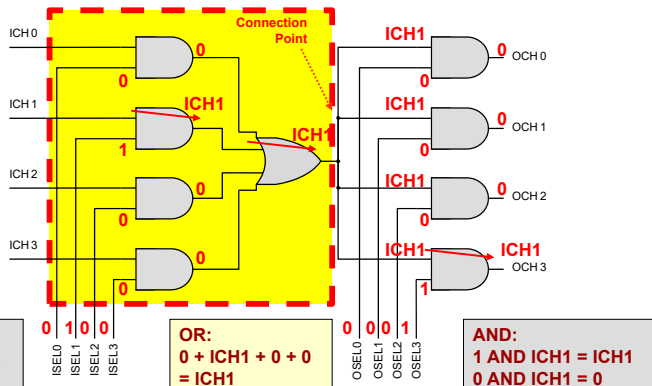


② Thus, input 1 = B[31:0] is selected and passed to the output

# Recall Using T1/T2

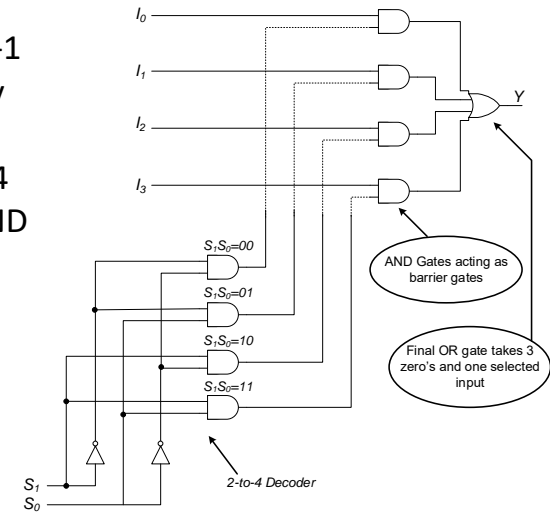
- 1<sup>st</sup> Level of AND gates act as barriers only passing 1 channel
- OR gates combines 3 streams of 0's with the 1 channel that got passed (i.e. ICH1)
- 2<sup>nd</sup> Level of AND gates passes the channel to only the selected output

Essentially this logic forms a 4-to-1 mux where one level of gates blocks all but 1 and then the OR gate combines all signals



