Combinational Logic
Chapter 4

EECE 256

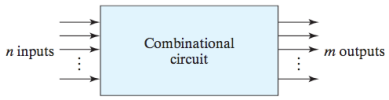
Dr. Sidney Fels
Steven Oldridge

Topics

- Combinational circuits
- Combinational analysis
- Design procedure
  - simple combined to make complex
  - adders, subtractors, converters
  - decoders, multiplexers
    - comb. design with decoders and muxes

Combination Circuit

- Output depends only on present value of input



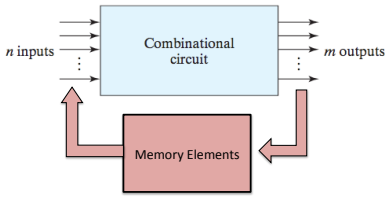
NOTE: No Memory or feedback paths

Combination Circuit

- What happens if we add memory?

Combination Circuit

- What happens if we add memory?



Called a: *Sequential Circuit*
- output function of input and memory
- changes over time

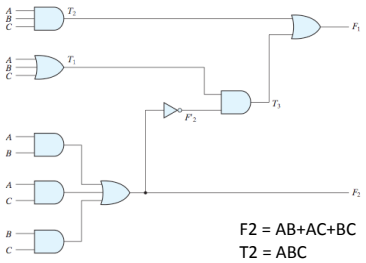
Combination Circuit

- We'll focus on combinational design first
  - useful for designing how memory will change when making sequential circuits

Combinational Analysis

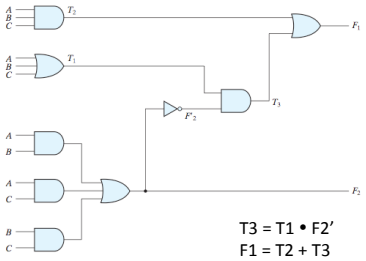
- Sometimes need to:
  - confirm circuit does what it is supposed to
  - reverse engineer circuit
- Steps (start at input and work to outputs):
  1. label all outputs that are fn' of inputs and derive Boolean expression
  2. label all outputs that fn' of inputs and labels done in step 1 and derive Boolean expressions
  3. repeat until all outputs have Boolean fn'
  4. use substitution to get Boolean fn' based only on inputs
     - fill in truth table

Combinational Analysis



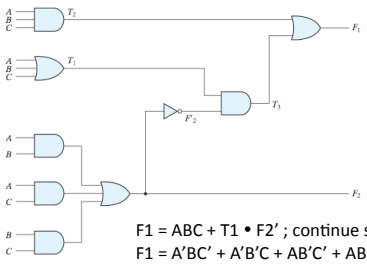
F2 = AB+AC+BC
T2 = ABC
T1 = A+B+C

Combinational Analysis



T3 = T1 • F2'
F1 = T2 + T3

Combinational Analysis



F1 = ABC + T1 • F2′ ; continue substituting
F1 = A′BC′ + A′B′C + AB′C′ + ABC

Combinational Design

1. Determine the number of inputs and outputs
2. Assign symbols
3. Derive the truth table
4. Obtain simplified functions for each output
5. Draw the logic diagram

Adders

- most fundamental unit in computer
- add two numbers
  - let's start with binary…

½ Adder Design

- Step 1 – # of inputs and outputs
  - let's start with ½ adder first
    - i.e. let's not worry about carry in just yet

	0	0	1	1
+	0	1	0	1
sum	0	1	1	0
carry out	0	0	0	1

½ Adder Design

- Step 2 – Assign symbol names

	0	0	1	1	**A**
+	0	1	0	1	**B**
sum	0	1	1	0	**S**
carry out	0	0	0	1	**C**

½ Adder Design

- Step 3 – Derive Truth Table

	0	0	1	1	**A**
+	0	1	0	1	**B**
sum	0	1	1	0	**S**
carry out	0	0	0	1	**C**

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

½ Adder Design

- Step 4 – Derive simplified form
  - use k-maps, Boolean Algrebra, etc.

