Introduction to Digital Logic

Lecture 9:

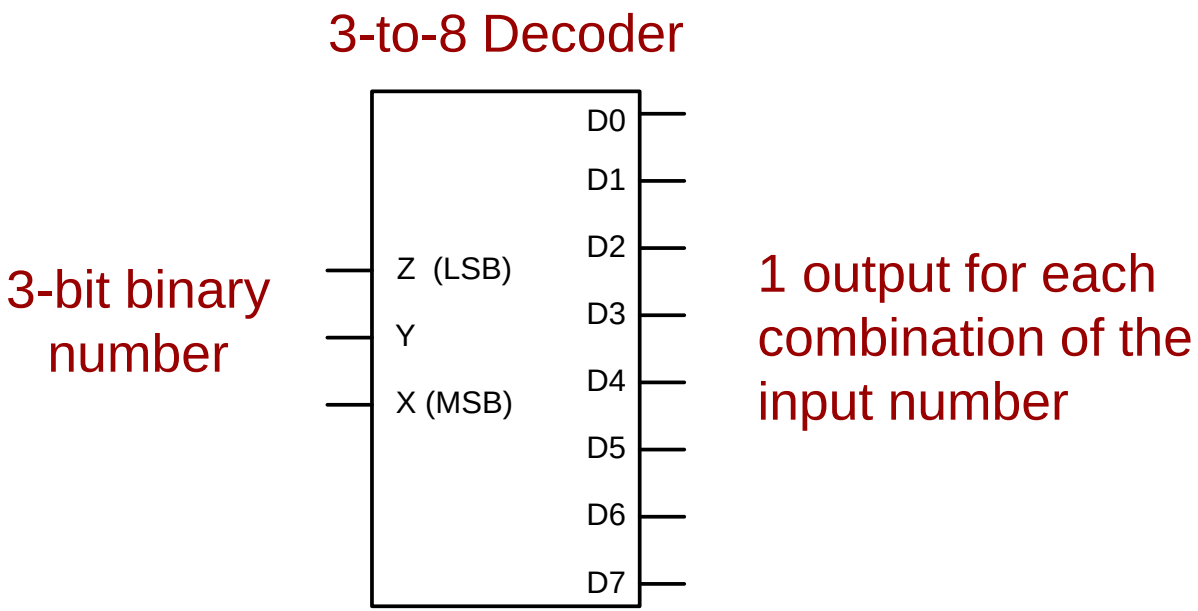
Implementing Logic Functions w/ Decoder, Multiplexers, and Memories

Logic Function Synthesis

- Given a function description as a T.T. or canonical form, how can we arrive at a circuit implementation or equation (i.e. perform logic synthesis)?
- 5 methods to be discussed
  - Canonical sum/product + Simplification w/ theorems
  - Karnaugh Maps
  - Decoders + 1 gate per output
  - Multiplexers
  - Memories (used as Look-Up-Tables [LUT's])

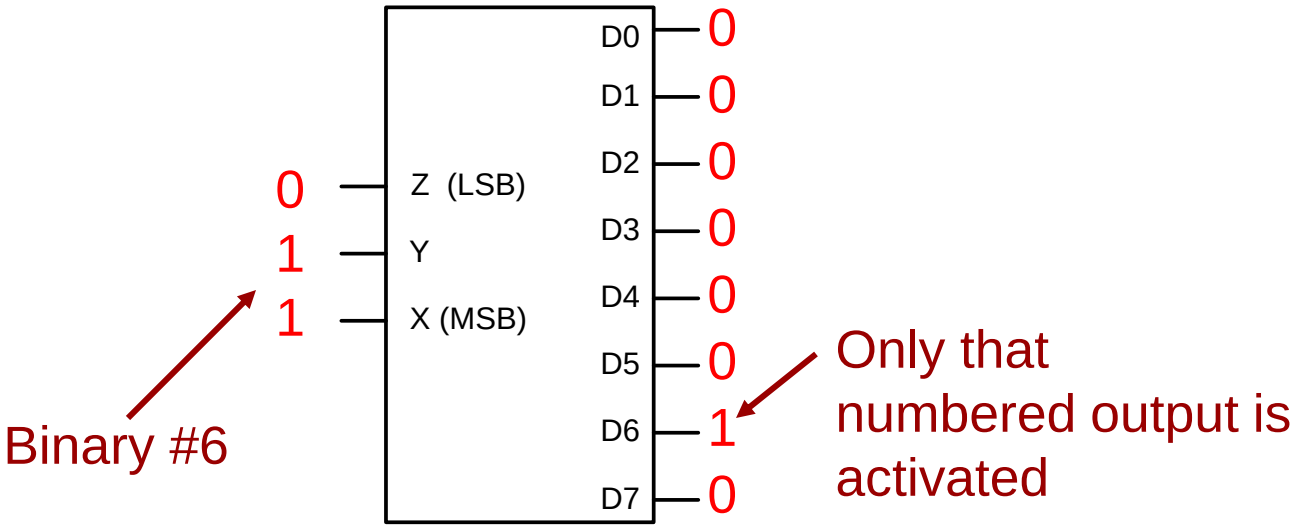
Decoders

- 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 EVERY input combination (i.e. 2^n outputs)



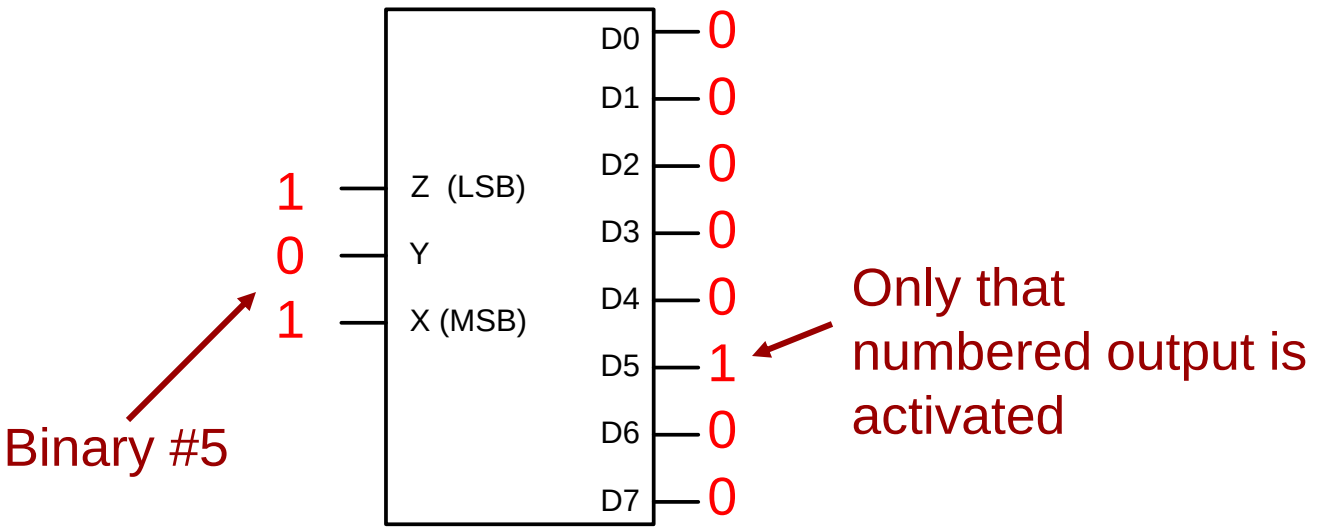
Decoders

- 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’)



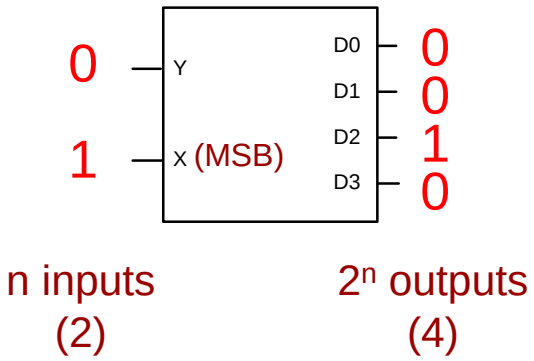
Decoders

- 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’)

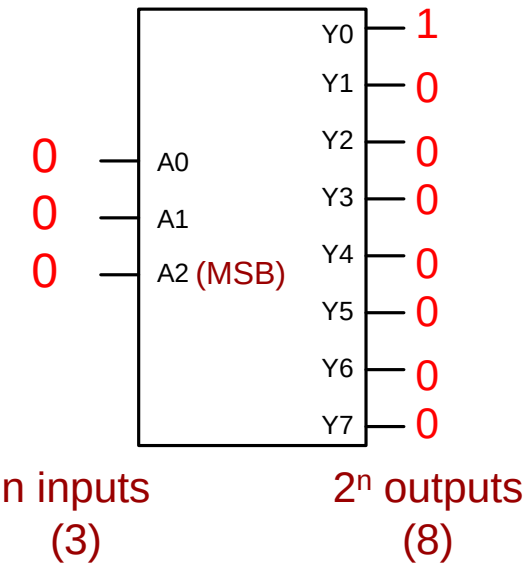


Decoder Sizes

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

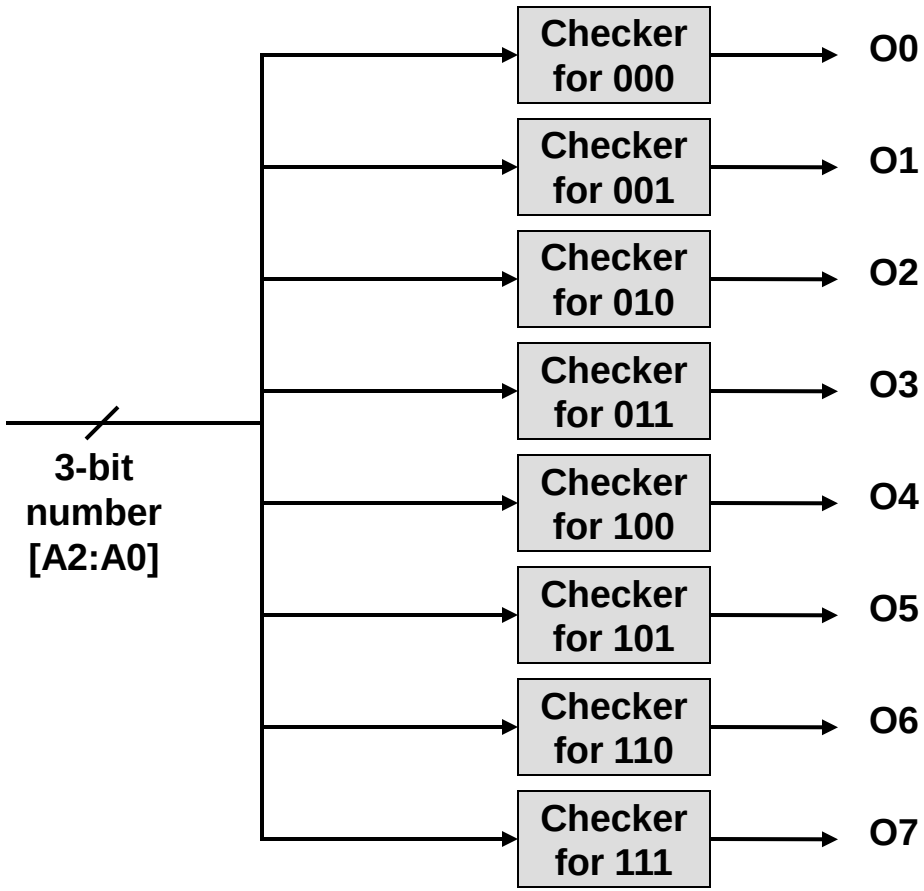


2-to-4 Decoder



3-to-8 Decoder

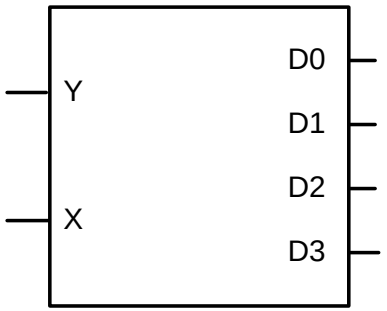
Building Decoders

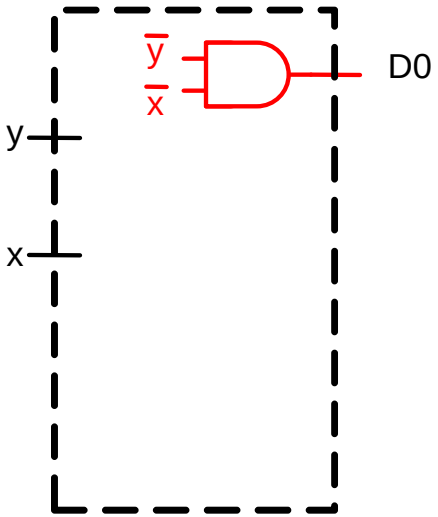


Building Decoders

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

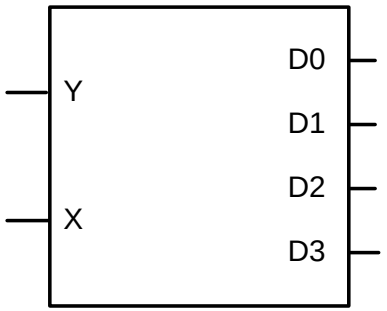
D0 = X'•Y'