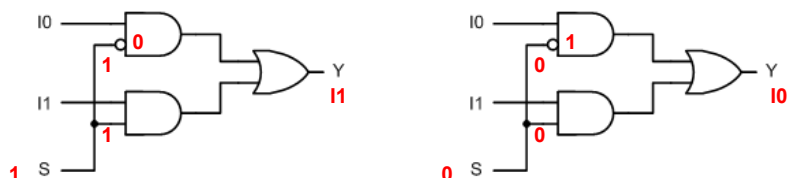
# Exercise: Build a 4-to-1 mux

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



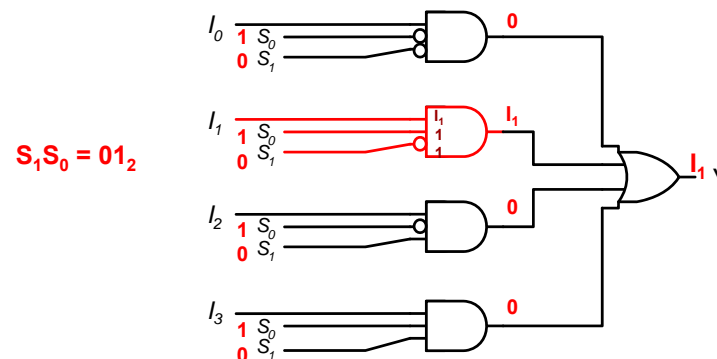
## Building a Mux

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



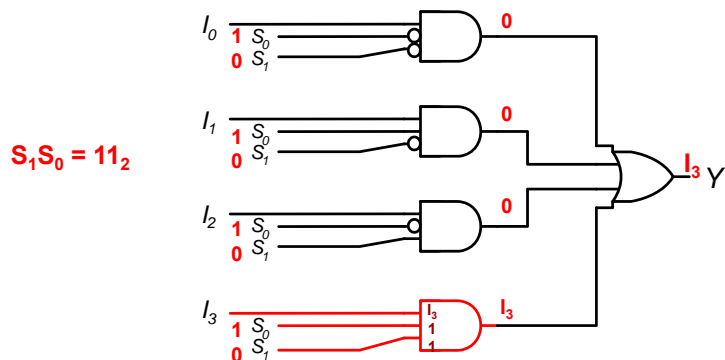
## Building a Mux

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



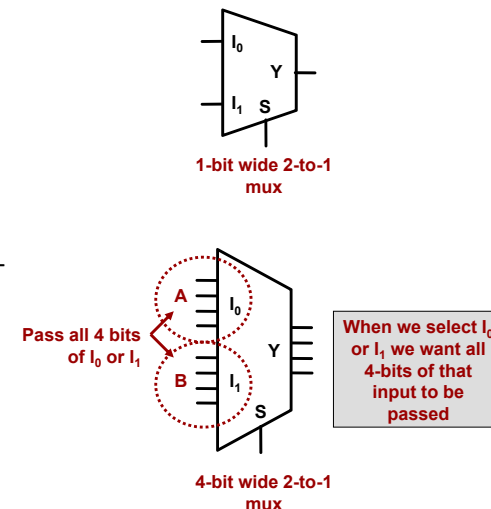
## Building a Mux

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



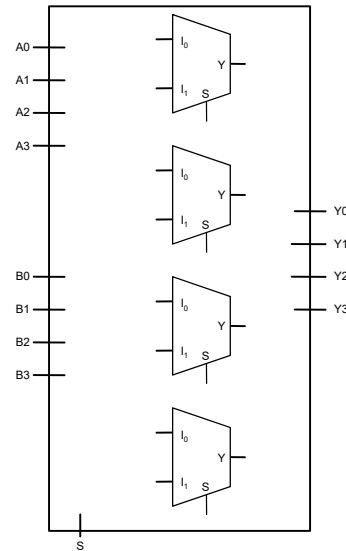
## Building Wide Muxes

- So far muxes only have single bit inputs...
  - $I_0$  is only 1-bit
  - $I_1$  is only 1-bit
- What if we still want to select between 2 inputs but now each input is a 4-bit number
- Use a 4-bit wide 2-to-1 mux



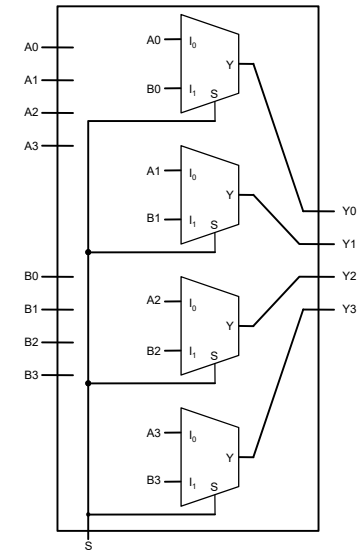
## Building Wide Muxes

- To build a 4-bit wide 2-to-1 mux, use 4 separate 2-to-1 muxes
- When  $S=0$ , all muxes pass their  $I_0$  inputs which means all the A bits get through
- When  $S=1$ , all muxes pass their  $I_1$  inputs which means all the B bits get through
- In general, to build an **m-bit wide n-to-1 mux**, use **m individual (separate) n-to-1 muxes**



## Building Wide Muxes

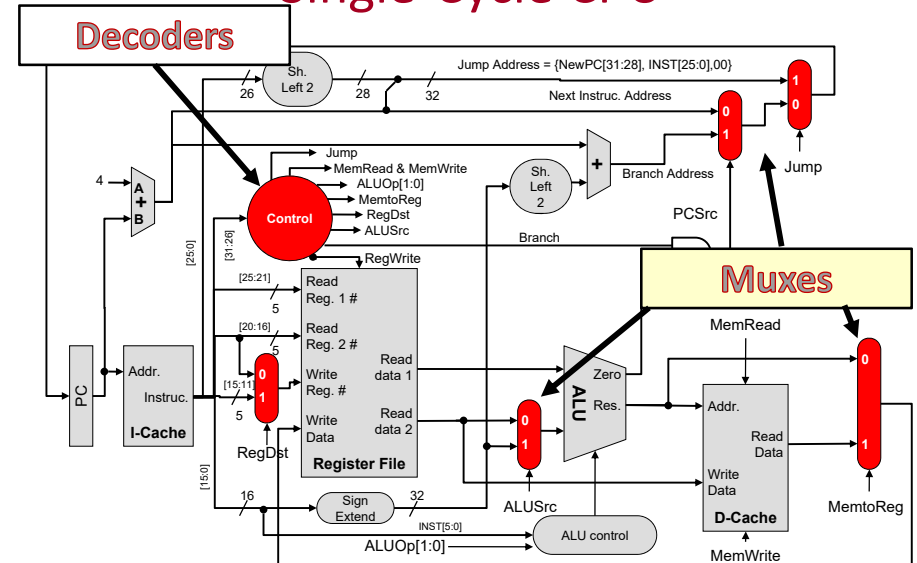
- To build a 4-bit wide 2-to-1 mux, use 4 separate 2-to-1 muxes
- When  $S=0$ , all muxes pass their  $I_0$  inputs which means all the A bits get through
- When  $S=1$ , all muxes pass their  $I_1$  inputs which means all the B bits get through
- In general, to build an **m-bit wide n-to-1 mux**, use **m individual (separate) n-to-1 muxes**



