USC Viterbi School of Engineering

Spiral 2-4

Function synthesis with:
Muxes (Shannon's Theorem)
Memories

Learning Outcomes

- I can implement logic for any truth table by using Shannon's theorem to decompose the function to create two smaller functions and a 2-to-1 mux
  - I can recursively apply Shannon's theorem k times to decompose any size truth table to arrive at 2^k smaller functions and a 2^k-to-1 mux
- I can implement logic for any truth table by using a memory as a look-up table
  - I understand how to determine the necessary dimensions of the memory
  - I understand how to reinterpret input combinations as address inputs to determine the correct row to place the desired output

Function Synthesis Techniques

- Given a combination function (i.e. truth table or other description) what methods can we use to arrive at a circuit?
  - Karnaugh maps
  - Sum of minterms / Produce of maxterms
  - Neither of these scale well to larger number of inputs
- Now we will see a few others

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

Implementing functions with muxes

SHANNON'S THEOREM

Simplify This

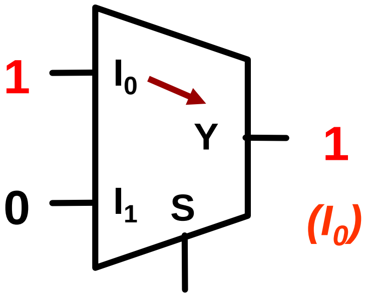
- Given $F(x,y,z) = x'yz + y'z'$,
  simplify $F(0,y,z) =$

  then simplify $F(1,y,z) =$

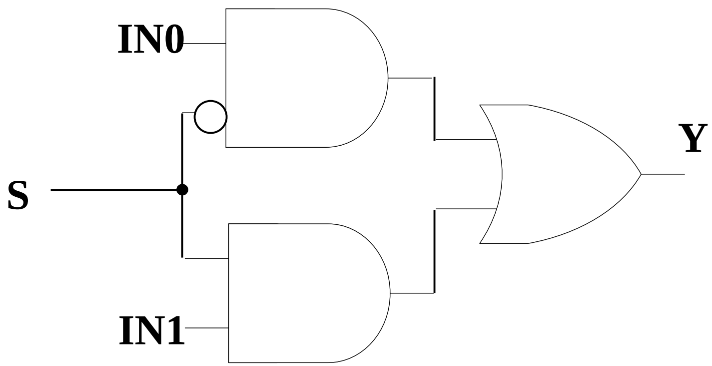
- Given $G(a,b,c,d) = bd' + ab'cd + ac'd'$
  $G(1,1,c,d) =$

2-to-1 Mux: Another View

Old Views:



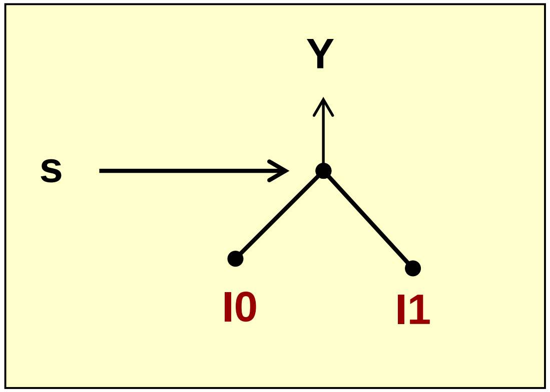
1-bit wide 2-to-1 mux




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

New View:

We can show the function of a 2-to-1 mux as a splitter where the variable 's' decides which input passes upwards



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

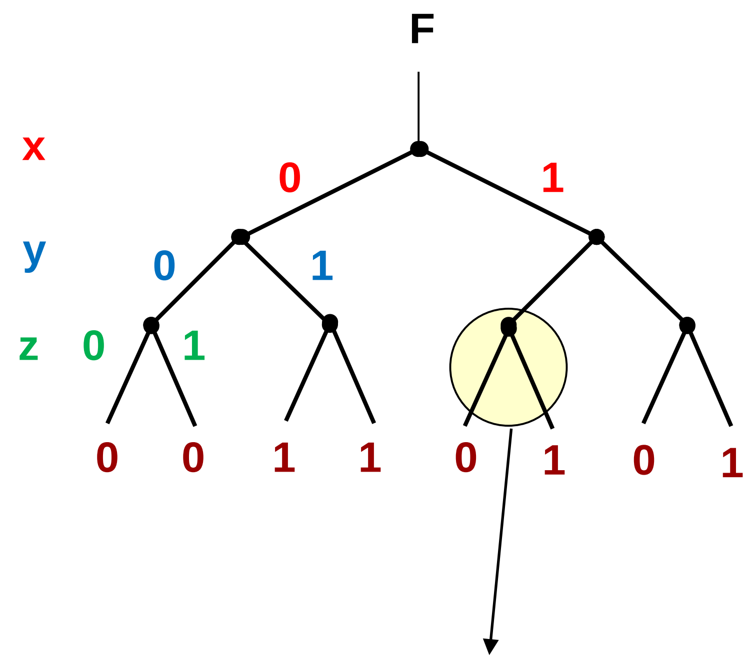
Primes between 0-7

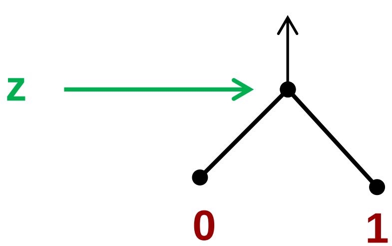
X	Y	Z	P
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Truth Table