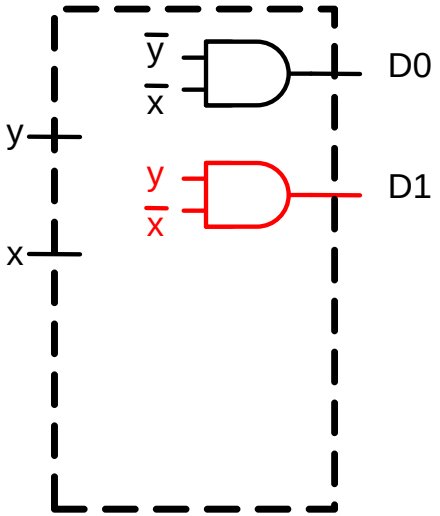
Building Decoders

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



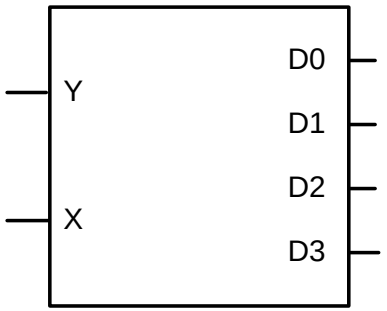
$D0 = X' \cdot Y'$

$D1 = X' \cdot Y$



Building Decoders

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

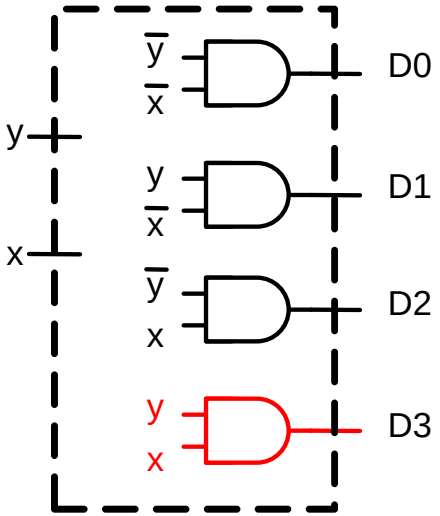


D0 = X'•Y'

D1 = X'•Y

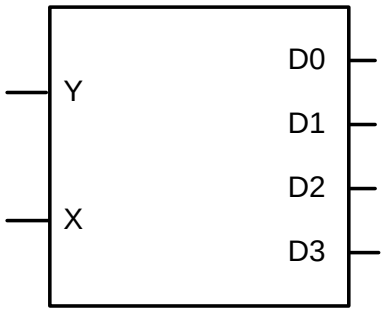
D2 = X•Y'

D3 = X•Y



Building Decoders

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



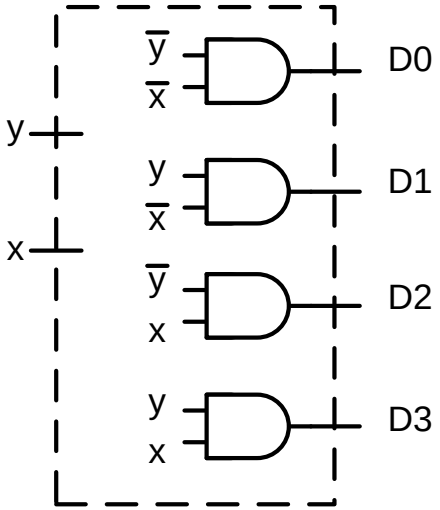
$D0 = X' \cdot Y' = m_0$

$D1 = X' \cdot Y = m_1$

$D2 = X \cdot Y' = m_2$

$D3 = X \cdot Y = m_3$

Notice output n is just minterm, m_n

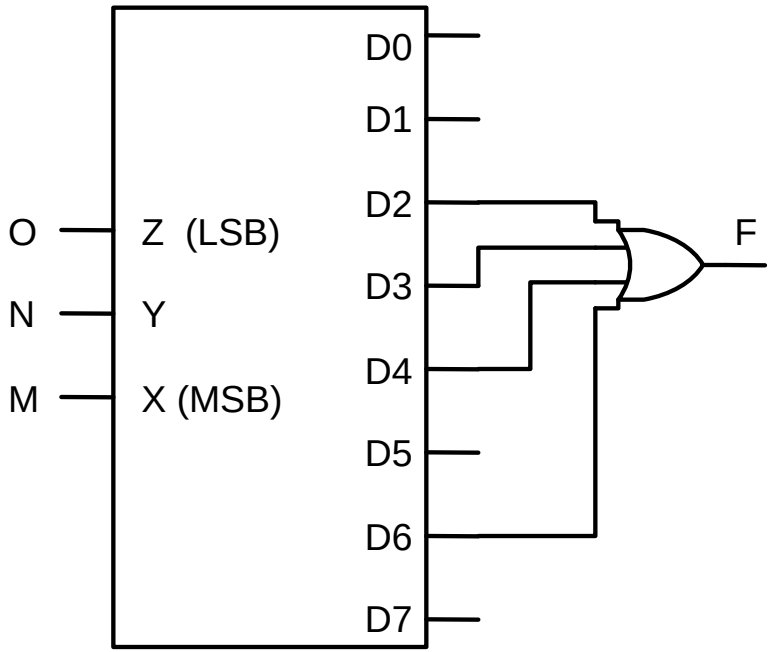


Building Decoders

- All decoders are built by simply implementing minterms for each combination of the inputs
- This means a decoder is really just AND gates with inverters!

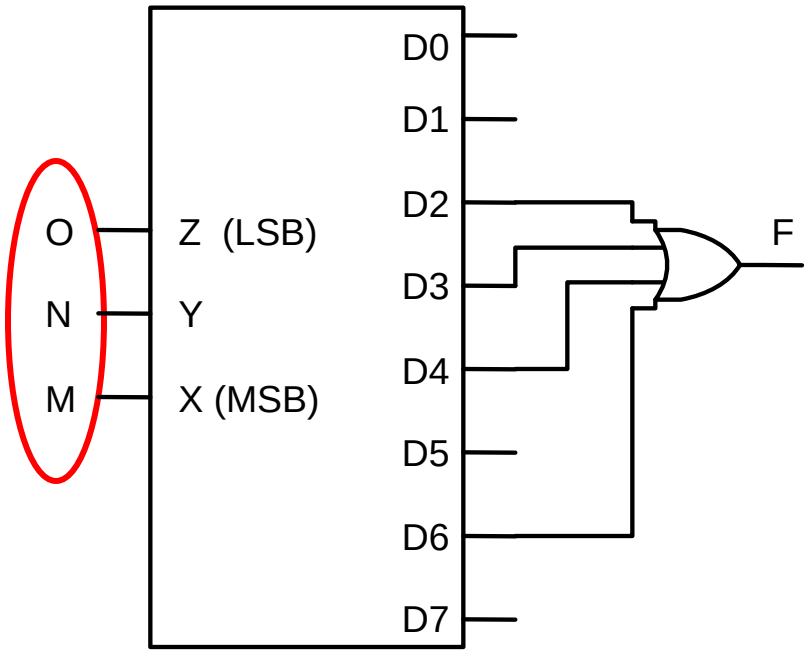
Implementing Logic Functions

- $F = \Sigma_{MNO}(2,3,4,6)$
  $= m_2 + m_3 + m_4 + m_6$
- Since decoders are just minterm generators, just OR together the appropriate minterms



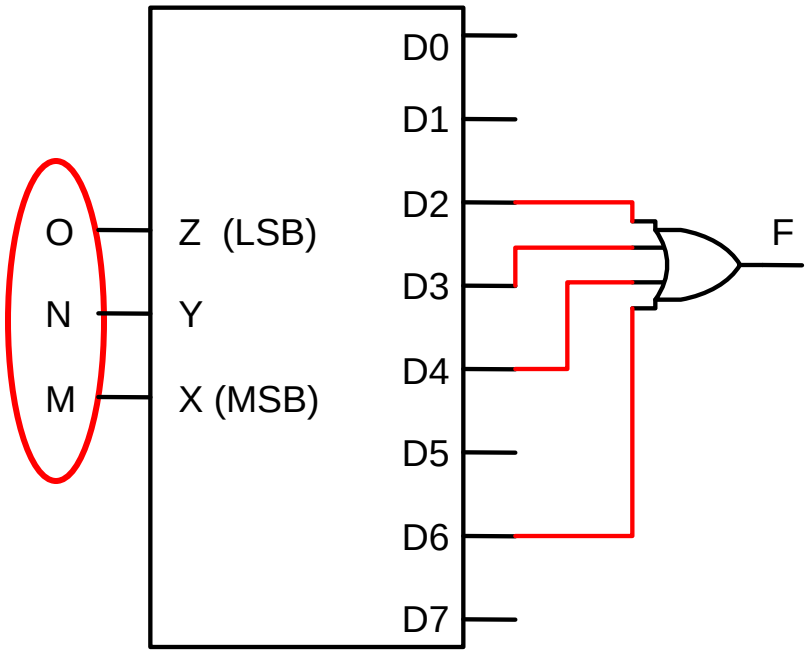
Implementing Logic Functions

- $F = \Sigma_{MNO}(2,3,4,6)$
  $= m_2 + m_3 + m_4 + m_6$
- Connect the input variables M,N,O to the inputs of the decoder to produce the minterms



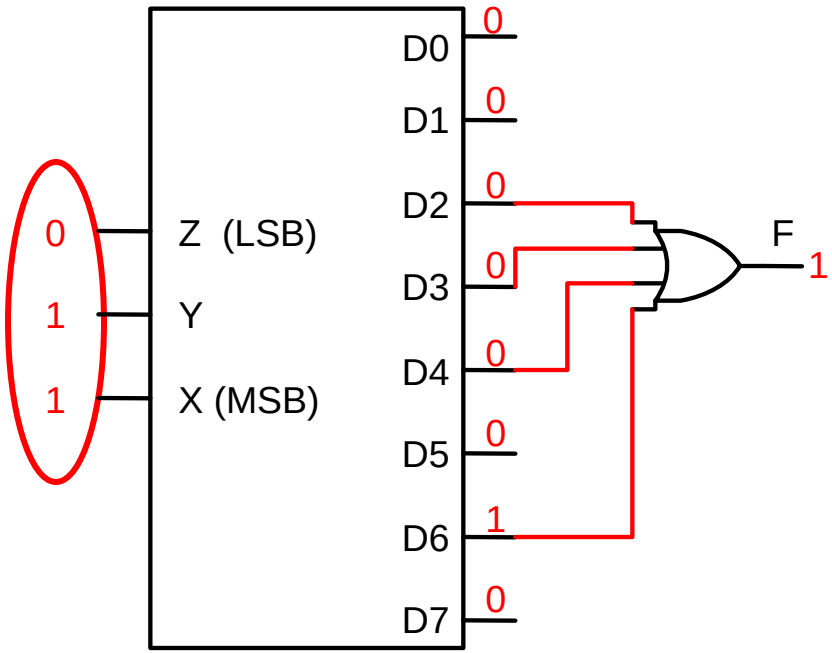
Implementing Logic Functions

- $F = \Sigma_{MNO}(2,3,4,6)$
  $= m_2 + m_3 + m_4 + m_6$
- Connect the input variables M,N,O to the inputs of the decoder to produce the minterms
- OR together where F = 1



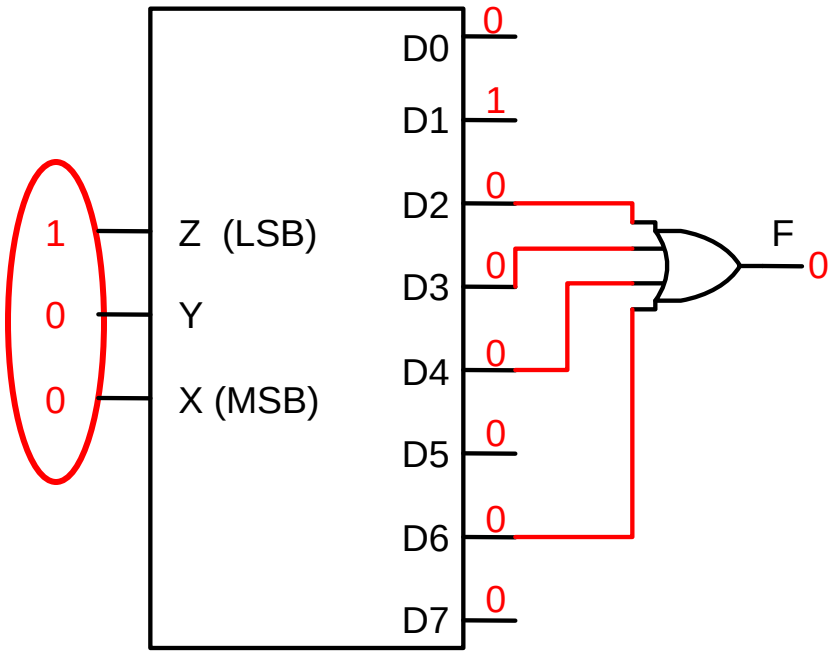
Implementing Logic Functions

- $F = \Sigma_{MNO}(2,3,4,6)$
  $= m_2 + m_3 + m_4 + m_6$
- Connect the input variables M,N,O to the inputs of the decoder to produce the minterms
- OR together where F = 1
- Test it MNO = 110