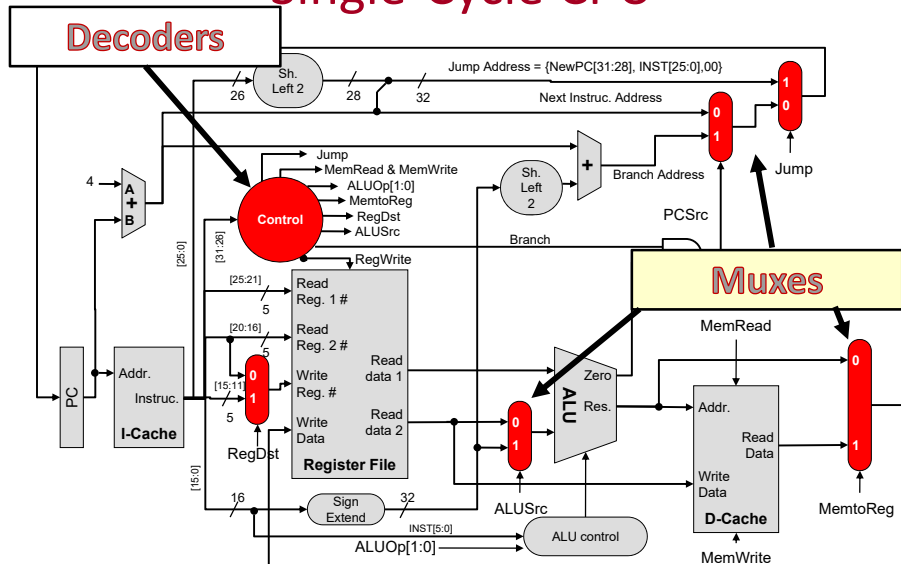
## Exercise

- How many 1-bit wide muxes and of what size would you need to build a **4-to-1, 8-bit wide mux** (i.e. there are 4 numbers:  $W[7:0]$ ,  $X[7:0]$ ,  $Y[7:0]$  and  $Z[7:0]$  and you must select one)
- How many 1-bit wide muxes and of what size would you need to build a **8-to-1, 2-bit wide mux**?

## Single-Cycle CPU

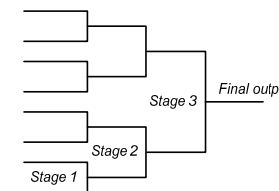


## Single-Cycle CPU

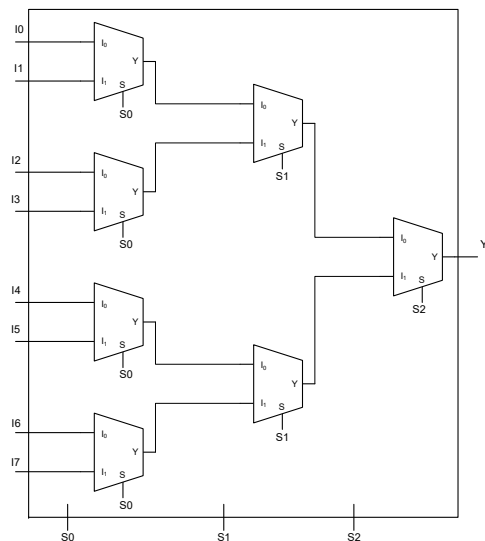


## Building Large Muxes

- Similar to a tournament of sports teams
  - Many teams enter and then are narrowed down to 1 winner
  - In each round winners play winners