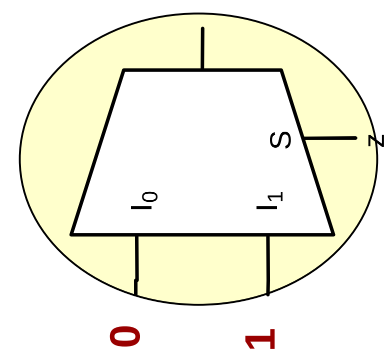
OFF-set

ON-set






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



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

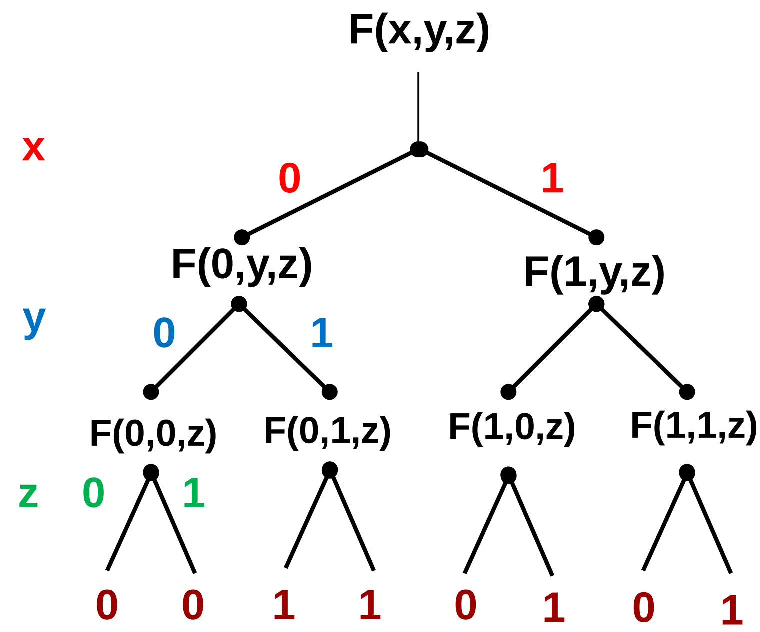
F(0,1,z)

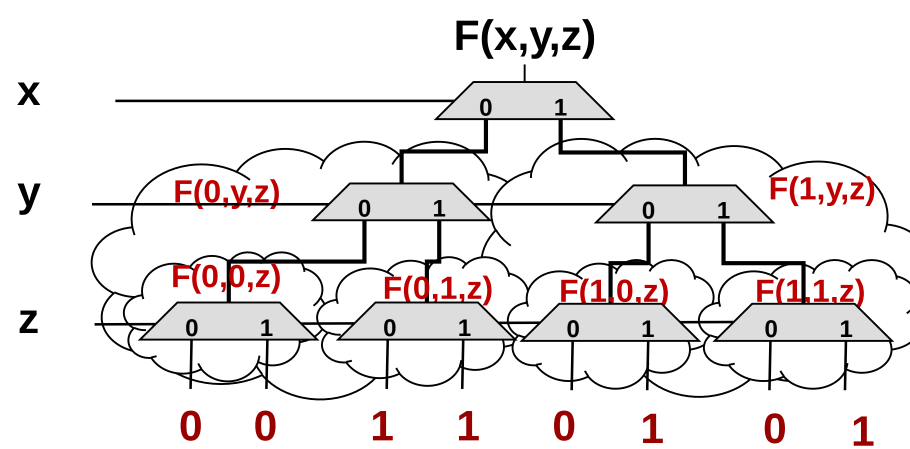
Z	F
0	0
1	1

F(1,0,z)

Z	F
0	0
1	1

F(1,1,z)

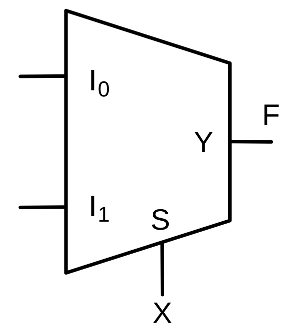




Splitting on X

- 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**



Put the k MSB's on the selects

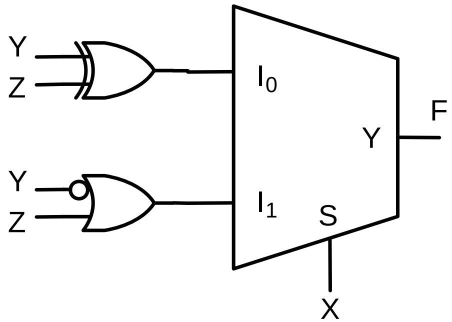
Splitting on X

- 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')

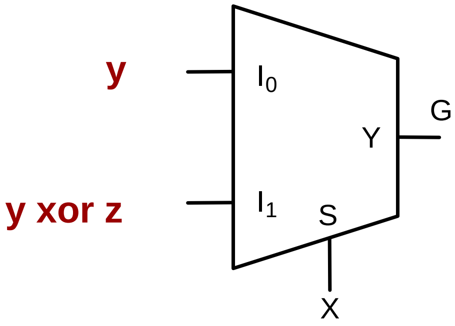
(y' + z)



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

- $F(X_1,X_2,\ldots,X_n) = X_1'\bullet F(0,X_2,\ldots,X_n) + X_1\bullet F(1,X_2,\ldots,X_n)$
- 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
- We can recursively apply Shannon's theorem to pull out more variables:
  - $F(X_1,X_2,\ldots,X_n) =$
    $X_1'X_2'\bullet F(0,0,\ldots,X_n) + X_1'X_2\bullet F(0,1,\ldots,X_n) +$
    $X_1X_2'\bullet F(1,0,\ldots,X_n) + X_1X_2\bullet F(1,1,\ldots,X_n) +$

