

## Lecture Summary – Module 2

### Combinational Logic Circuits

**Learning Outcome:** *an ability to analyze and design combinational logic circuits*

#### Learning Objectives:

- 2-1. identify minterms (product terms) and maxterms (sum terms)
- 2-2. list the standard forms for expressing a logic function and give an example of each: sum-of-products (SoP), product-of-sums (PoS), ON set, OFF set
- 2-3. analyze the functional behavior of a logic circuit by constructing a truth table that lists the relationship between input variable combinations and the output variable
- 2-4. transform a logic circuit from one set of symbols to another through graphical application of DeMorgan's Law
- 2-5. realize a combinational function directly using basic gates (NOT, AND, OR, NAND, NOR)
- 2-6. draw a Karnaugh Map ("K-map") for a 2-, 3-, 4-, or 5-variable logic function
- 2-7. list the assumptions underlying function minimization
- 2-8. identify the prime implicants, essential prime implicants, and non-essential prime implicants of a function depicted on a K-map
- 2-9. use a K-map to minimize a logic function (including those that are incompletely specified) and express it in either minimal SoP or PoS form
- 2-10. use a K-map to convert a function from one standard form to another
- 2-11. calculate and compare the cost (based on the total number of gate inputs plus the number of gate outputs) of minimal SoP and PoS realizations of a given function
- 2-12. realize a function depicted on a K-map as a two-level NAND circuit, two-level NOR circuit, or as an open-drain NAND/wired-AND circuit
- 2-13. define and identify static-0, static-1, and dynamic hazards
- 2-14. describe how a static hazard can be eliminated by including consensus terms
- 2-15. describe a circuit that takes advantage of the existence of hazards and analyze its behavior
- 2-16. draw a timing chart that depicts the input-output relationship of a combinational circuit
- 2-17. identify properties of XOR/XNOR functions
- 2-18. simplify an otherwise non-minimizable function by expressing it in terms of XOR/XNOR operators
- 2-19. describe the genesis of programmable logic devices
- 2-20. list the differences between complex programmable logic devices (CPLDs) and field programmable gate arrays (FPGAs) and describe the basic organization of each
- 2-21. list the basic features and capabilities of a hardware description language (HDL)
- 2-22. list the syntactic elements of a Verilog module
- 2-23. identify operators and keywords used to create Verilog modules
- 2-24. write equations using Verilog dataflow syntax
- 2-25. define functional behavior by creating truth tables with the `casez` construct in Verilog
- 2-26. define the function of a decoder and describe how it can be use as a combinational logic building block
- 2-27. illustrate how a decoder can be used to realize an arbitrary Boolean function
- 2-28. define the function of an encoder and describe how it can be use as a combinational logic building block
- 2-29. discuss why the inputs of an encoder typically need to be prioritized
- 2-30. define the function of a multiplexer and describe how it can be use as a combinational logic building block
- 2-31. illustrate how a multiplexer can be used to realize an arbitrary Boolean function

## Lecture Summary – Module 2-A

### Combinational Circuit Analysis and Synthesis

**Reference:** *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 196-210, 5<sup>th</sup> Ed. 100-117

- overview
  - we *analyze* a combinational logic circuit by obtaining a *formal description* of its logic function
  - a *combinational logic circuit* is one whose output depend only on its *current combination of input values* (or “input combination”)
  - a *logic function* is the *assignment* of “0” or “1” to each possible combination of its input variables
- examples of formal descriptions (“standard forms”)
  - a *literal* is a variable or the complement of a variable
  - a *product term* is a single literal or a logical product of two or more literals
  - a *sum-of-products expression* is a logical sum of product terms
  - a *sum term* is a single literal or a logical sum of two or more literals
  - a *product-of-sums expression* is a logical product of sum terms
  - a *normal term* is a product or sum term in which no variable appears more than once
  - an *n-variable minterm* is a normal product term with n literals
  - an *n-variable maxterm* is a normal sum term with n literals
  - the *canonical sum* of a logic function is a sum of minterms corresponding to input combinations for which the function produces a “1” output
  - the *canonical product* of a logic function is a product of maxterms corresponding to input combinations for which the function produces a “0” output
- minterm and maxterm identification