	0	0	1	1	**A**
+	0	1	0	1	**B**
sum	0	1	1	0	**S**
carry out	0	0	0	1	**C**

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S = A \oplus B$
$C = AB$

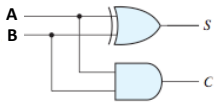
½ Adder Design

- Step 5 – Draw circuit diagram

	0	0	1	1	**A**
+	0	1	0	1	**B**
sum	0	1	1	0	**S**
carry out	0	0	0	1	**C**

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S = A \oplus B$
$C = AB$



Full Adder

- We need to worry about possible carry-in though
- Same procedure
  - input: x, y, Cin
  - output: S and Cout
  - find TT

x	y	Cin	S	Cout
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Full Adder

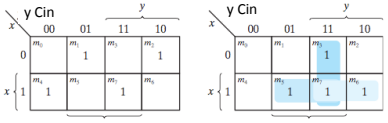
- We need to worry about possible carry-in though
- Same procedure
  - input: x, y, Cin
  - output: S and Cout
  - find TT

x	y	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full Adder

- Derive Boolean expressions

x	y	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



S **C_{out}**

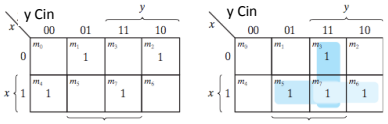
$S = x'y'C_{in} + x'yC_{in}' + xy'C_{in}' + xyC_{in}$

$C_{out} = xy + xC_{in} + yC_{in}$

Full Adder

- Derive Boolean expressions

x	y	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

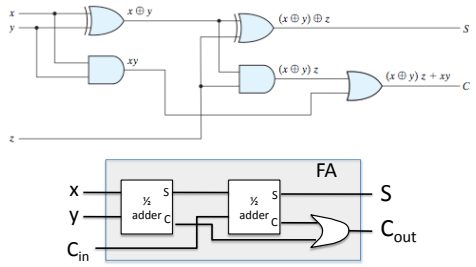


S **C_{out}**

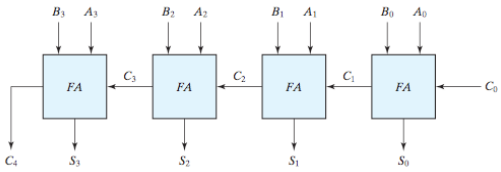
$S = x'y'C_{in} + x'yC_{in}' + xy'C_{in}' + xyC_{in} = (x \oplus y) \oplus Cin$

$C_{out} = xy + xC_{in} + yC_{in} = (x \oplus y)\, C_{in} + xy$

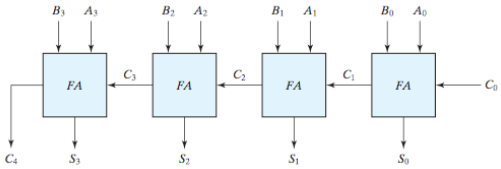
Full Adder



Cascade them for 4-bit Adder



Cascade them for a parallel 4-bit Adder



ripple carry adder

How big is the truth table?

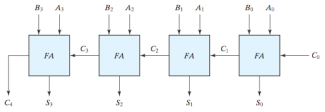
What's the problem with this circuit?

Can we fix this?

- we need to compute carry bits in parallel rather than cascade
- Then, use some ½ adders with extra circuits to fix the sum depending upon the computed carries

n-bit Carry look-ahead Adder

- Recall that for the design of the parallel adder to work, the signal must propagate through the gates before the correct output sum is available.
- *Total propagation time =*
  propagation delay of a typical gate x *the number of gates*
- Let's look at S3.
  - Inputs A3 and B3 are available immediately.
  - However, C3 is available only after C2 is available.
  - C2 has to wait for C1, etc.



n-bit Carry look-ahead Adder

- The number of gate levels for the carry to propagate is found from the FA circuit

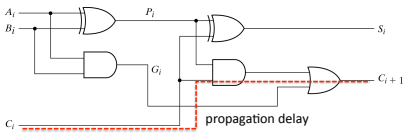


Fig. 4-10 Full Adder with P and G Shown

n-bit Carry look-ahead Adder

- let's look at each stage to see how we know what the carry is

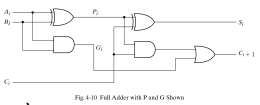


Fig. 4-10 Full Adder with P and G Shown

- $P_i = A_i \oplus B_i$ (carry prop)
- $G_i = A_i B_i$ (carry generate)

- $S_i = P_i \oplus C_i$
- $C_{i+1} = G_i + P_i C_i$

n-bit Carry look-ahead Adder

- let's look at each stage to see how we know what the carry is

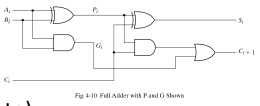


Fig. 4-10 Full Adder with P and G Shown

- $P_i = A_i \oplus B_i$ (carry prop)
- $G_i = A_i B_i$ (carry generate)

- $S_i = P_i \oplus C_i$
- $C_{i+1} = G_i + P_i C_i$

Now, calculate each stage's Carry in terms of Cin (i.e. C_0) and Ps or Gs.