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*

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

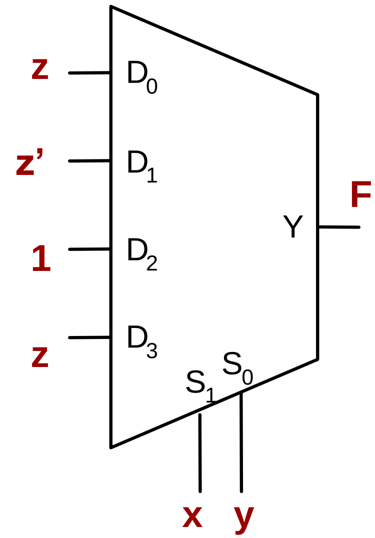
X	Y	Z	F	
0	0	0	0	z
		1	1	
0	1	0	1	z'
		1	0	
1	0	0	1	1
		1	1	
1	1	0	0	z
		1	1	

$F_0(z)$

$F_1(z)$

$F_2(z)$

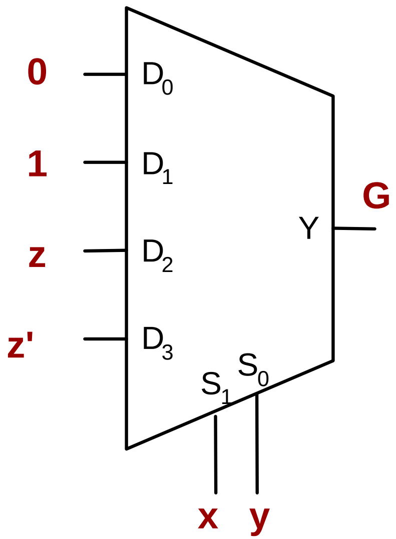
$F_3(z)$



Put the n-1 MSB's on the selects

More Practice

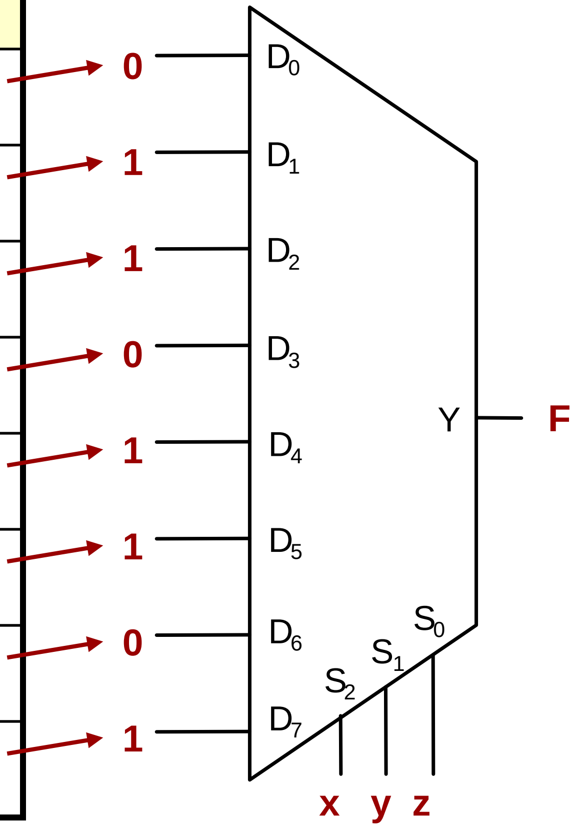
X	Y	Z	G
0	0	0	0
		1	0
0	1	0	1
		1	1
1	0	0	0
		1	1
1	1	0	1
		1	0



As Far as We like

- We can take this tactic all the way down and use ONLY a mux to implement any function
- 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