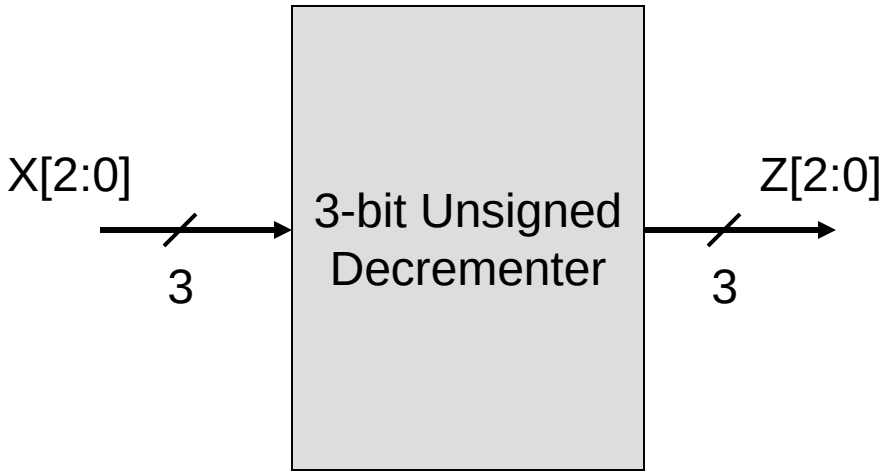
Implementing Logic Functions

- $F = \Sigma_{MNO}(2,3,4,6)$
  $= m_2 + m_3 + m_4 + m_6$
- Connect the input variables M,N,O to the inputs of the decoder to produce the minterms
- OR together where F = 1
- Test it MNO = 110
- Test it MNO = 001



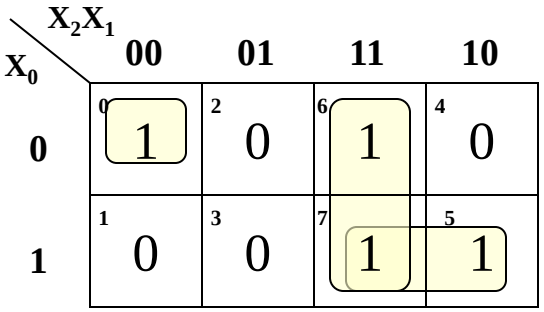
Designing Circuits w/ K-Maps

- Given a description…
  - Block Diagram
  - Truth Table
  - K-Map for each output bit (each output bit is a separate function of the inputs)
- 3-bit unsigned decrementer (Z = X-1)
  - If X[2:0] = 000 then Z[2:0] = 111, etc.

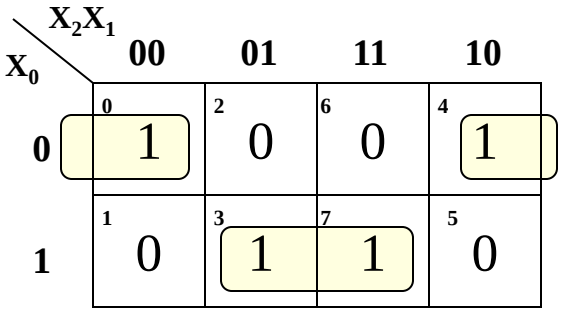


3-bit Number Decrementer

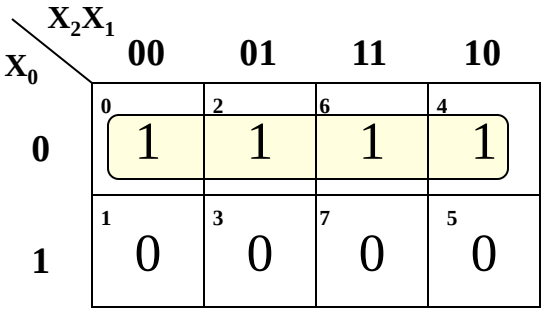
X_2	X_1	X_0	Z_2	Z_1	Z_0
0	0	0	**1**	**1**	**1**
0	0	1	**0**	**0**	**0**
0	1	0	**0**	**0**	**1**
0	1	1	**0**	**1**	**0**
1	0	0	**0**	**1**	**1**
1	0	1	**1**	**0**	**0**
1	1	0	**1**	**0**	**1**
1	1	1	**1**	**1**	**0**



$Z_2 = X_2X_0 + X_2X_1 + X_2'X_1'X_0'$



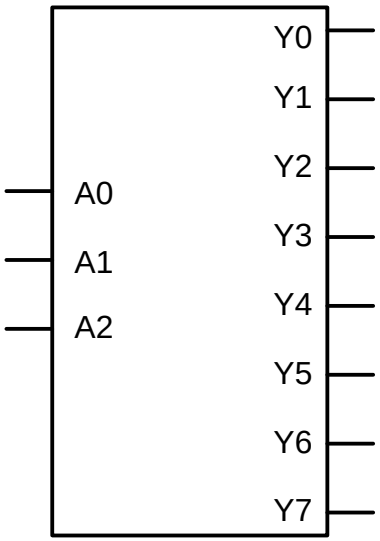
$Z_1 = X_1'X_0' + X_1X_0$



$Z_0 = X_0'$

3-bit Decrementer

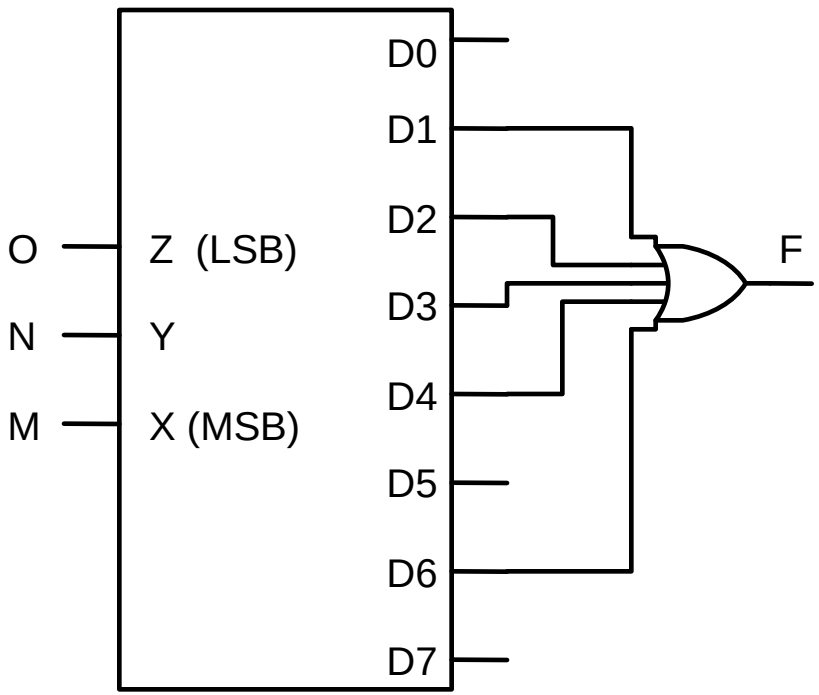
X_2	X_1	X_0	Z_2	Z_1	Z_0
0	0	0	**1**	**1**	**1**
0	0	1	**0**	**0**	**0**
0	1	0	**0**	**0**	**1**
0	1	1	**0**	**1**	**0**
1	0	0	**0**	**1**	**1**
1	0	1	**1**	**0**	**0**
1	1	0	**1**	**0**	**1**
1	1	1	**1**	**1**	**0**



- When we have many outputs that are a function of the same inputs, it may be smaller to use a decoder rather than separate logic for each output

Implementing Logic Functions

- $F = \Sigma_{MNO}(1,2,3,4,6)$
- Requires 5-input OR gate

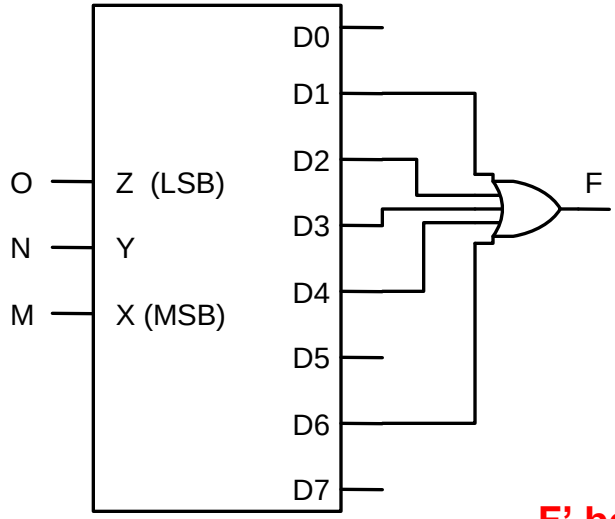


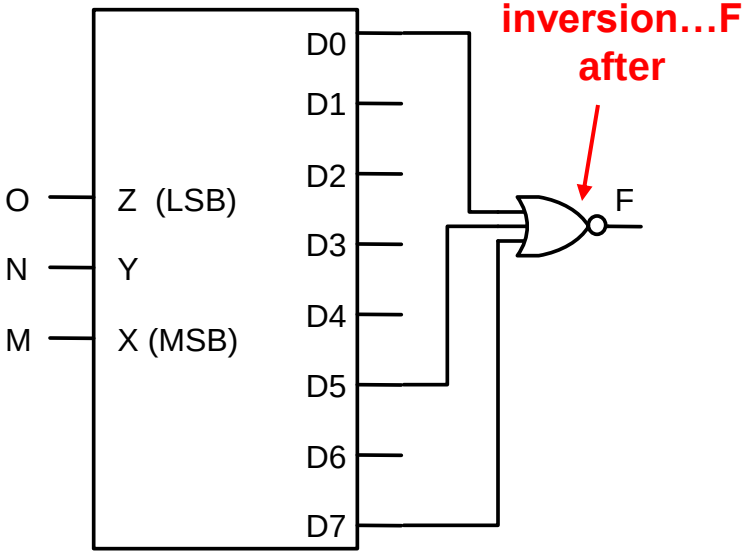
Implementing Logic Functions

- $F = \Sigma_{MNO}(1,2,3,4,6)$
- Requires 5-input OR gate…***too big!***

Implement F another way…

- $F' = \Sigma_{MNO}(0,5,7)$
- Requires a 3-input OR gate
- To get F, just add an inversion at the end
- OR becomes NOR





MULTIPLEXERS

Single Variable Theorem (T1)

X+0 = X (T1)

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

OR

Hold Y constant

X•1 = X (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 (T2)

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

OR

Hold Y constant

X•0 = 0 (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…

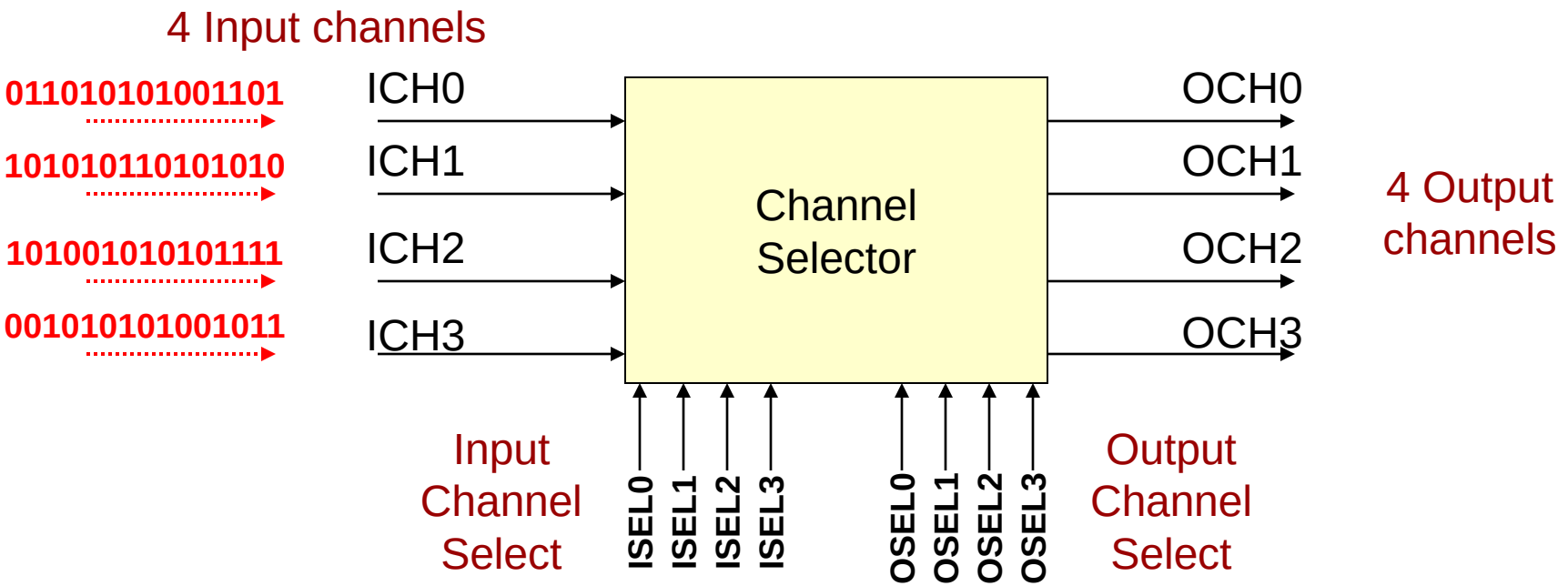
"1 OR anything equals 1"