0 → complemented  
 1 → true

0 → true  
 1 → complemented

Row	X	Y	Z	F	Minterm	Maxterm
0	0	0	0	F(0,0,0)	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	F(0,0,1)	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	F(0,1,0)	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	F(0,1,1)	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	F(1,0,0)	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	F(1,0,1)	$X \cdot Y' \cdot Z$	$X' + Y + Z'$
6	1	1	0	F(1,1,0)	$X \cdot Y \cdot Z'$	$X' + Y' + Z$
7	1	1	1	F(1,1,1)	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

Row	X	Y	Z	F	Minterm	Maxterm
0	0	0	0	F(0,0,0)	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	F(0,0,1)	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	F(0,1,0)	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	F(0,1,1)	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	F(1,0,0)	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	F(1,0,1)	$X \cdot Y' \cdot Z$	$X' + Y + Z'$
6	1	1	0	F(1,1,0)	$X \cdot Y \cdot Z'$	$X' + Y' + Z$
7	1	1	1	F(1,1,1)	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

• on sets and off sets

- the minterm list that “turns on” an output function is called the *on set*
- the maxterm list that “turns off” an output function is called the *off set*

1. The **ON set** for a 3-input NAND gate (with inputs X, Y, and Z) is:

A.  $\Sigma_{X,Y,Z}(7)$   
 B.  $\Sigma_{X,Y,Z}(0)$   
 C.  $\Sigma_{X,Y,Z}(0,1,2,3,4,5,6)$   
 D.  $\Sigma_{X,Y,Z}(1,2,3,4,5,6,7)$   
 E. none of the above

X	Y	Z	$F_{NAND}(X,Y,Z)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

2. The **OFF set** for a 3-input NOR gate (with inputs X, Y, and Z) is:

A.  $\Pi_{X,Y,Z}(7)$   
 B.  $\Pi_{X,Y,Z}(0)$   
 C.  $\Pi_{X,Y,Z}(0,1,2,3,4,5,6)$   
 D.  $\Pi_{X,Y,Z}(1,2,3,4,5,6,7)$   
 E. none of the above

X	Y	Z	$F_{NOR}(X,Y,Z)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

3. If the function  $F(X,Y,Z)$  is represented by the **ON SET**  $\Sigma_{X,Y,Z}(0,3,5,6)$ , then the **complement** of this function  $F'(X,Y,Z)$  is represented by the **ON SET**:

A.  $\Sigma_{X,Y,Z}(0,3,5,6)$   
 B.  $\Sigma_{X,Y,Z}(1,2,4,7)$   
 C.  $\Sigma_{X,Y,Z}(1,2,4,6)$   
 D.  $\Sigma_{X,Y,Z}(1,3,5,7)$   
 E. none of the above

X	Y	Z	$F(X,Y,Z)$	$F'(X,Y,Z)$
0	0	0	1	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	0	1

4. If the function  $F(X,Y,Z)$  is represented by the **ON SET**  $\Sigma_{X,Y,Z}(0,3,5,6)$ , then the **dual** of this function  $F^D(X,Y,Z)$  is represented by the **ON SET**:

A.  $\Sigma_{X,Y,Z}(0,3,5,6)$   
 B.  $\Sigma_{X,Y,Z}(1,2,4,7)$   
 C.  $\Sigma_{X,Y,Z}(1,2,4,6)$   
 D.  $\Sigma_{X,Y,Z}(1,3,5,7)$   
 E. none of the above

**DUAL truth table rule:**  
 “flip and complement”

X	Y	Z	$F(X,Y,Z)$	$F^D(X,Y,Z)$
0	0	0	1	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	0	1