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

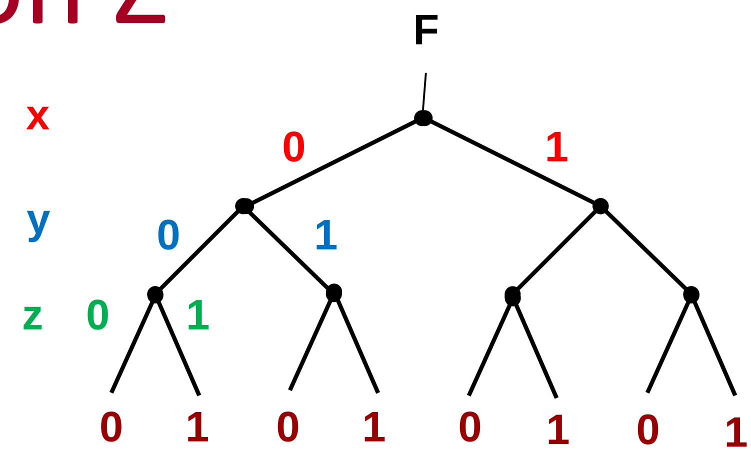


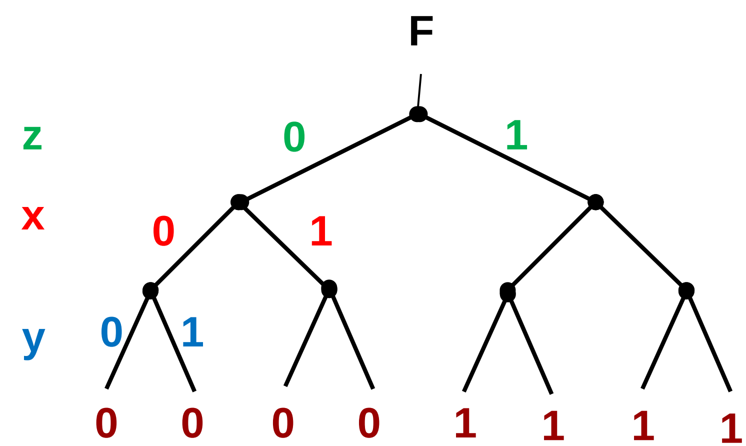
Splitting on Z

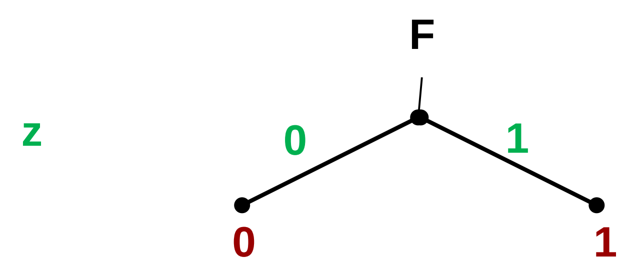
- We can always rearrange our variables if it helps make the function simpler to implement

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





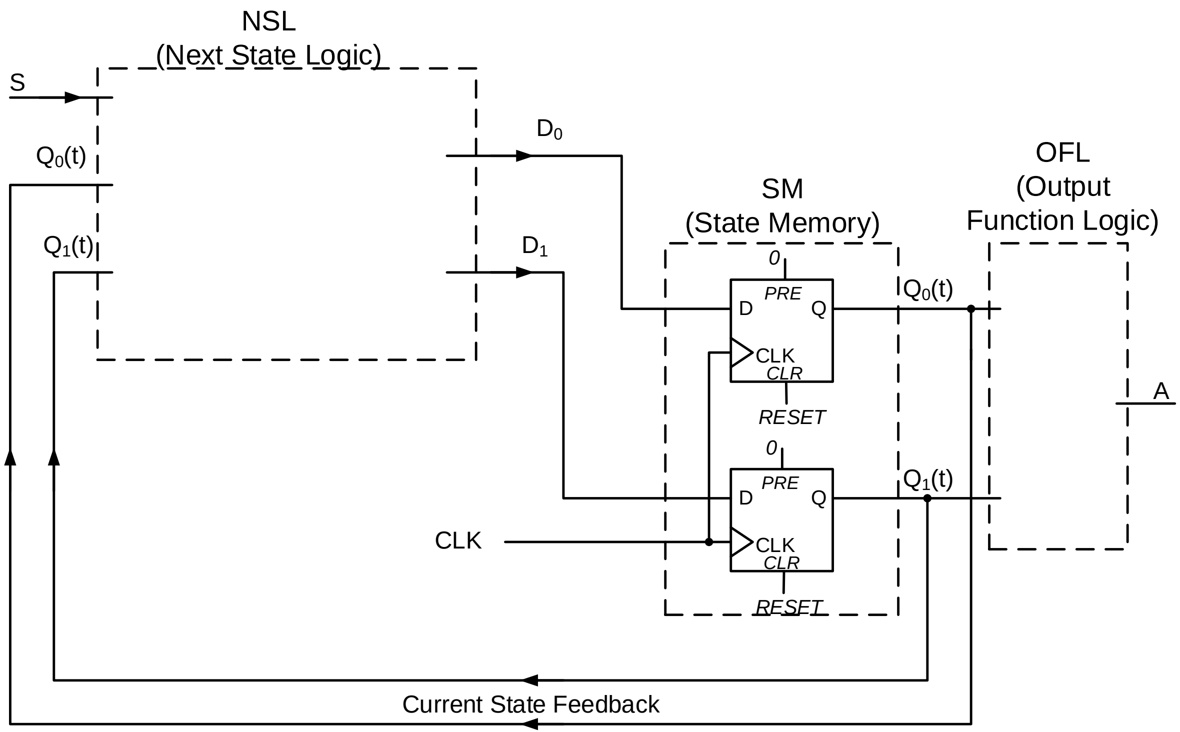


Implementing Logic Functions

- We can use muxes to implement any arbitrary logic function
  - Choose one variable to split the large function into two smaller functions: f(0,x2,x3,…) and f(1,x2,x3,…)
  - A 2-to-1 mux will produce the output bit and the chosen "split" variable will be the select
  - Implement f(0,x2,x3,…) using any known method and connect it to input 0 of the 2-to-1 mux
  - Implement f(1,x2,x3,…) using any known method and connect it to input 1 of the 2-to-1 mux

Implementing an Initial State

- Since the NSL is just a combinational function of the current state and inputs, we can use Shannon's theorem (i.e. muxes)to find an implementation rather than K-Maps



USC Viterbi School of Engineering

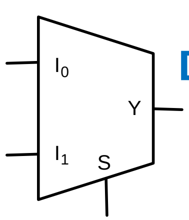
Example 1

- Implement D1 & D0 using 2-to-1 muxes with S as the select

Current State			Next State						Output
			S = 0			**S = 1**			
State	Q_1	Q_0	State	Q1*= D1	Q0*= D0	State	Q1* =D1	Q0* =D0	A
G01	0	0	G00	1	1	G10	0	1	1
G10	0	1	G01	0	0	G11	1	0	1
G00	1	1	G00	1	1	G10	0	1	0
G11	1	0	G01	0	0	G11	1	0	0