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

"0 AND anything equals 0"

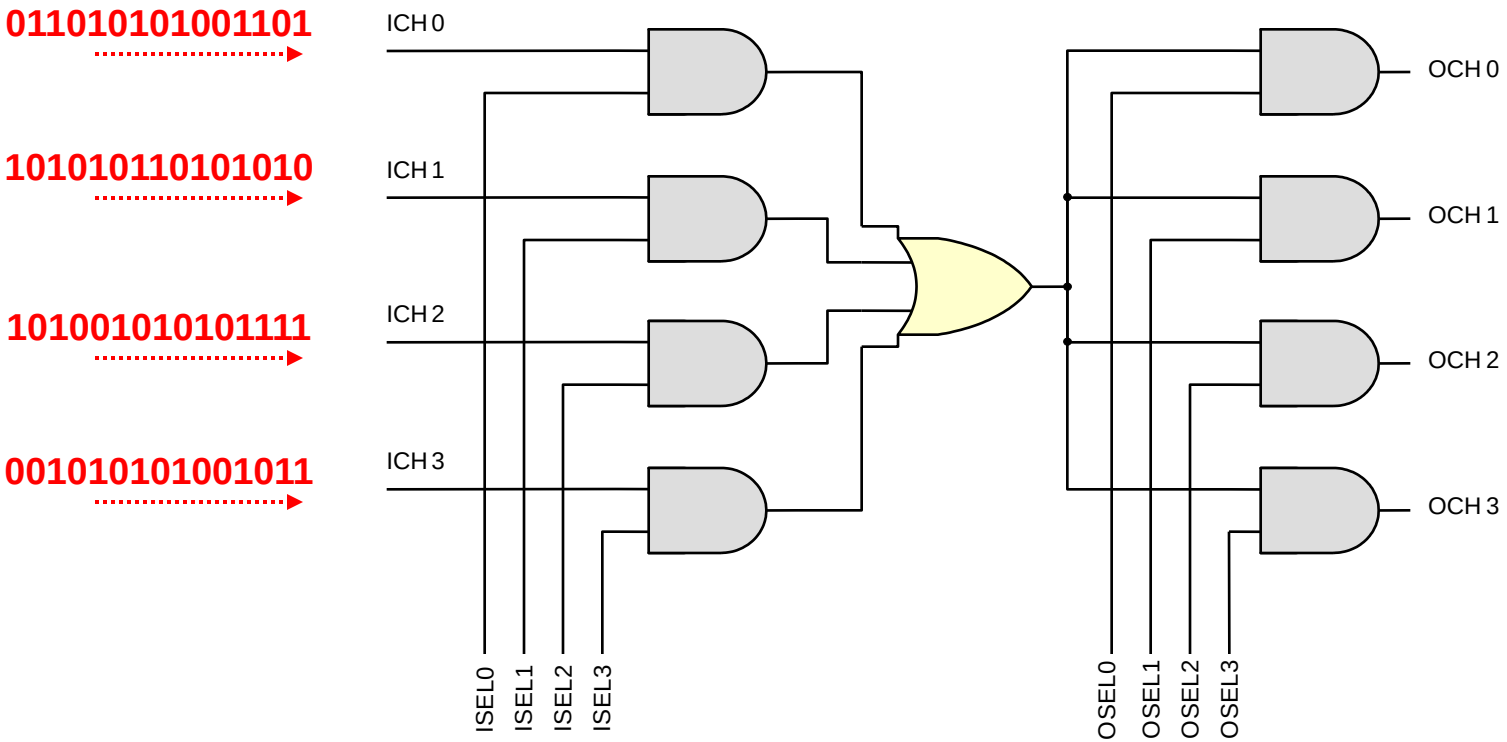
Application: Channel Selector

- 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



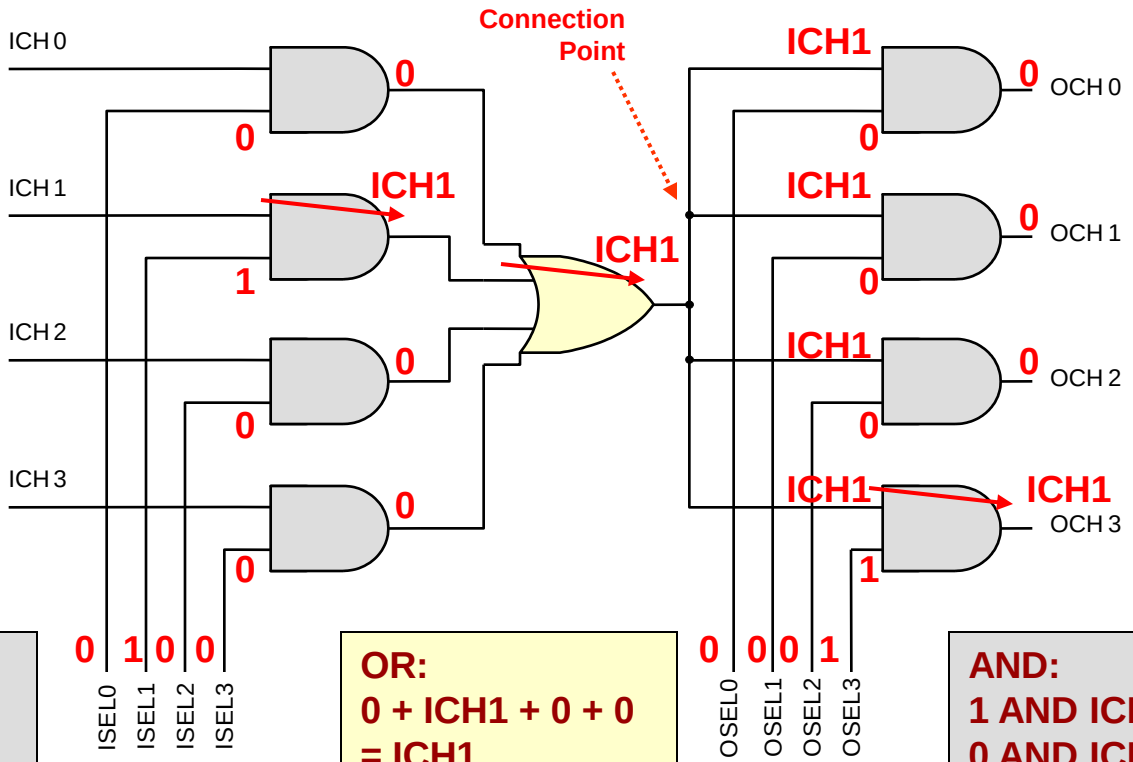
Application: Steering Logic

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



Application: Steering Logic

- 1st 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)
- 2nd Level of AND gates passes the channel to only the selected output



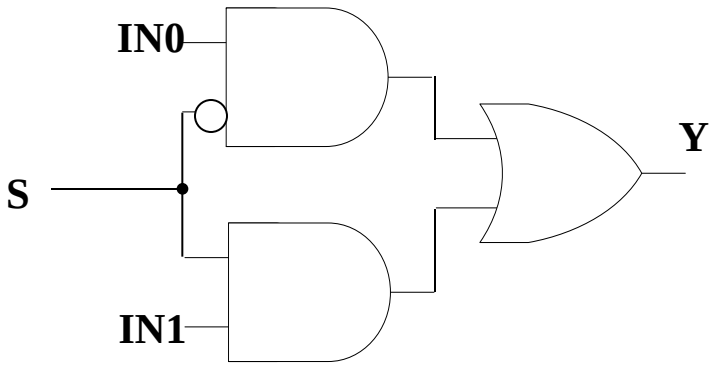
AND:
1 AND ICHx = ICHx
0 AND ICHx = 0

OR:
0 + ICH1 + 0 + 0 = ICH1

AND:
1 AND ICH1 = ICH1
0 AND ICH1 = 0

Your Turn

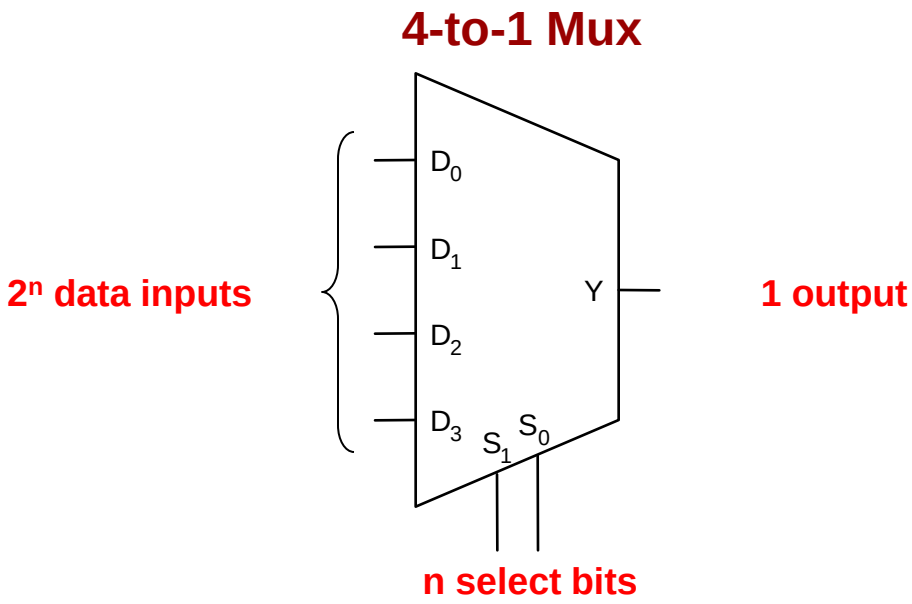
- Build a circuit that takes 3 inputs: S, IN0, IN1 and outputs a single bit Y.
- It's functions should be:
  - If S = 0, Y = IN0 (IN0 passes to Y)
  - If S = 1, Y = IN1 (IN1 passes to Y)



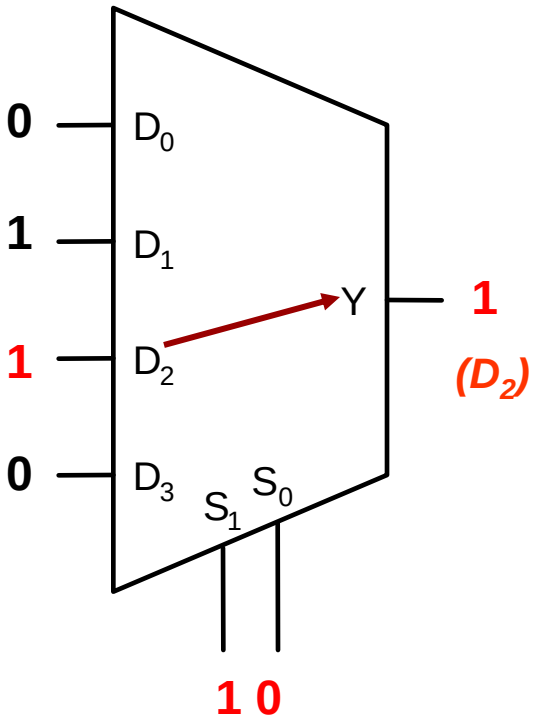

```
if(s==0)
  Y =  IN0
else
  Y =  IN1
```

Multiplexers

- Along with adders, multiplexers are most used building block
- 2^n data inputs, n select bits, 1 output
- A multiplexer ("mux" for short) selects one data input and passes it to the output



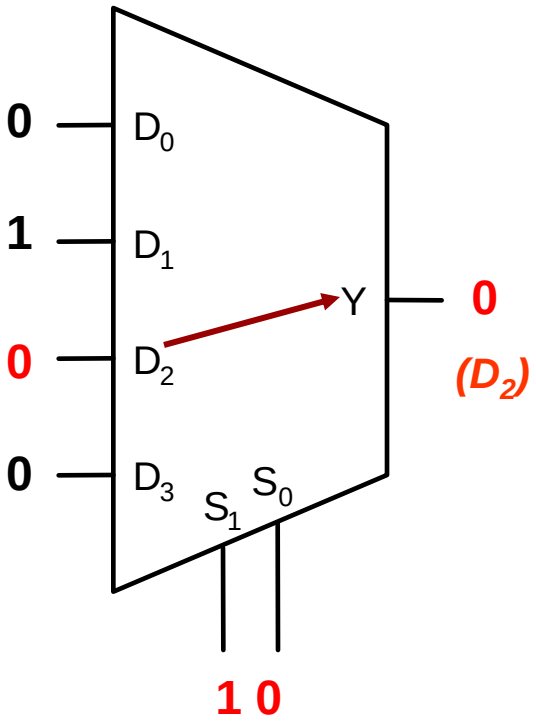
Multiplexers



② **Thus, D2 is selected and passed to the output**

① **Select bits = $10_2 = 2_{10}$.**

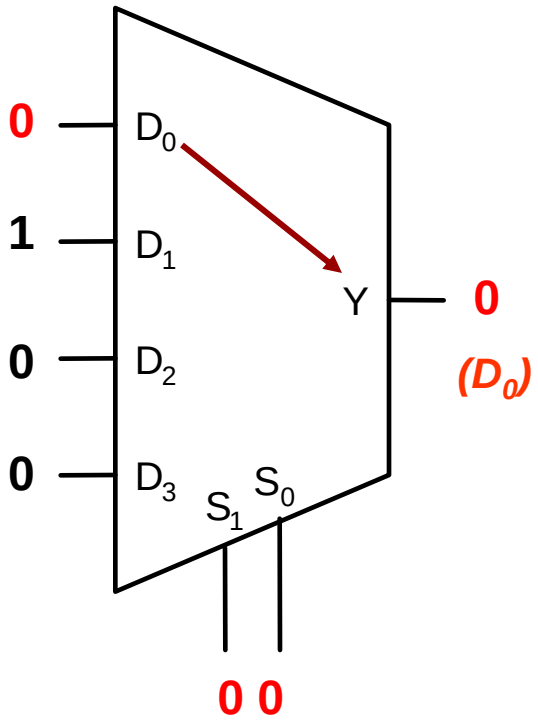
Multiplexers



D2 is being selected and passed.
So if it changes the output changes as well.