• combinational analysis

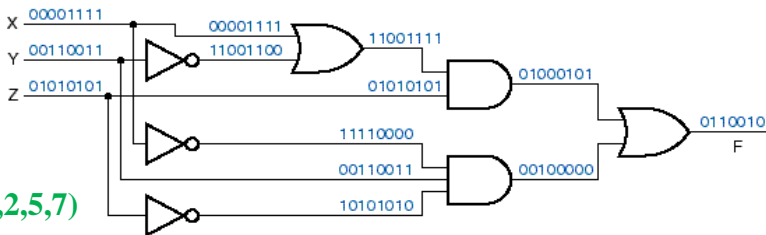
- truth table

- on set:  $\Sigma_{X,Y,Z}(1,2,5,7)$

- canonical sum:  $f(X,Y,Z) = X' \cdot Y' \cdot Z + X' \cdot Y \cdot Z' + X \cdot Y' \cdot Z + X \cdot Y \cdot Z$

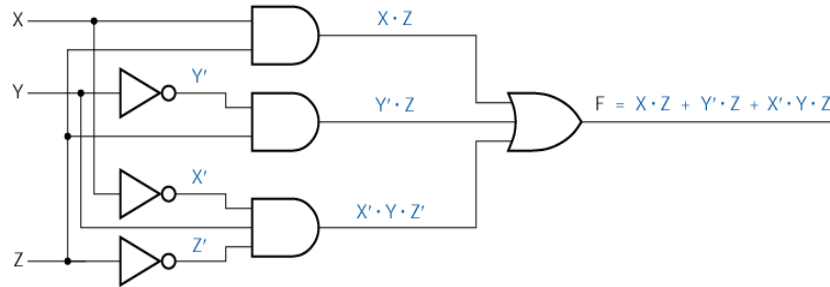
- off set:  $\Pi_{X,Y,Z}(0,3,4,6)$

- canonical product:  $f(X,Y,Z) = (X+Y+Z) \cdot (X+Y'+Z') \cdot (X'+Y+Z) \cdot (X'+Y'+Z)$

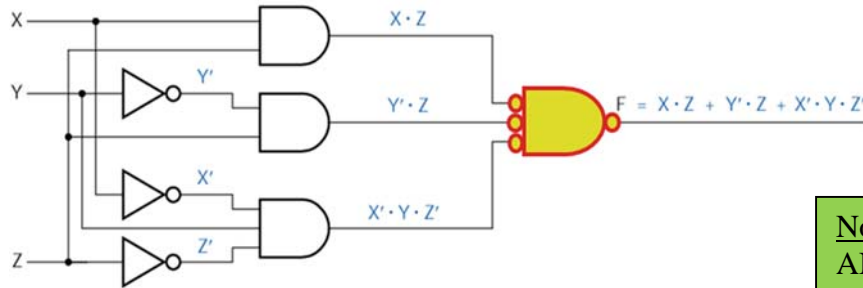


Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

graphical application of DeMorgan’s law

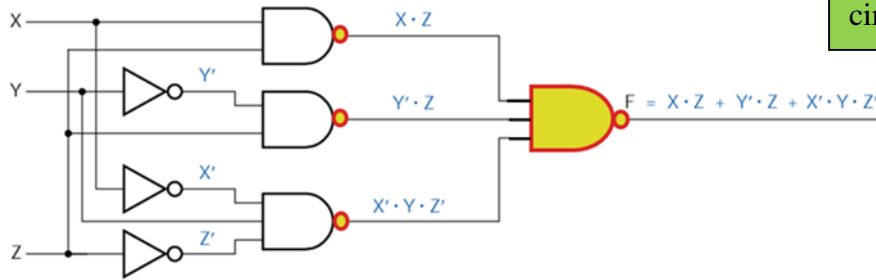


- step 1 – starting at the “output end”, replace the original gate with its dual symbol and complement all its inputs and outputs