Q1 xnor Q0

Q1 xor Q0

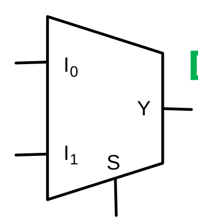


D1

S

Q1 xnor Q0

Q1 xnor Q0



D0

S

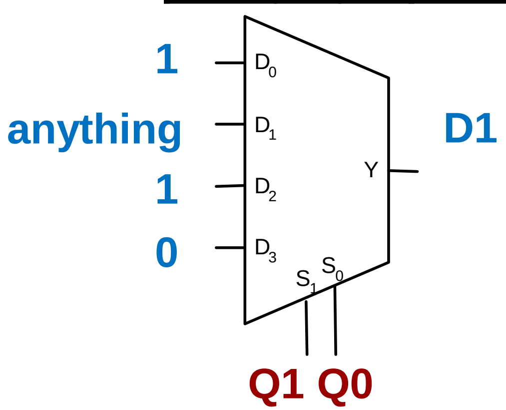
D0 = Q1 xnor Q0

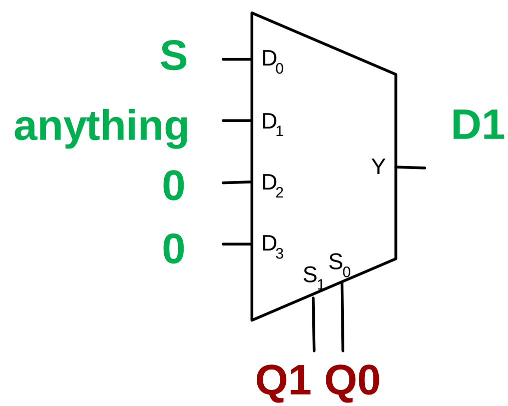
(Since both inputs are xnor, we don't need the mux)

Example 2

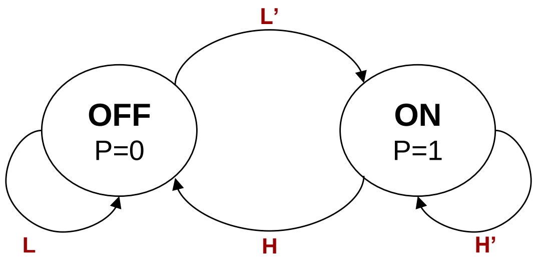
- Implement D1 and D0 using (2) 4-to-1 muxes with Q1,Q0 as the selects

Current State			Next State						Output		
			S = 0			S = 1					
State	Q_1	Q_0	State	Q_1^*	Q_0^*	State	Q_1^*	Q_0^*	SSG	MTG	MSG
SS	0	0	MS	1	0	MT	1	1	1	0	0
N/A	0	1	X	d	d	X	d	d	d	d	d
MT	1	1	MS	1	0	MS	1	0	0	1	0
MS	1	0	SS	0	0	SS	0	0	0	0	1





Example 3



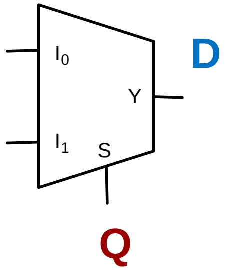
- Implement D using a mux

Current State		Next State							
		H L = 0 0		H L = 0 1		H L = 1 1		**H L = 1 0**	
Symbol	Q	Sym.	Q*	Sym.	Q*	Sym.	Q*	Sym.	**Q***
OFF	0	ON	1	OFF	0	OFF	0	X	**d**
ON	1	ON	1	ON	1	OFF	0	X	d

Note: The State Value, Q forms the Pump output (i.e. 1 when we want the pump to be on and 0 othewise)

Due to don't care:
Option 1: H'L'
Option 2: L'

Option 1: H'
Option 2: H'+L'

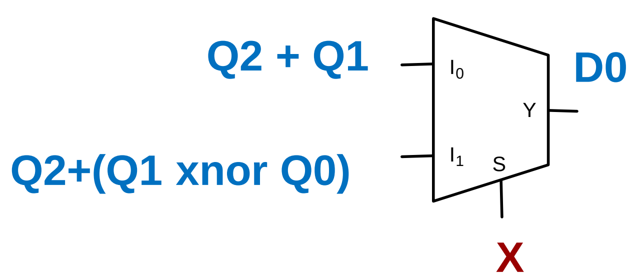


Example 4

- Implement D0 using a mux.

Do a K-Map for each column separately since the mux splits them

Current State				Next State								Output
				X = 0				X = 1				
State	Q2	Q1	Q0	State*	D2	D1	D0	State*	D2	D1	D0	Z
Sinit	0	0	0	Sinit	0	0	0	S1	0	1	1	0
S10	0	0	1	Sinit	0	0	0	S101	0	1	0	0
S1	0	1	1	S10	0	0	1	S1	0	1	1	0
S101	0	1	0	S10	0	0	1	S1011	1	1	0	0
S1011	1	1	0	S10	0	0	1	S1	0	1	1	1



Summary

- Shannon's theorem allows us to decompose an ARBITRARILY large function into many smaller functions
- This allows a method that can scale for a function with many variables
- It is at the heart of many computer algorithms that will find logic implementation given high-level descriptions of a function