Multiplexers

② **Thus, D0 is selected and passed to the output**

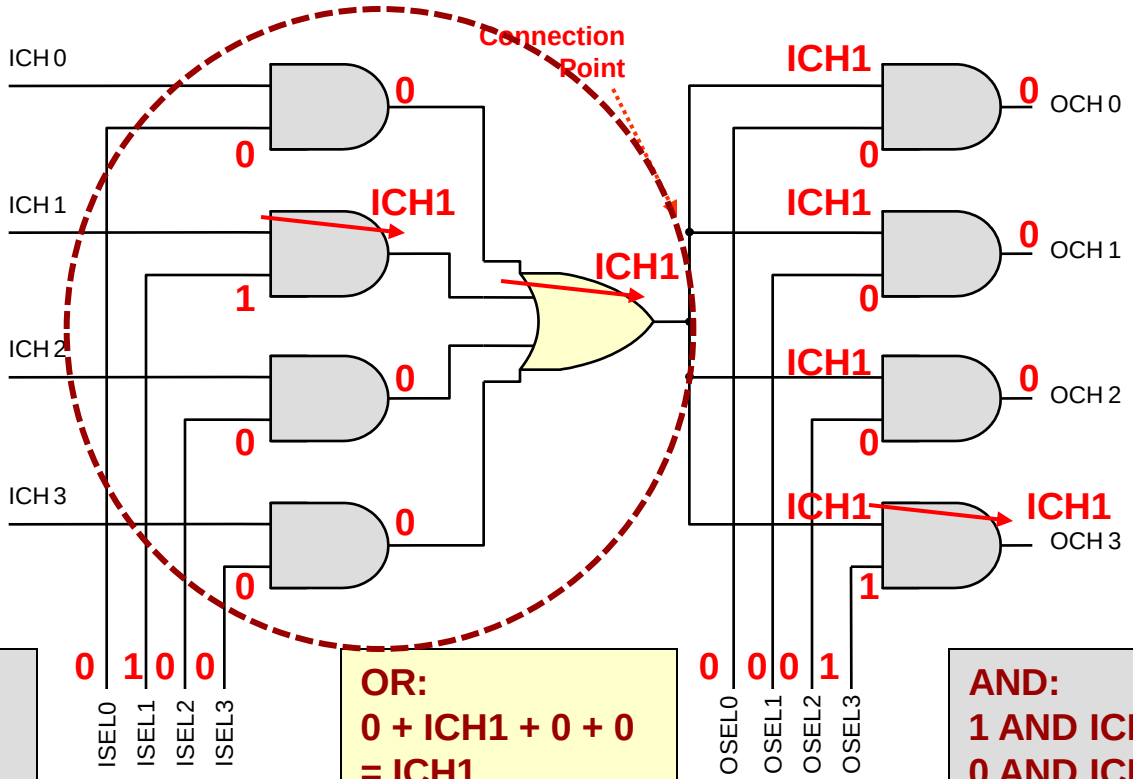



```
if( S1S0 == 00)
  Y =  D0
else if(S1S0 == 01)
  Y =  D1
else if(S1S0 == 10)
  Y =  D2
else if(S1S0 == 11)
  Y =  D3
```

① **Select bits = $00_2 = 0_{10}$.**

Application: Steering Logic

- 1st 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)
- 2nd Level of AND gates passes the channel to only the selected output



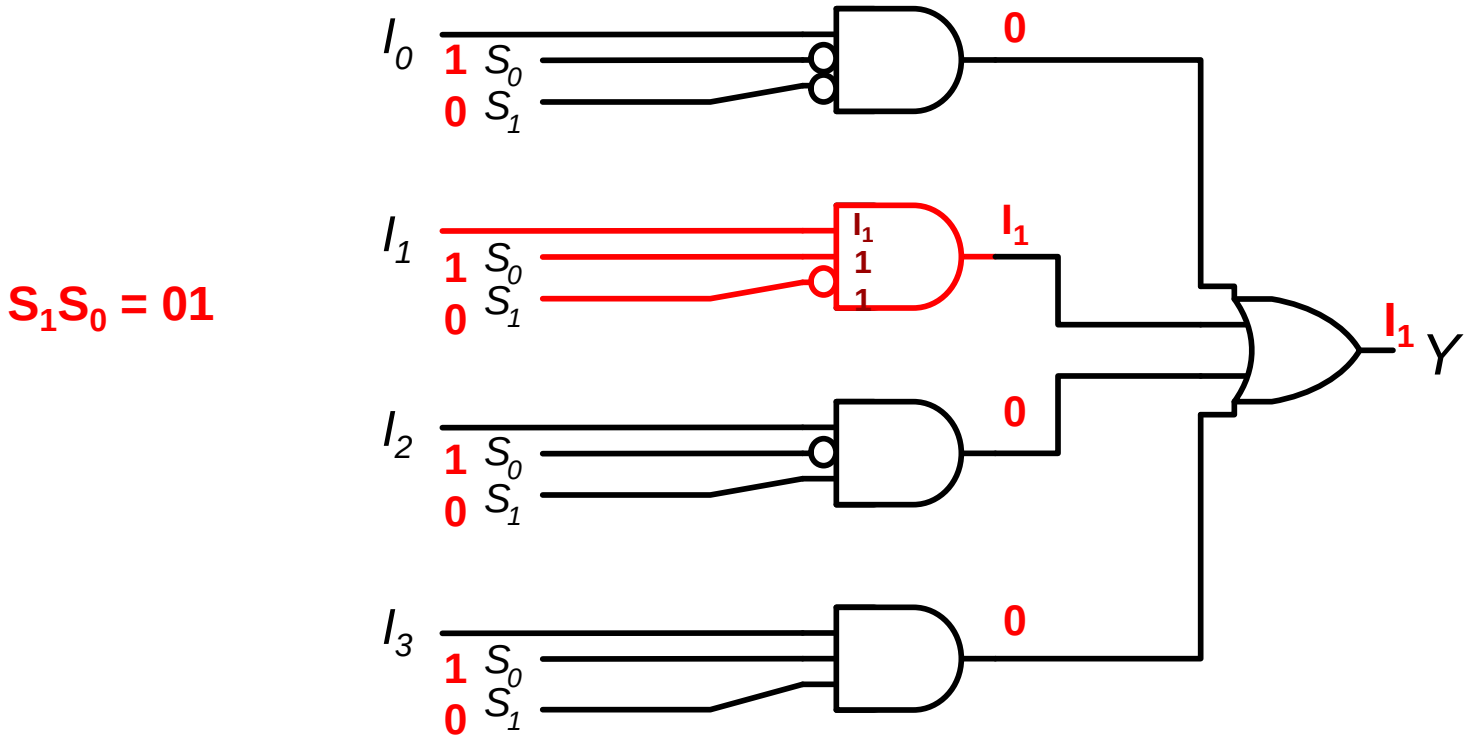
AND:
1 AND ICHx = ICHx
0 AND ICHx = 0

OR:
0 + ICH1 + 0 + 0 = ICH1

AND:
1 AND ICH1 = ICH1
0 AND ICH1 = 0

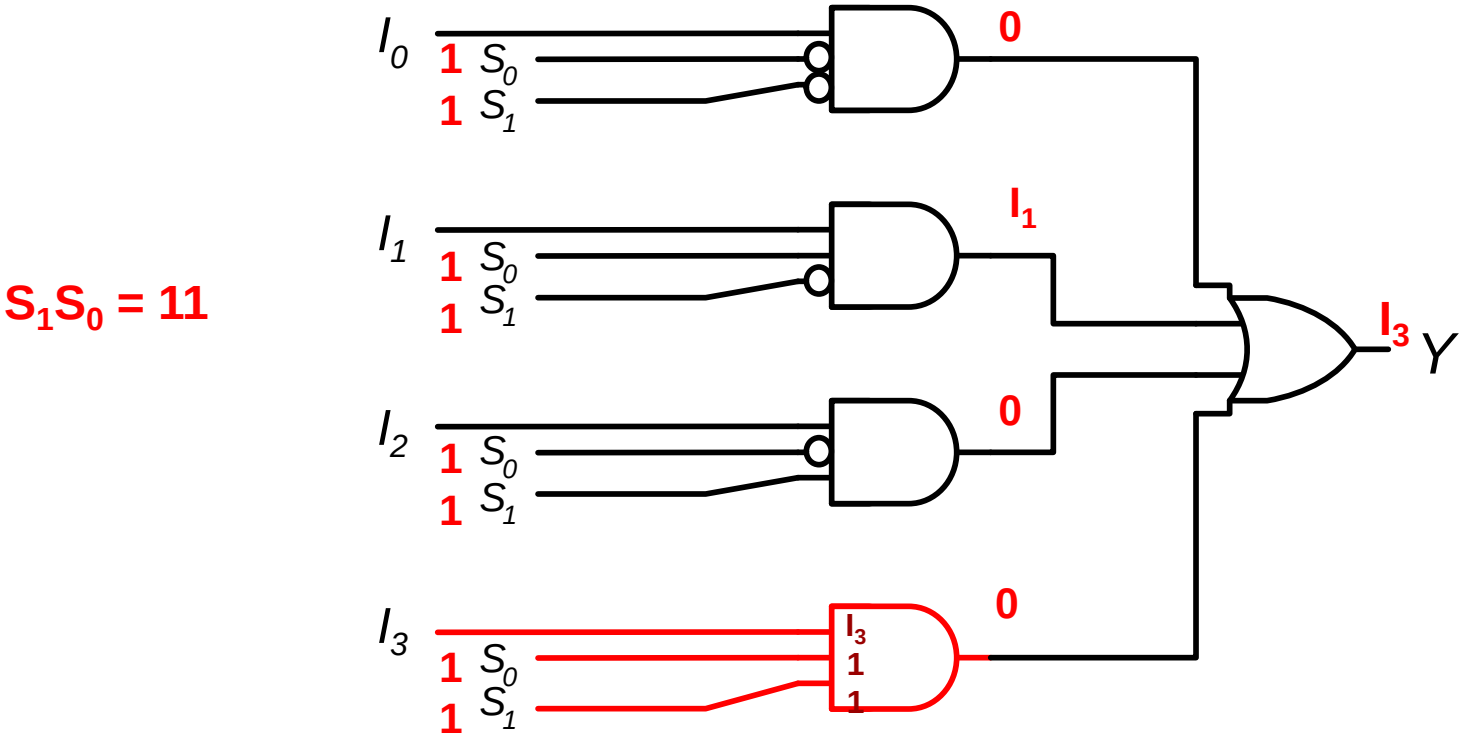
Building a Mux

- 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.