Note: A two-level AND-OR circuit is equivalent to a two-level NAND-NAND circuit

- step 2 – migrate the “inversion bubbles” by applying involution



1. A NOR gate is **logically equivalent** to:

- A. an AND gate with inverted inputs
- B. an OR gate with inverted inputs
- C. a NAND gate with inverted inputs
- D. a NOR gate with inverted inputs
- E. none of the above

43

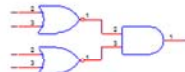
2. An OR gate is **logically equivalent** to:

- A. an AND gate with inverted inputs
- B. an OR gate with inverted inputs
- C. a NAND gate with inverted inputs
- D. a NOR gate with inverted inputs
- E. none of the above

44

3. A circuit consisting of a level of **NOR gates** followed by a level of **AND gates** is **logically equivalent** to:

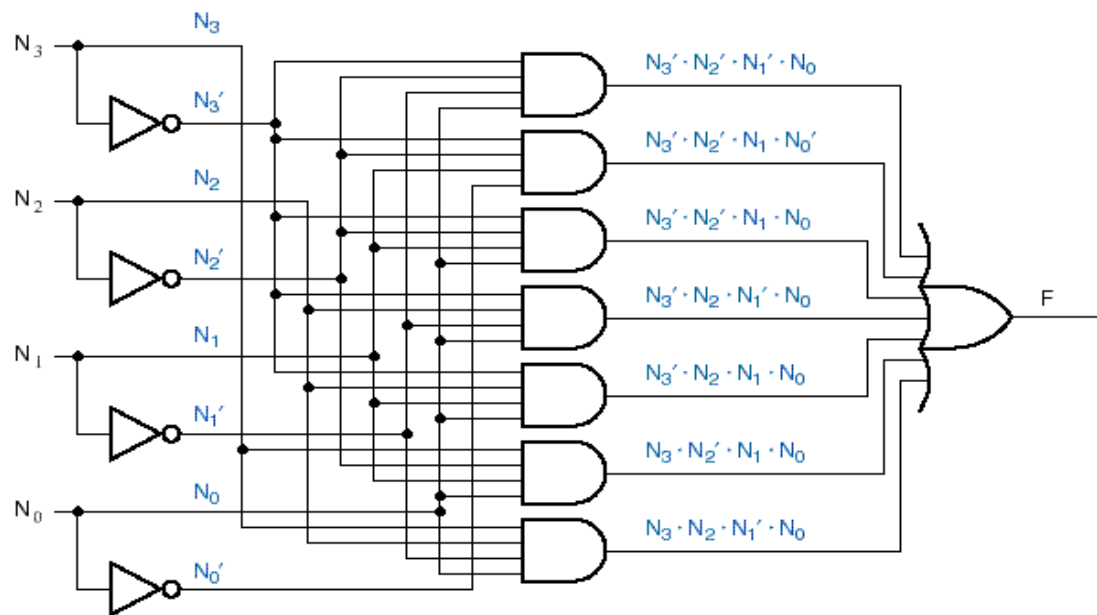
- A. a multi-input OR gate
- B. a multi-input AND gate
- C. a multi-input NOR gate
- D. a multi-input NAND gate
- E. none of the above



45

• **combinational synthesis**

- a circuit *realizes* (“makes real”) an expression if its output function equals that expression (the circuit is called a *realization* of the function)
- typically there are *many possible realizations* of the same function
  - algebraic transformations
  - graphical transformations
- starting point is often a “word description”
  - **example:** Design a 4-bit prime number detector (or, Given a 4-bit input combination  $M = N_3N_2N_1N_0$ , design a function that produces a “1” output for  $M = 1, 2, 3, 5, 7, 11, 13$  and a “0” output for all other numbers)
  - $f(N_3, N_2, N_1, N_0) = \Sigma_{N_3, N_2, N_1, N_0}(1, 2, 3, 5, 7, 11, 13)$