Using a LookUp-Table to implement a function

MEMORIES

Memories as Look-Up Tables

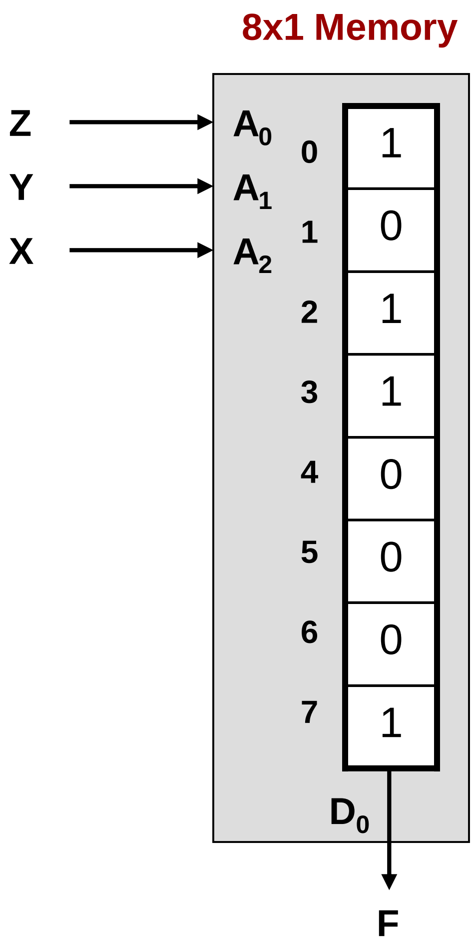
- One major application of memories in digital design is to use them as LUT's (Look-Up Tables) to implement logic functions
- Given a logic function use a memory to hold all the possible answers and feed the inputs of the function to the address inputs to look-up the answer

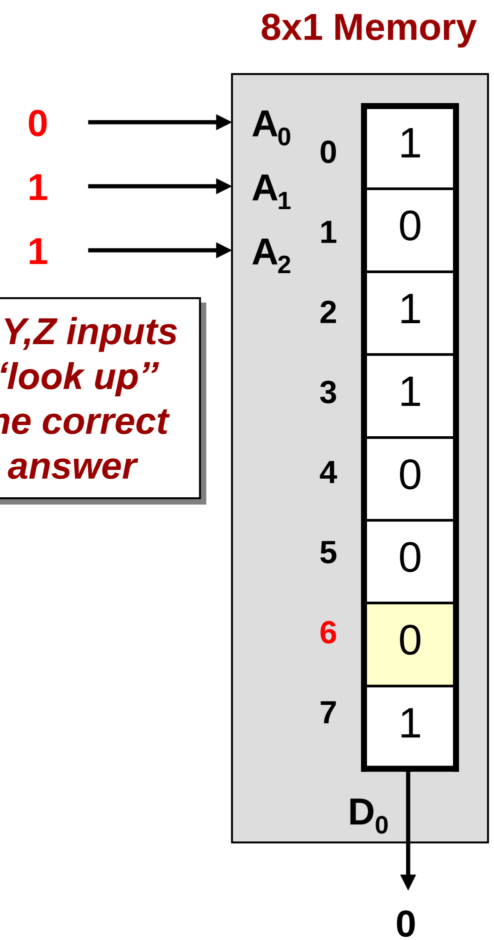
USC Viterbi School of Engineering

Implementing Functions w/ Memories

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



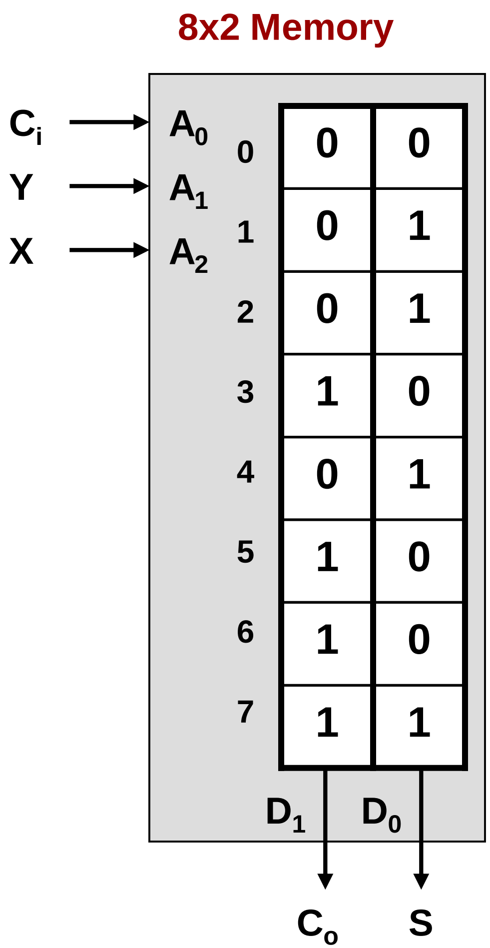


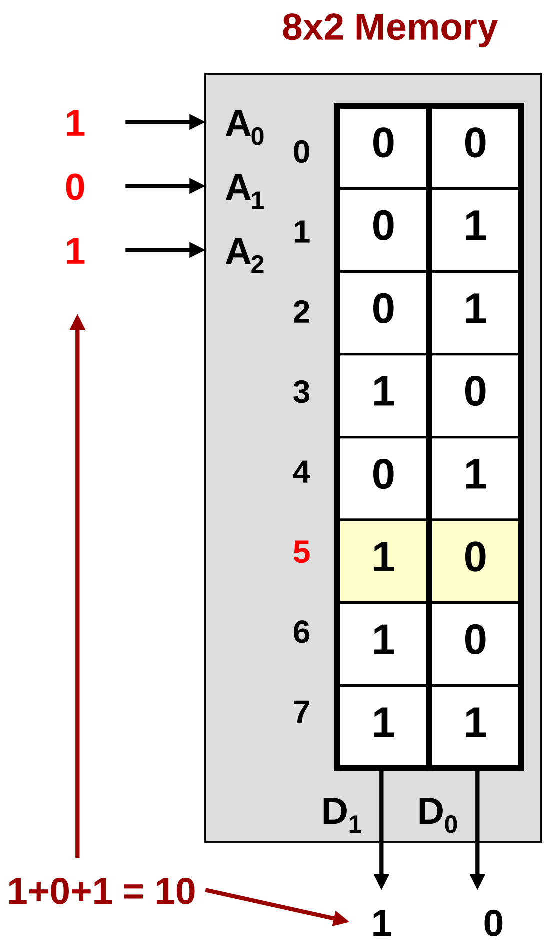
X,Y,Z inputs "look up" the correct answer