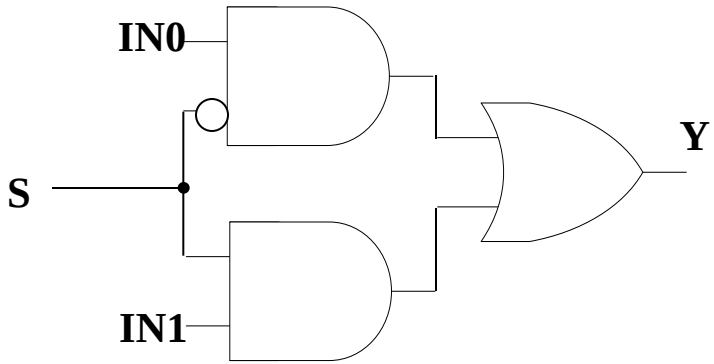
Simplify This

- Given F(x,y,z) = x'yz + y'z',
  F(0,y,z) =

  F(1,y,z) =

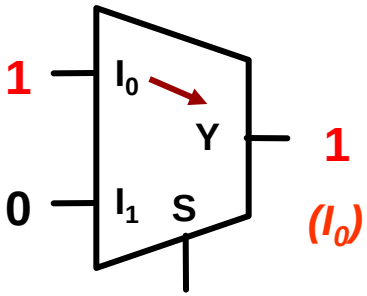
- Given G(a,b,c,d) = bd' + ab'cd + ac'd'
  G(1,1,c,d) =

2-to-1 Mux

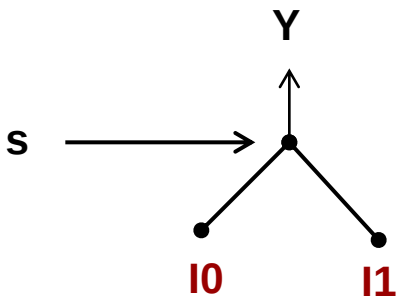



```
if(s==0)
  Y =  IN0
else
  Y =  IN1
```

② **Thus, I0 is selected and passed to the output**



1-bit wide 2-to-1 mux



① **Select bits = 0**

3-bit Prime Number Function

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

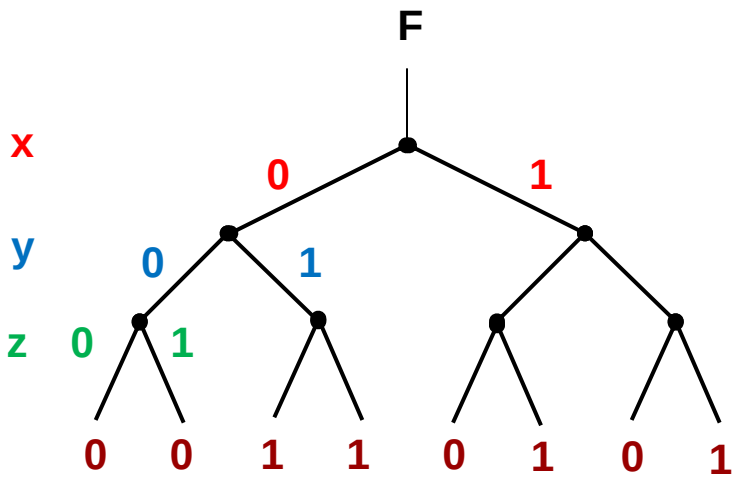
Primes between 0-7

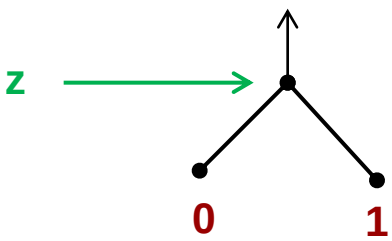
X	Y	Z	P
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

OFF-set

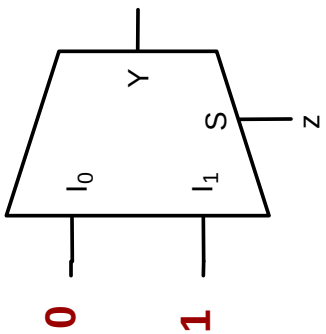
ON-set

Truth Table






```
if(z==0)
  output 0
else
  output 1
```



Example

Function Implementation w/ Muxes

- Implementing a function using muxes relies is based on Shannon's expansion theorem which states:
  - $F(X_1,X_2,\ldots,X_n) = X_1' \bullet F(0,X_2,\ldots,X_n) + X_1 \bullet F(1,X_2,\ldots,X_n)$
  - X_1 can be pulled out of F if we substitute an appropriate constant and qualify it with X_1' or X_1
- Now recall a 2-to-1 mux can be built as:
  - $F = S' \bullet I_0 + S \bullet I_1$
  - Comparing the two equations, Shannon's theorem says we can use X_1 as our select bit to a 2-to-1 mux with $F(0,X_2,\ldots X_n)$ as input 0 of our mux and $F(1,X_2,\ldots,X_n)$ as input 1

Binary Decision Trees & Muxes

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

F(x,y,z)

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

F(0,y,z)

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

F(1,y,z)

Z	F
0	0
1	0

F(0,0,z)

Z	F
0	1
1	1

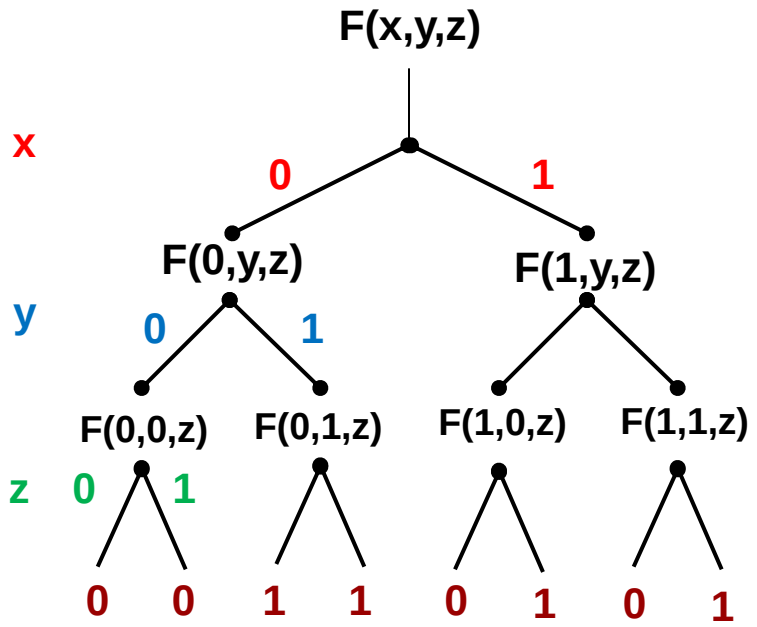
F(0,1,z)

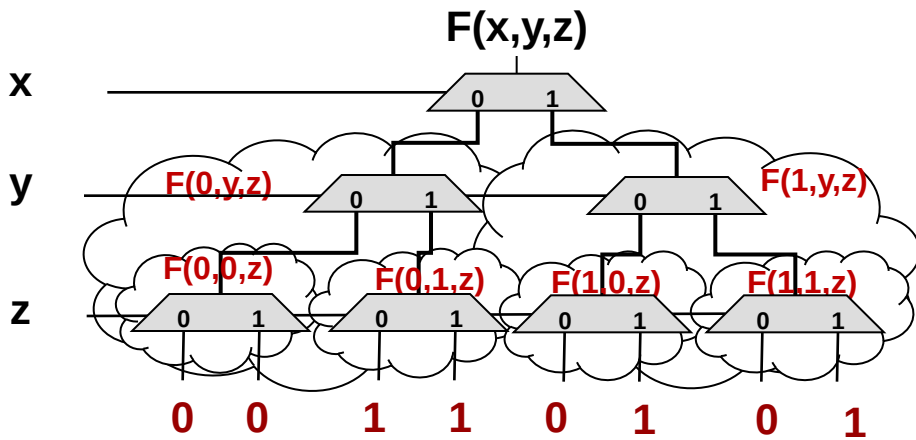
Z	F
0	0
1	1

F(1,0,z)

Z	F
0	0
1	1

F(1,1,z)





Splitting on X

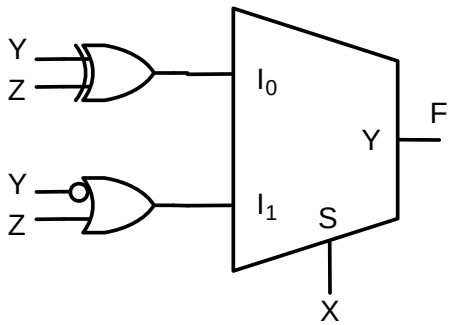
- We can use smaller muxes by breaking the truth table into fewer disjoint sets
  - **This increases the amount of logic at the inputs though**
- Break the truth table into groups based on some number (k) of MSB's
- For each group, describe F as a function of the n-k LSB's

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

y xor z

(y'z + yz')

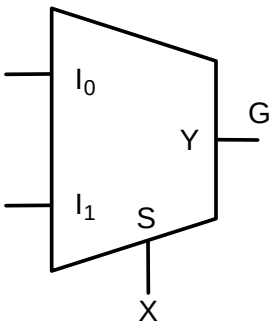
(y' + z)



Put the k MSB's on the selects

Implement G

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



Shannon's Theorem

- $F(X_1,X_2,\ldots,X_n) = X_1' \cdot F(0,X_2,\ldots,X_n) + X_1 \cdot F(1,X_2,\ldots,X_n)$
- Now recall a 2-to-1 mux can be built as:
  - $F = S' \cdot I_0 + S \cdot I_1$
  - Comparing the two equations, Shannon's theorem says we can use X_1 as our select bit to a 2-to-1 mux with $F(0,X_2,\ldots X_n)$ as input 0 of our mux and $F(1,X_2,\ldots,X_n)$ as input 1
- We can recursively apply Shannon's theorem to pull out more variables:
  - $F(X_1,X_2,\ldots,X_n) =$
    $X_1'X_2' \cdot F(0,0,\ldots,X_n) + X_1'X_2 \cdot F(0,1,\ldots,X_n) +$
    $X_1X_2' \cdot F(1,0,\ldots,X_n) + X_1X_2 \cdot F(1,1,\ldots,X_n) +$

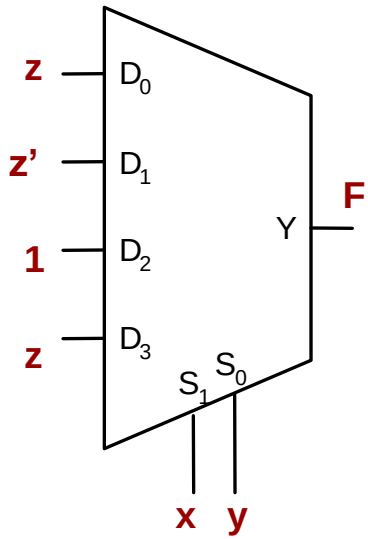
Additional Logic

- Muxes allow us to break a function into some number of smaller, disjoint functions
- Use MSB's to choose which small function we want
- By including the use of inverters we can use a mux with *n-1* select bits (given a function of n-var's)
- Break the truth table into groups of 2 rows
- For each group, put F in terms of: *z, z', 0, or 1*

X	Y	Z	F	
0	0	0	0	z
		1	1	
0	1	0	1	z'
		1	0	
1	0	0	1	1
		1	1	
1	1	0	0	z
		1	1	

$F_0(z)$ $F_1(z)$ $F_2(z)$ $F_3(z)$

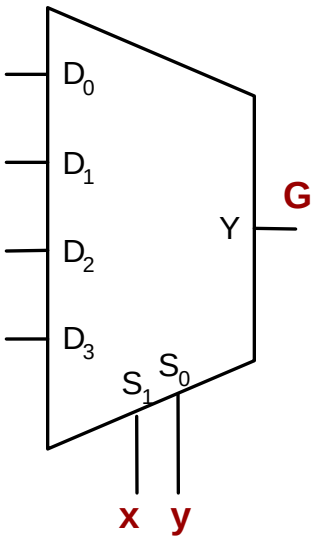
F(x,y,z) can be broken into several disjoint functions



Put the n-1 MSB's on the selects

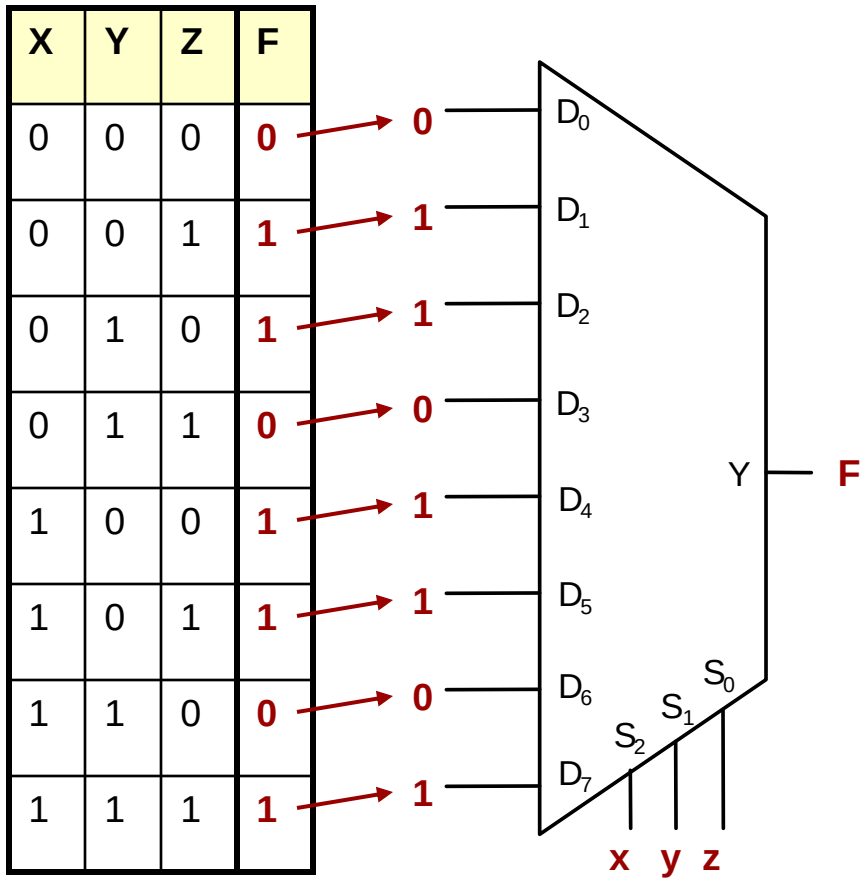
More Practice

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



Fundamental Method

- Connect the input variables to the select bits of the mux
- The output of the mux is the output of the function
- Whatever the output should be for each input value, attach that to the input of the mux