n-bit Carry look-ahead Adder

- $C_0 = C_{in}$
- $C_1 = G_0 + P_0 C_0$
- $C_2 =$

- $C_3 =$

n-bit Carry look-ahead Adder

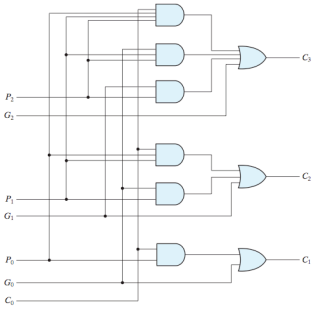
- $C_0 = C_{in}$
- $C_1 = G_0 + P_0C_0$
- $C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0)$
  $= G_1+P_1G_0+P_1P_0C_0$
- $C_3 =$

n-bit Carry look-ahead Adder

- $C_0 = C_{in}$
- $C_1 = G_0 + P_0C_0$
- $C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0)$
  $= G_1+P_1G_0+P_1P_0C_0$
- $C_3 = G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0)$
  $= G_2 + P_2G_1 + P_1P_2G_0 + P_2P_1P_0C_0$

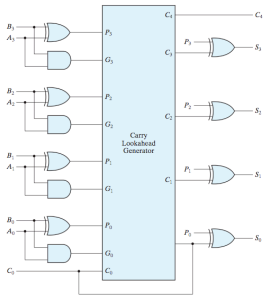
So, all carries are now computed in P2, Gs and C_0

Carry look-ahead circuit



4 bit Carry look-ahead Adder

- $P_i = A_i \oplus B_i$
- $G_i = A_i B_i$
- $S_i = P_i \oplus C_i$



What is propagation delay?

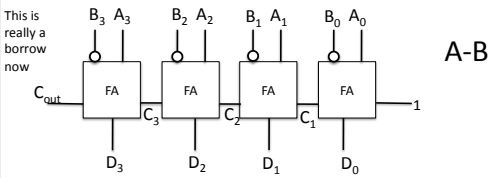
Binary Subtractor

- How to make a binary subtractor?

Binary Subtractor

- How to make a binary subtractor?
  - remember: 2's complement converts subtraction into addition
  - so, when we want to subtract,
    - make the input a 2's complement number
    - and add
  - check for overflow
  - 2's complement?
    - invert the bits + 1

4-bit Binary subtractor



What about overflow?

- if unsigned – if B > A; look at C_{out}
- if signed – if A +ve and B –ve or A –ve and B +ve - look at C_{out} and sign bit (C_3) – should be same

overflow

case 1: A,B unsigned: B>A and result < -7

	0	0	1	1	3
-	1	0	1	1	11 unsigned
					-8

	0	0	1	1	3
+	0	1	0	1	2's comp of 11 unsigned
1	1	0	0	0	-8

overflow

case 2 - signed: a is –ve, b is +ve; result < -7, i.e. -6 – (+6)

	1	0	1	0	-6
-	0	1	1	0	- (+6) signed

	1	0	1	0	-6
+	1	0	1	0	+ (2'S complement of +6)
					-12 – but 4 bits only -7 to +7

overflow

case 3 - signed: a is +ve, b is -ve; result > +7, i.e. 6 – (-6)

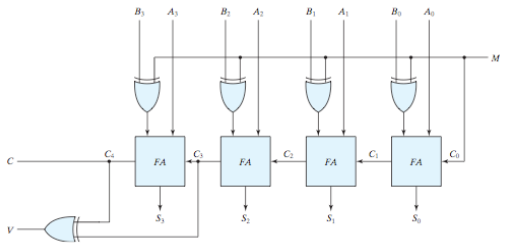
	0	1	1	0	+6
-	1	0	1	0	- (-6) signed

	0	1	1	0	+6
+	0	1	1	0	+ (2'S complement of -6)
					+12 – but 4 bits only -7 to +7

4-bit Binary adder/subtractor

- We can put this together to make a more functional element
  - M = mode for add (0) or subtract (1)

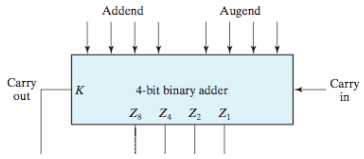
4-bit Binary adder/subtractor



Design of a BCD Adder

- Add two decimal numbers
  - (0-9)+(0-9)+(1)= 0-19 - don't forget carry
- How to begin?
  - truth table
  - can we use existing circuit
    - binary adder?