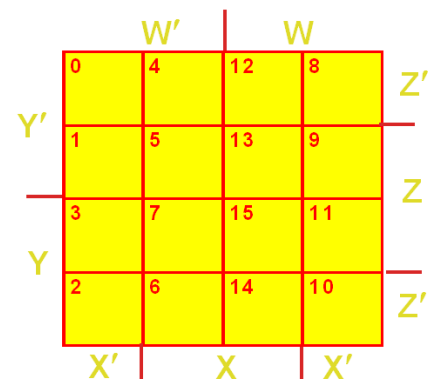
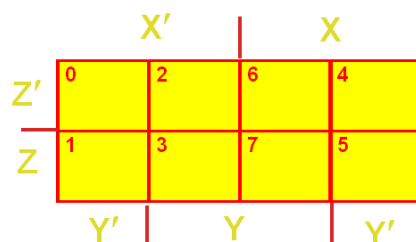
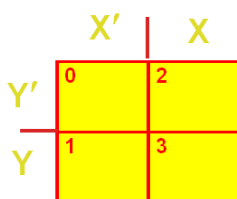
- but...how do we know if a given realization of a function is “best” in terms of:
  - speed (propagation delay)
  - cost
    - total number of gates
    - total number of gate inputs (fan-in)
- Need two things:
  - a way to transform a logic function to its simplest form (“minimization”)
  - a way to calculate the “cost” of different realizations of a given function in order to compare them

## Lecture Summary – Module 2-B

### Mapping and Minimization

**Reference:** *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 210-222, 5<sup>th</sup> Ed. pp. 117-125

- overview of minimization
  - minimization is an important step in both ASIC (*application specific integrated circuit*) design and in PLD-based (*programmable logic device*) design
  - minimization reduces the cost of two-level AND-OR, OR-AND, NAND-NAND, NOR-NOR circuits by:
    - minimizing the number of first-level gates
    - minimizing the number of inputs on each first-level gate
    - minimizing the number of inputs on the second-level gate
  - most minimization methods are based on a generalization of the Combining Theorems
  - limitations of minimization methods
    - no restriction on *fan-in* is assumed (i.e., the total number of inputs a gate can have is assumed to be “infinite”)
    - minimization of a function of more than 4 or 5 variables is *not practical* to do “by hand” (a computer program must be used!)
    - both *true and complemented* versions of all input variables are assumed to be readily available (i.e., the cost of input inverters is not considered)
- Karnaugh (K) maps
  - a Karnaugh map ( “K-map”) is a graphical representation of a logic function’s truth table (an array with  $2^n$  cells, one for each minterm)
  - features
    - minterm correspondence
    - on set correspondence
    - relationship between pairs of adjacent squares – differ by *only one literal*
    - sides of map are contiguous
    - combination of adjacent like squares in groups of  $2^k$
  - practice drawing 2-, 3-, and 4-variable K-maps



• minimization

○ theory / terminology

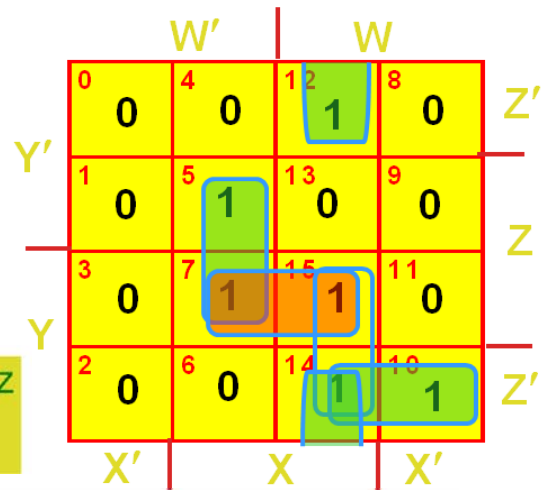
- the *minimal sum* has the fewest possible product terms (first-level gates / second-level gate inputs) and the fewest possible literals (first-level gate inputs)
- prime implicants
  - a *prime implicant* is the largest possible grouping of size  $2^k$  adjacent, like squares
  - an *essential prime implicant* has at least one square in the grouping *not shared* by another prime implicant, i.e., it has at least one “unique” square, called a *distinguished 1-cell*
  - a *non-essential prime implicant* is a grouping with no unique squares
- the *cost criterion* we will use is that *gate inputs and outputs are of equal cost*