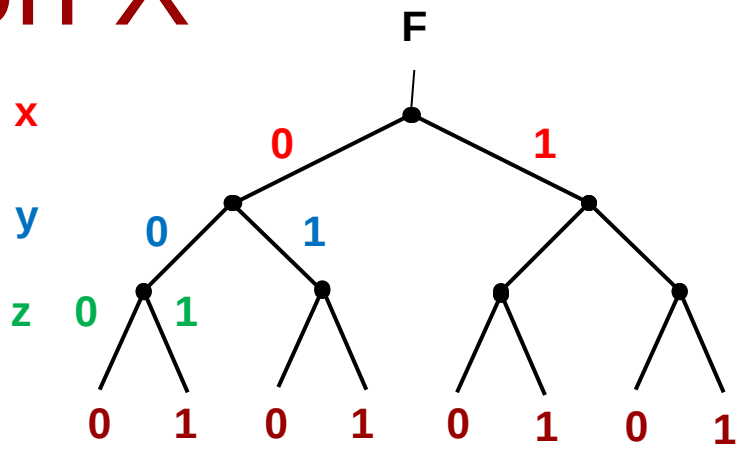


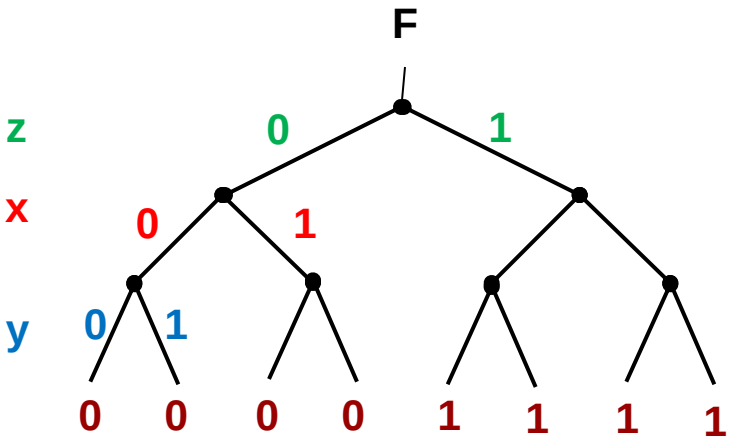
Splitting on X

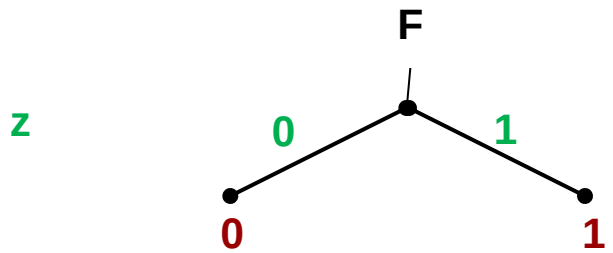
- We can always rearrange our variables if it helps make the function simpler to implement

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

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







Implementing Logic Functions

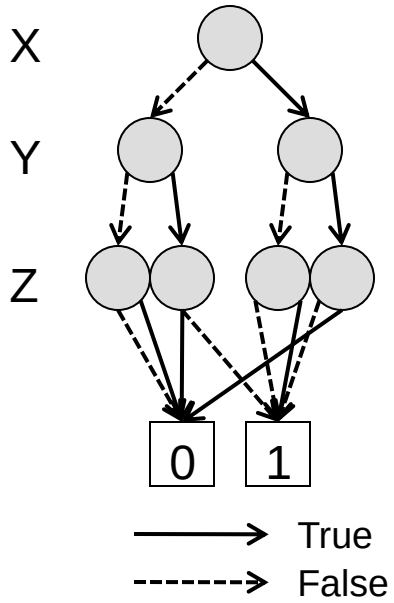
- We can use muxes to implement any arbitrary logic function
- Fundamental method
  - if function has n-input variables, need a mux w/ n-selects
  - requires no additional logic; only the mux
- By adding extra logic in front of the mux inputs we can use a smaller mux…

Binary Decision Diagram

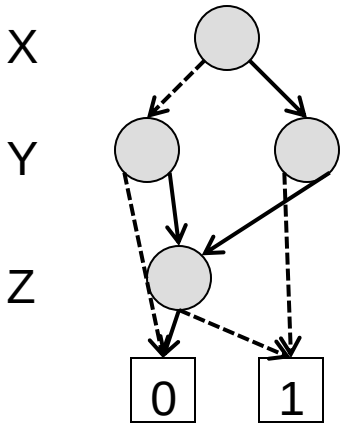
- Graph (binary tree) representation of logic function
- Vertex = Variable/Decision
- Edge = Variable value (T / F)

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

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

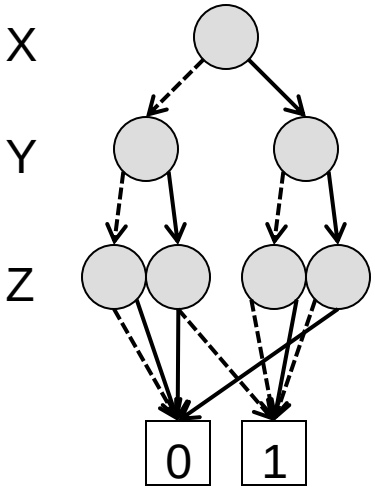


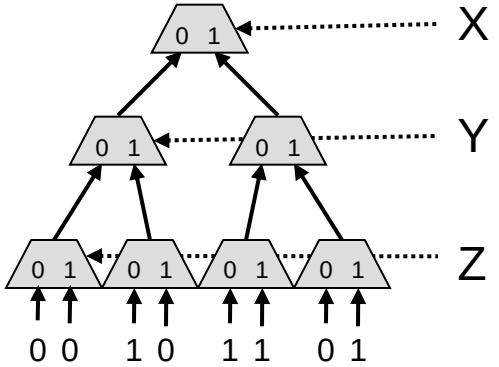
BDD for F

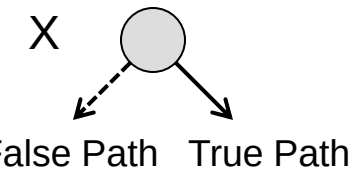


BDD's, Muxes, Shannon's Thm.

- Every node in the BDD is just a 2-to-1 mux
  - View it in the opposite direction (bottom to top)
- Really using Shannon's theorem







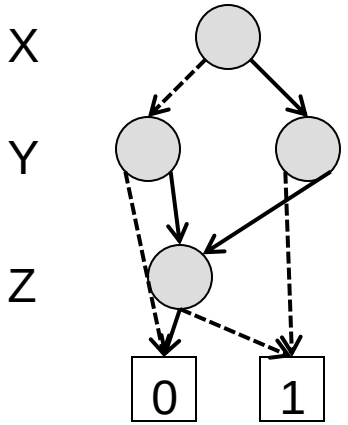
If X=0,
choose the 0 (False) path
If X=1,
choose the 1 (True) path

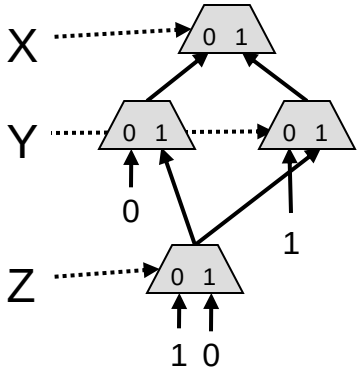
X	Y	Z	F
0	0	0	0
		1	0
	1	0	1
		1	0
1	0	0	1
		1	1
	1	0	0
		1	1

BDD for F

True

False





Using a LookUp-Table to implement a function

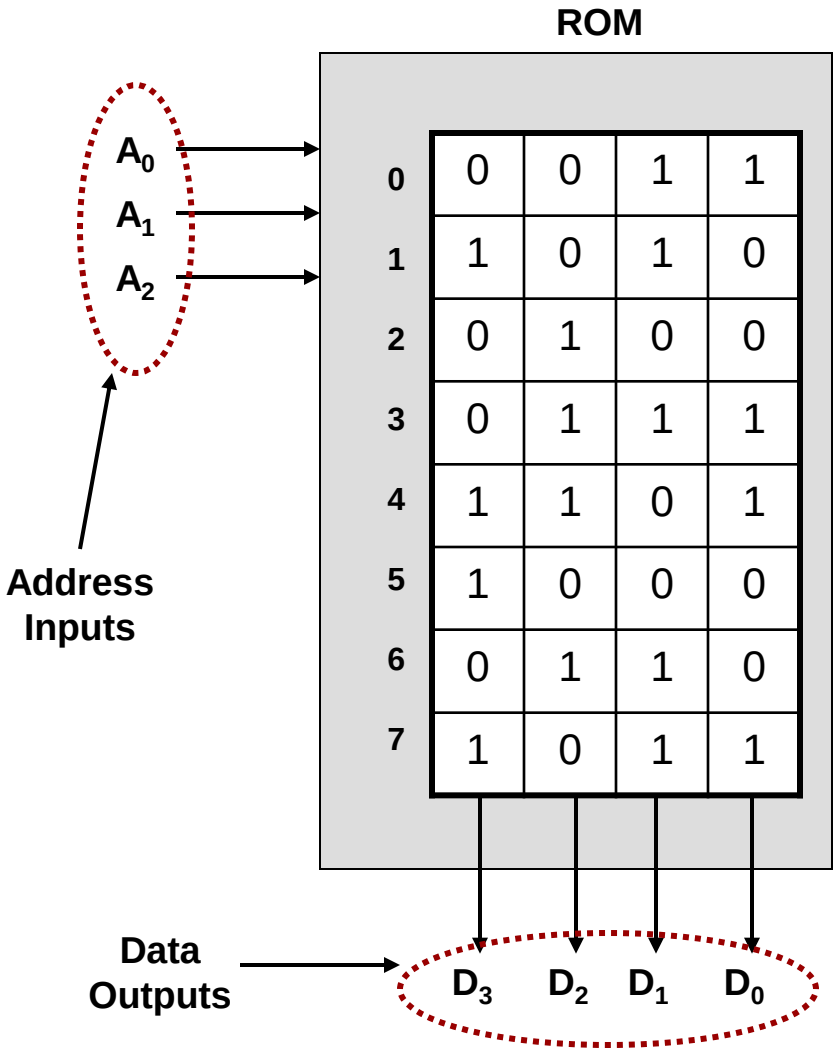
ROMS AND MEMORIES

Memories

- Memories store (write) and retrieve (read) data
  - Read-Only Memories (ROM's): Can only retrieve data (contents are initialized and then cannot be changed)
  - Read-Write Memories (RWM's): Can retrieve data and change the contents to store new data

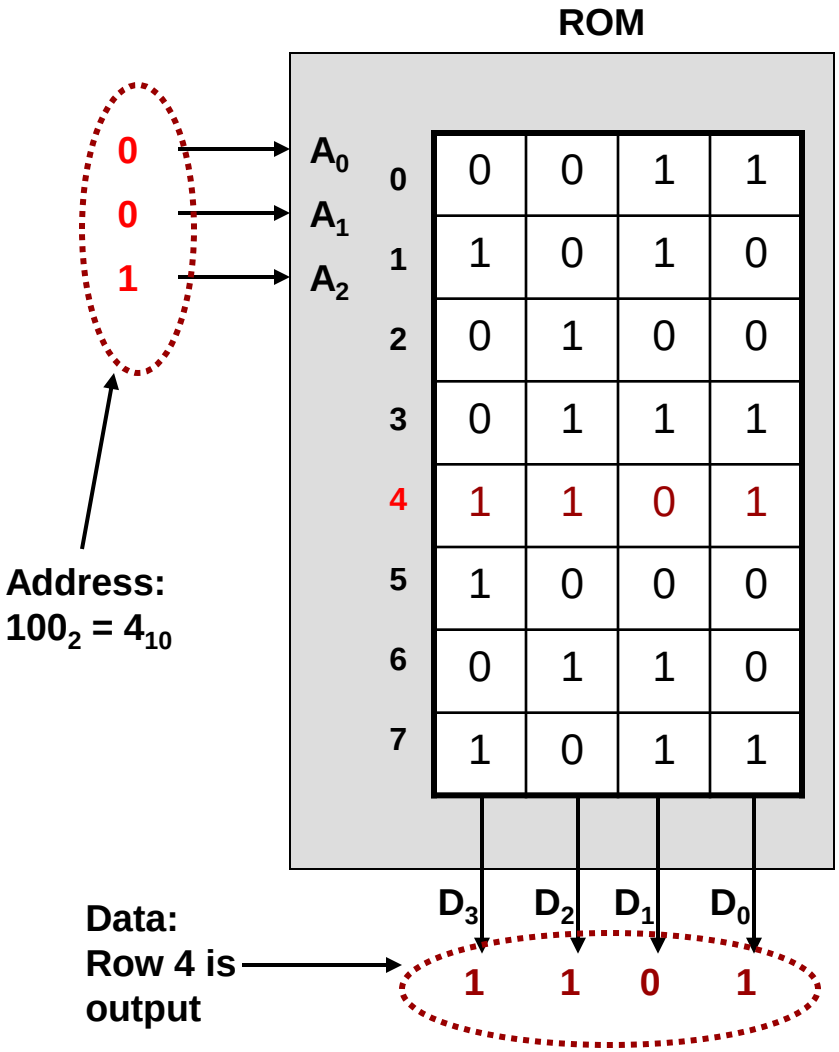
ROM's

- Memories are just tables of data with rows and columns
- When data is read, one entire row of data is read out
- The row to be read is selected by putting a binary number on the address inputs