Implementing Functions w/ Memories

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

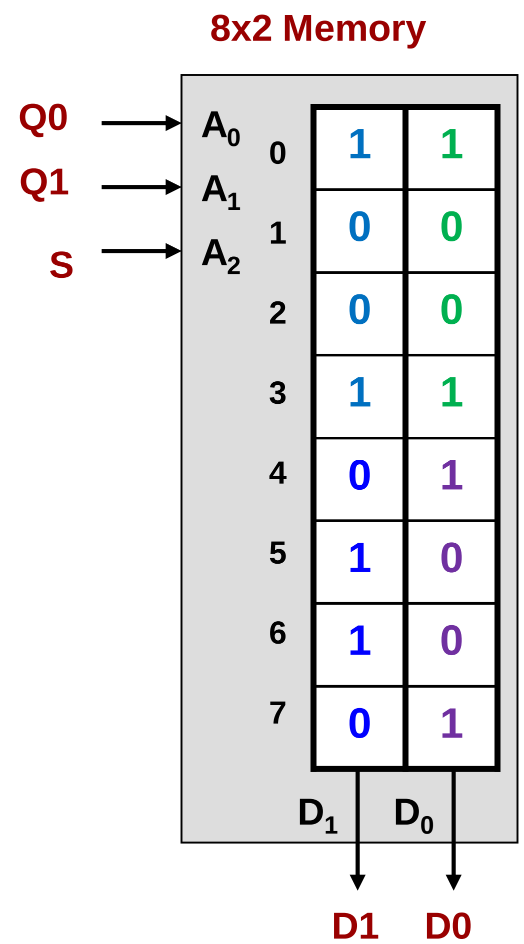




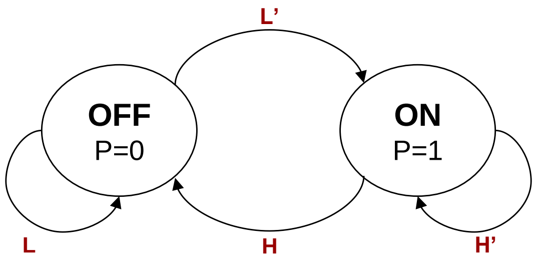
Example 1

- Implement D1 & D0 using a memory

Current State			Next State					
			S = 0			S = 1		
State	Q_1	Q_0	State	Q1*	Q0*	State	Q1*	Q0*
G01	0	0	G00	1	1	G10	0	1
G10	0	1	G01	0	0	G11	1	0
G00	1	1	G00	1	1	G10	0	1
G11	1	0	G01	0	0	G11	1	0

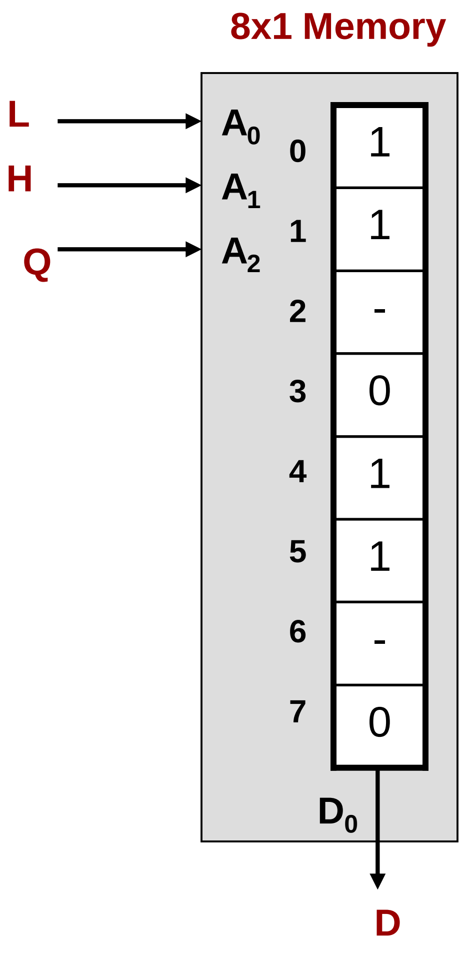


Example 2



- Implement D using a memory

Current State		**Next State**							
		H L = 0 0		H L = 0 1		H L = 1 1		**H L = 1 0**	
Symbol	Q	Sym.	Q*	Sym.	Q*	Sym.	Q*	Sym.	**Q***
OFF	0	ON	1	OFF	0	OFF	0	X	**d**
ON	1	ON	1	ON	1	OFF	0	X	d