**COST = No. of Gate Inputs + No. of Gate Outputs**

○ procedure for finding minimal SoP expression → NAND-NAND realization

- step 1 - circle all the **prime implicants**
- step 2 - note the **essential prime implicants**
- step 3 - if there are still any uncovered squares, include **non-essential prime implicants**
- step 4 - write a minimal, non-redundant sum-of-products expression
- (revisit step 3)

$$f(W,X,Y,Z) = W' \cdot X \cdot Z + W \cdot X \cdot Z' + W \cdot Y \cdot Z' + X \cdot Y \cdot Z$$



		W'		W		
Y'		0	1	0	0	Z'
		0	1	1	1	Z
Y		1	1	1	0	
		0	0	1	0	Z'
		X'	X	X'		

1. The number of *prime implicants* is:

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

		W'		W		
		0	1	0	0	Z'
Y'		0	1	1	1	Z
		1	1	1	0	
Y		0	0	1	0	Z'
		X'	X		X'	

2. The number of **essential prime implicants** is:

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

86

		W'		W		
		0	1	0	0	Z'
Y'		0	1	1	1	Z
		1	1	1	0	
Y		0	0	1	0	Z'
		X'	X		X'	

3. The number of **non-essential prime implicants** is:

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

87

		W'		W		
		0	1	0	0	Z'
Y'		0	1	1	1	Z
		1	1	1	0	
Y		0	0	1	0	Z'
		X'	X		X'	

4. The number of **product terms** in the **minimal sum** is:

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

88



		W'		W		
		0	4	12	8	Z'
Y'		0	1	0	0	
		1	5	13	9	Z
Y'		0	1	1	1	
		3	7	15	11	Z
Y		1	1	1	0	
		2	6	14	10	Z'
Y		0	0	1	0	
		X'	X	X'		

5. The ON SET for this function:

- A.  $\sum_{W,X,Y,Z}(2,4,5,6,9,10,11,12)$
- B.  $\sum_{W,X,Y,Z}(3,4,5,7,9,13,14,15)$
- C.  $\sum_{W,X,Y,Z}(3,4,5,7,9,10,11,13)$
- D.  $\sum_{W,X,Y,Z}(2,4,5,6,9,13,14,15)$
- E. none of the above

89

- procedure for finding minimal PoS expression: group 0's to find a minimal SoP expression for  $f'$ ; then find an SoP expression for  $f$  by applying DeMorgan's Law → NOR-NOR realization

		W'		W		
		0	4	12	8	Z'
Y'		1	0	1	1	
		1	1	1	1	Z
Y'		1	1	1	1	
		3	7	15	11	Z
Y		0	1	0	0	
		2	6	14	10	Z'
Y		0	0	0	0	
		X'	X	X'		

$f' = W \cdot Y + X' \cdot Y + W' \cdot X \cdot Z'$

Apply DeMorgan's Law

$f = (W' + Y') \cdot (X + Y') \cdot (W + X' + Z)$

- procedure for handling NAND-wired AND configuration: note that this configuration realizes the *complement* of a NAND-NAND circuit, so find a minimal SoP expression for  $f'$  (by grouping 0's) and implement these terms "directly" (as if it were NAND-NAND)
- procedure for converting from one form of an expression to another: use a K-map!
- procedure for handling *incompletely specified* functions (logic functions that do not assign a specific binary output value (0/1) to each of the  $2^n$  input combinations)
  - unused combinations – called "don't cares" or "d-set"
  - rules for grouping
    - allow d's to be included when circling sets of 1's, to make the sets as large as possible
    - do *not* circle any sets that contain *only* d's
    - look for distinguished 1-cells only, *not* distinguished d-cells