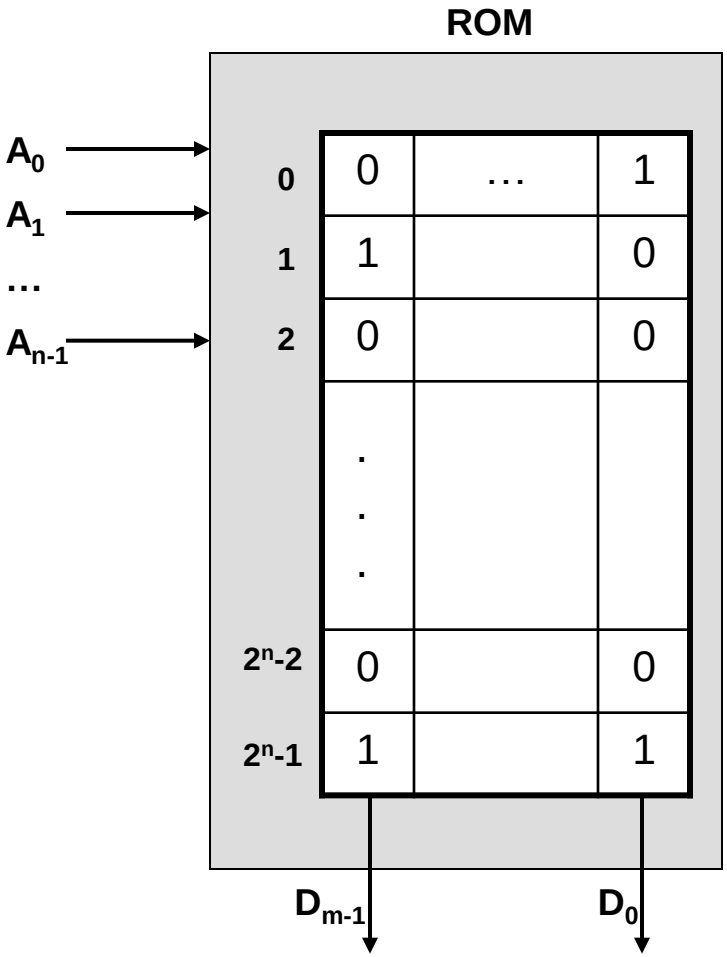
ROM's

- Memories are just tables of data with rows and columns
- When data is read, one entire row of data is read out
- The row to be read is selected by putting a binary number on the address inputs



ROM's

- ROM's are named by their dimensions:
  - Rows x Columns
- *n* rows and *m* columns => n x m ROM
- 2^n rows => n address bits (or k rows => $\log_2 k$ address bits)
- m cols. => m data outputs



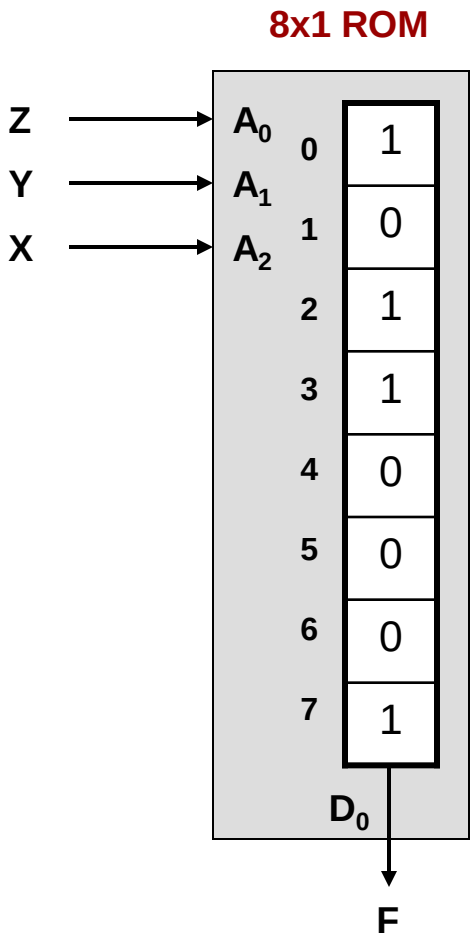
ROM's

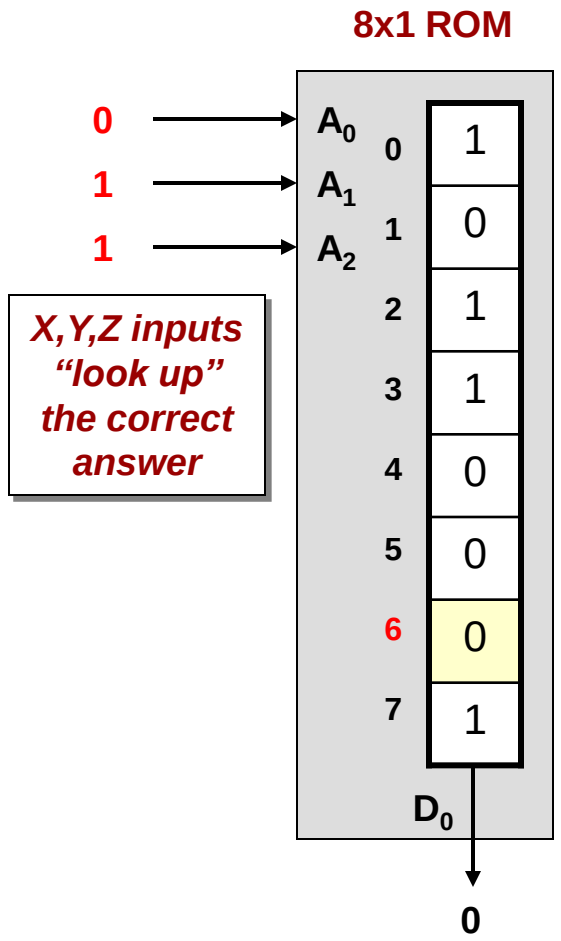
- One major application of ROM's is to use them as LUT's (Look-Up Tables) to implement logic functions
- Given a logic function use a ROM to hold all the possible answers and feed the inputs of the function to the address inputs to look-up the answer

Implementing Functions w/ ROM's

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

Arbitrary Logic Function



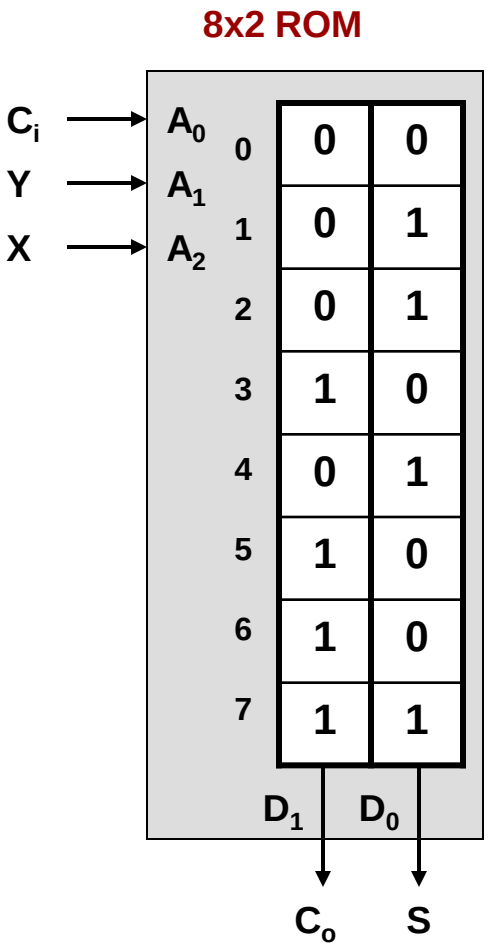


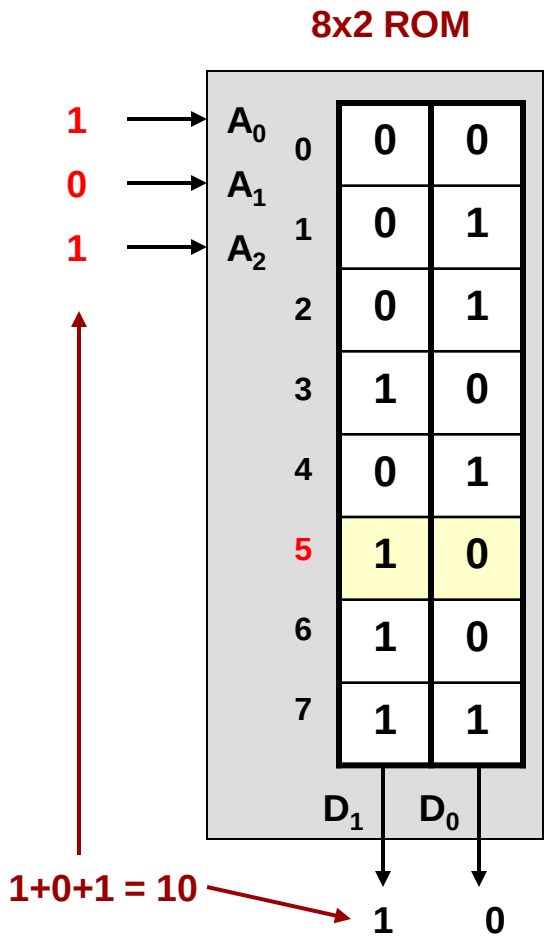
X,Y,Z inputs "look up" the correct answer

Implementing Functions w/ ROM's

X	Y	C_i	C_o	S
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

Full Adder

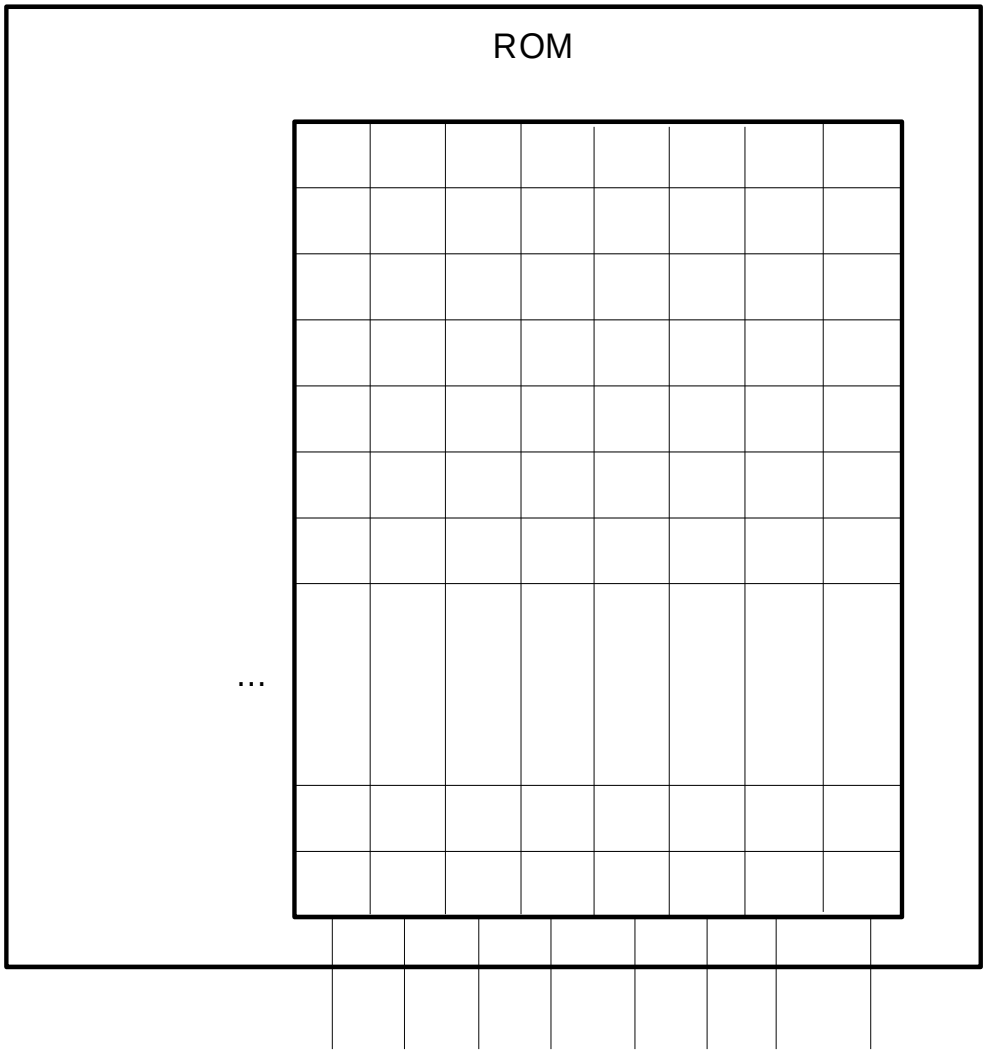




4x4 Multiplier Example

Determine the dimensions of the ROM that would be necessary to implement a 4x4-bit unsigned multiplier with inputs X[3:0] and Y[3:0] and outputs P[??:0]

(Question: How many bits are needed for P).



Example:

$X_3X_2X_1X_0$=0010

$Y_3Y_2Y_1Y_0$=0001

P = X * Y = 2 * 1 = 2

= 0010

Implementing Functions w/ ROM's

- To implement a function w/ n-variables and m outputs
- Use a ROM w/ 2^n rows and m columns
- Just place the output truth table values in the ROM