Design of a BCD Adder



If we just put decimal numbers in, it almost works…

BCD adder TT

Table 4.5
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

BCD adder TT

Table 4.5
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

same as binary repn'

need decimal carry

BCD adder TT

Table 4.5
Derivation of BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

same as binary repn'

need decimal carry

Z_8Z_2

Z_8Z_4

K

$C = K + Z_8Z_4 + Z_8Z_2$

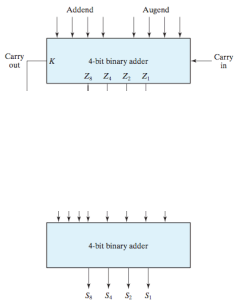
BCD adder TT

0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

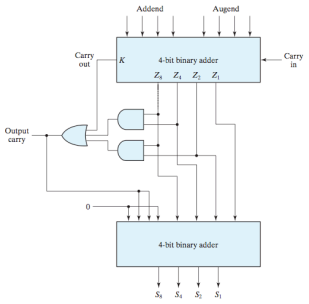
if C=1; need to add 6 to the Binary sum
- so we need another binary adder

So, we can now draw circuit as we have C and final Sum

BCD adder design



BCD adder design



Decoders

- A binary code of *n* bits can represent 2^n distinct combinations (or unique "cases").
- **Decoder**: a combinational circuit that converts *n* binary lines into 2^n unique output lines
- Example: a 3-to-8 line decoder
  - 3 inputs are decoded to 8 outputs – representing the 8 minterms

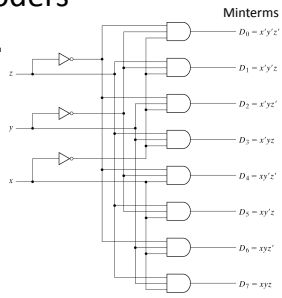
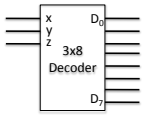


Fig. 4-18 3-to-8-Line Decoder

Decoders

- A binary code of *n* bits can represent 2^n distinct combinations (or unique "cases").
- **Decoder**: a combinational circuit that converts *n* binary lines into 2^n unique output lines
- Example: a 3-to-8 line decoder
  - 3 inputs are decoded to 8 outputs – representing the 8 minterms



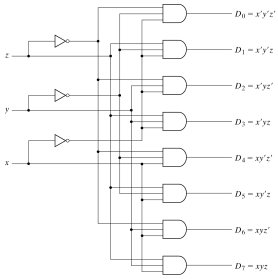


Fig. 4-18 3-to-8-Line Decoder

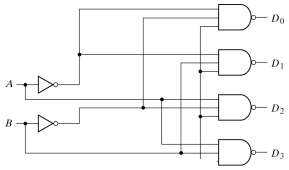
Decoder design

Inputs			Outputs							
x	y	z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0								
0	0	1								
0	1	0								
0	1	1								
1	0	0								
1	0	1								
1	1	0								
1	1	1								

Decoder truth table

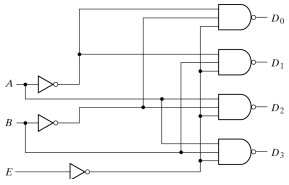
Inputs			Outputs							
x	y	z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Decoder with NAND



A	B	D_0	D_1	D_2	D_3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

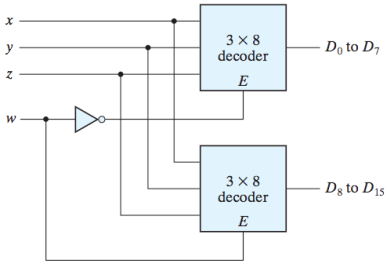
NAND Decoder with enable



E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Now we can combine them to get bigger ones.

Combining Decoders

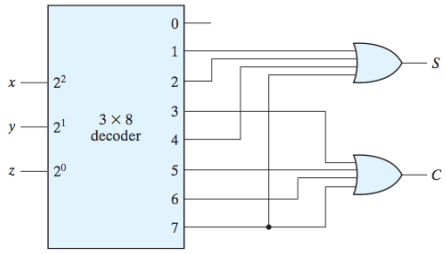


4x16 Decoder – check specs whether inverted or not

Boolean Functions with Decoders

- since we have all minterms it is easy to combine them for our Boolean functions
- For example:
  - S = Σ(1, 2, 4, 7)
  - C = Σ(3, 5, 6, 7)

Boolean Function with Decoders



Encoders

- Inverse operation of a decoder
  - It has 2n inputs and generates n codewords
- Example: Design a 8 x 3 encoder

8x3 Encoders

Table 4.7
Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

X = D4 + D5 + D6 + D7
Y = D2 + D3 + D6 + D7
Z = D1 + D3 + D5 + D7

8x3 Encoders

Table 4.7
Truth Table of an Octal-to-Binary Encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

X = D4 + D5 + D6 + D7
Y = D2 + D3 + D6 + D7
Z = D1 + D3 + D5 + D7

What's the problem here?