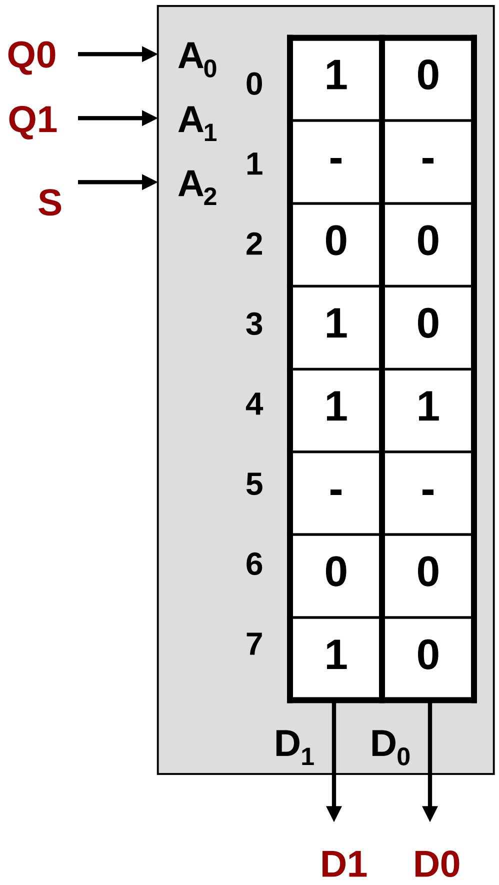


Example 3

- Implement D1 and D0 using a memory

Current State			Next State						Output		
			S = 0			S = 1					
State	Q_1	Q_0	State	Q_1^*	Q_0^*	State	Q_1^*	Q_0^*	SSG	MTG	MSG
SS	0	0	MS	1	0	MT	1	1	1	0	0
N/A	0	1	X	d	d	X	d	d	d	d	d
MT	1	1	MS	1	0	MS	1	0	0	1	0
MS	1	0	SS	0	0	SS	0	0	0	0	1

8x2 Memory



4x4 Multiplier Example

Determine the dimensions of the memory 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).

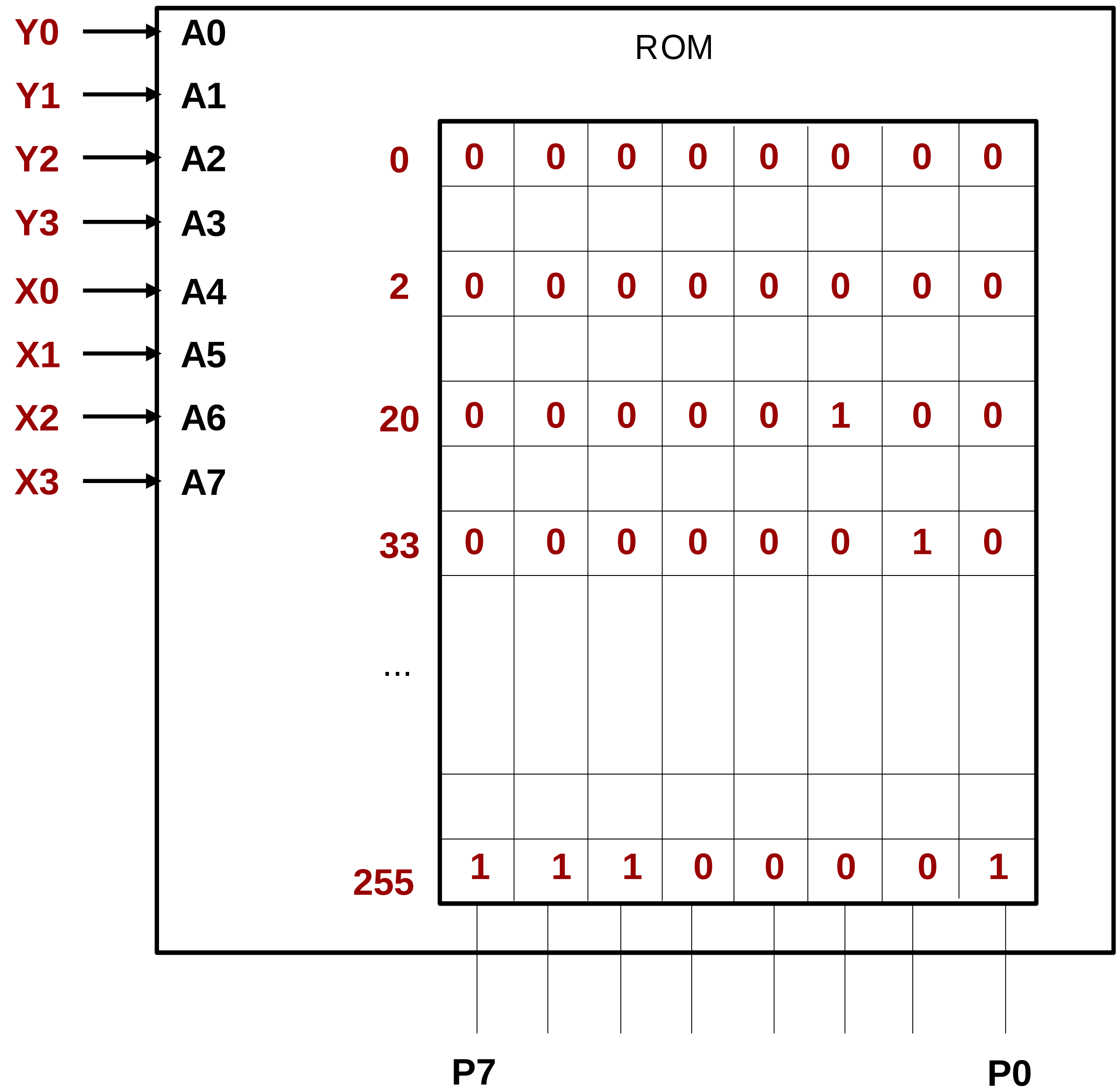
Example:

$X_3X_2X_1X_0$=0010

$Y_3Y_2Y_1Y_0$=0001

P = X * Y = 2 * 1 = 2

= 00010



Implementing Functions w/ Memories

- To implement a function w/ n-variables and m outputs
- Just place the output truth table values in the memory
- Memory will have dimensions: 2^n rows and m columns
  - Still does not scale terribly well (i.e. n-inputs requires memory w/ 2^n outputs)
  - But it is easy and since we can change the contents of memories it allows us to create "reconfigurable" logic
  - This idea is at the heart of FPGAs