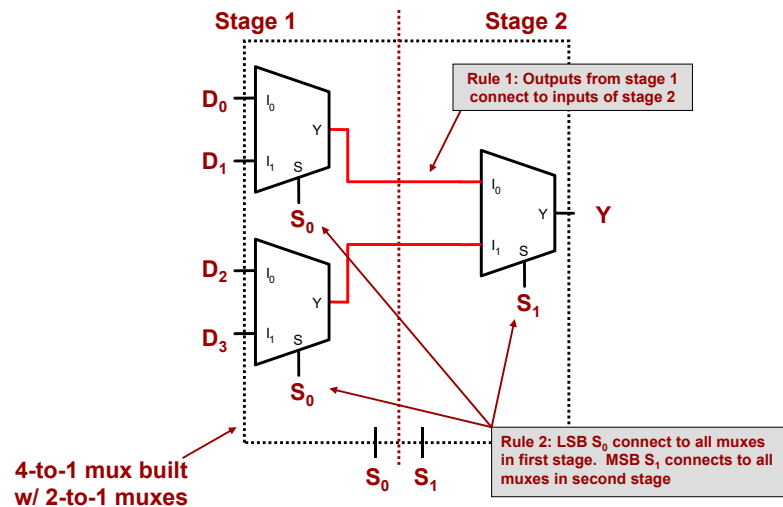
## Design an 8-to-1 mux with 2-to-1 Muxes



## 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 1 stage feed to inputs of the next
  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

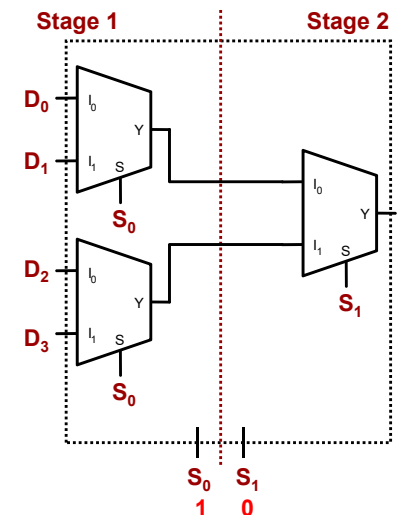


## Building a 4-to-1 Mux

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

Walk through an example:

$$S_1 S_0 = 01$$

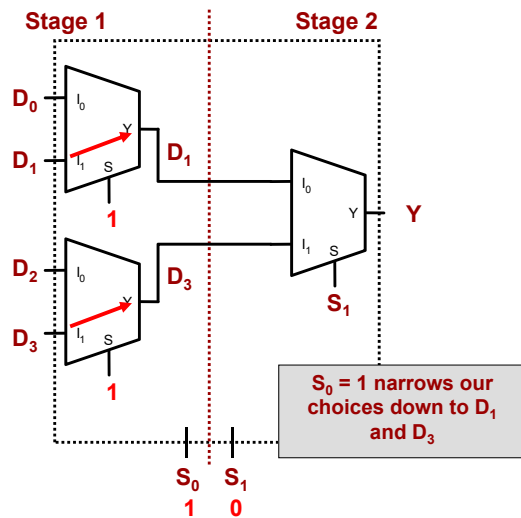


## Building a 4-to-1 Mux

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

Walk through an example:

$$S_1 S_0 = 01$$

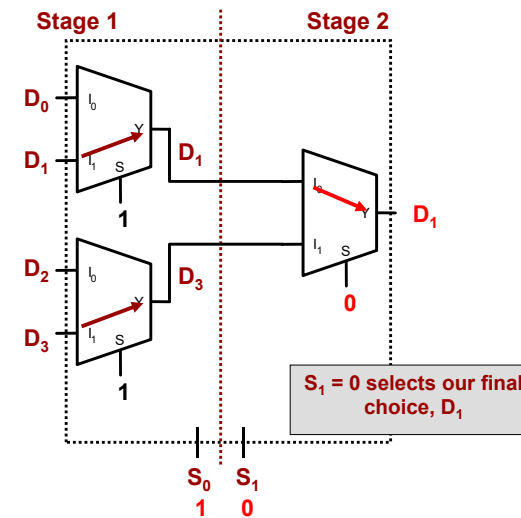


## Building a 4-to-1 Mux

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

Walk through an example:

$$S_1 S_0 = 01$$



## Exercise

- Create a 3-to-1 mux using 2-to-1 muxes
  - Inputs: I0, I1, I2 and select bits S1, S0
  - Output: Y