Priority Encoder

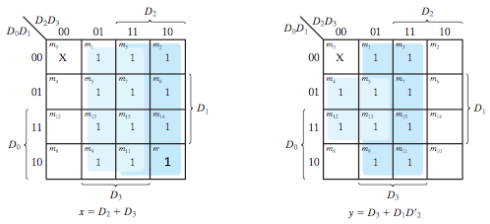
- For the other inputs use priority to determine output
  - i.e. D3 takes priority over D2; D2 over D1 etc.

Priority Encoder

Table 4.8
Truth Table of a Priority Encoder

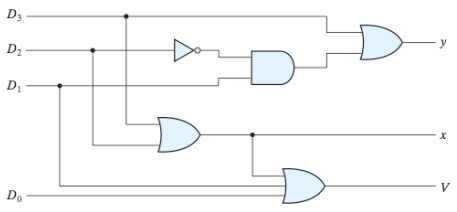
Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Priority Encoder



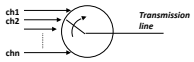
Notice, we can use the don't cares to design this.

Priority Encoder



Multiplexers

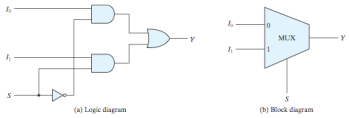
- A multiplexer selects one of many inputs and directs it to the output.



- The selection may be controlled by "select lines"
- Normally 2n input lines: n select lines

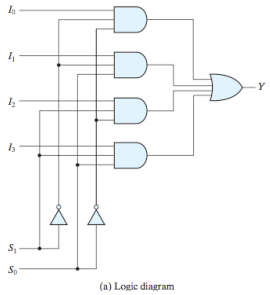
Multiplexers

- Example: 2 x 1 multiplexer



- How to design?
  - let's design 4x1 MUX
  - code redirects input
    - use AND gate with minterm
    - like decoder with AND gate

4x1 MUX



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

Using MUXes for Boolean functions

- Use a multiplexer to implement the following function:
  - F = x'y'z + x'yz' + xy'z + xyz
- Idea:
  - notice that MUX is a decoder + OR gate
  - use the selector to direct correct value to output

Using MUXes for Boolean functions

- Example
  - F(x,y,z) = Σ(1,2,6,7)

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(a) Truth table

Using MUXes for Boolean functions

- Example
  - F(x,y,z) = Σ(1,2,6,7)

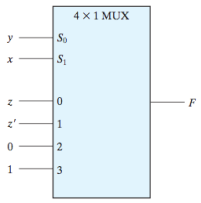
x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table

Using MUXes for Boolean functions

x	y	z	F	
0	0	0	0	$F = z$
0	0	1	1	
0	1	0	1	$F = z'$
0	1	1	0	
1	0	0	0	$F = 0$
1	0	1	0	
1	1	0	1	$F = 1$
1	1	1	1	

(a) Truth table



(b) Multiplexer implementation

Using MUX for Boolean Fn'

- Design a Full-adder
  - S(x,y,z) = Σ(1,2,4,7); C(x,y,z) = Σ(1,2,4,7)

x	y	z	S	C
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Using MUX for Boolean Fn'

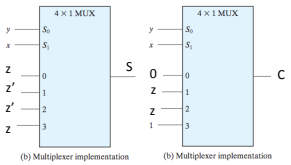
- Design a Full-adder
  - S(x,y,z) = Σ(1,2,4,7); C(x,y,z) = Σ(3,5,6,7)

x	y	z	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Using MUX for Boolean Fn'

- Design a Full-adder
  - S(x,y,z) = Σ(1,2,4,7); C(x,y,z) = Σ(3,5,6,7)

x	y	z	S	C	S_m	C_m
0	0	0	0	0	z	0
0	0	1	1	0	z	0
0	1	0	1	0	z'	z
0	1	1	0	1	z'	z
1	0	0	1	0	z'	z
1	0	1	0	1	z'	z
1	1	0	0	1	z	1
1	1	1	1	1	z	1



Summary

- Combinational circuits
- Combinational analysis
- Design procedure
  - simple combined to make complex
  - adders, subtractors, converters
  - decoders, multiplexers
    - comb. design with decoders and muxes