	X'		X	
Z'	1	1	0	d
Z	0	0	1	0
	Y'	Y	Y'	

1. The **cost** of a **minimal sum of products** realization of this function (assuming **both true and complemented variables** are available) is:  
**A. 9 B. 10 C. 11 D. 12 E. none of the above**

121

	X'		X	
Z'	1	1	0	d
Z	0	0	1	0
	Y'	Y	Y'	

2. The **cost** of a **minimal products of sum** realization of this function (assuming **both true and complemented variables** are available) is:  
**A. 9 B. 10 C. 11 D. 12 E. none of the above**

122

	X'		X	
Z'	1	1	0	d
Z	0	0	1	0
	Y'	Y	Y'	

3. Assuming the availability of **only true** input variables, the **fewest number of 2-input NAND gates** that are needed to realize this function is:  
**A. 6 B. 7 C. 8 D. 9 E. none of the above**

123

		X'		X	
Z'		1	1	0	d
Z		0	0	1	0
		Y'		Y	Y'

4. Assuming the availability of **only true** input variables, the **fewest number of 2-input NOR gates** that are needed to realize this function is:  
**A. 6 B. 7 C. 8 D. 9 E. none of the above**

124

		X'		X	
Z'		1	1	0	d
Z		0	0	1	0
		Y'		Y	Y'

5. Assuming the availability of **only true** input variables, the **fewest number of 2-input open-drain NAND gates** that are needed to realize this function is:  
**A. 6 B. 7 C. 8 D. 9 E. none of the above**

125

		X'		X	
Z'		1	1	0	d
Z		0	0	1	0
		Y'		Y	Y'

6. The **number of pull-up resistors** required for realizing this function **using only 2-input open drain NAND gates** (assuming the availability of **only true** input variables) is:  
**A. 1 B. 2 C. 3 D. 4 E. none of the above**

126

## Lecture Summary – Module 2-C

### Timing Hazards

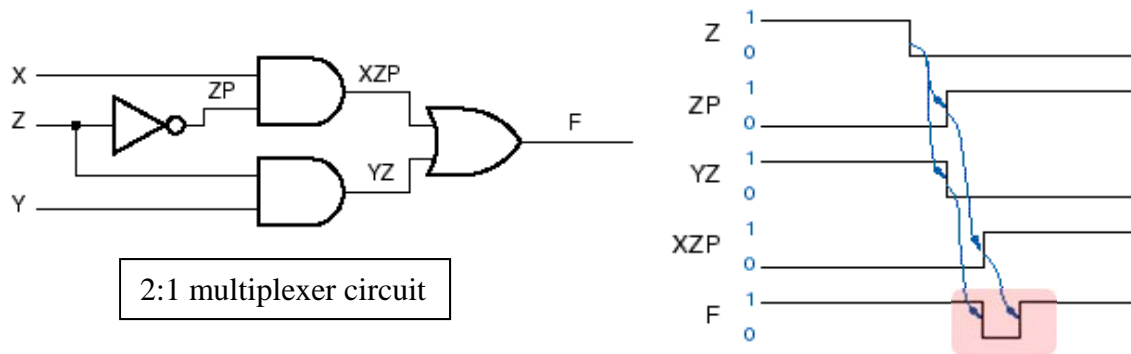
**Reference:** *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 224-229, 5<sup>th</sup> Ed. pp. 122-126

- introduction

- gate propagation delay may cause the transient behavior of logic circuit to *differ* from that predicted by steady state analysis
- short pulse – glitch/hazard

- definitions

- a static-1 hazard is a *pair of input combinations* that: (a) differ in only one input variable and (b) both produce a “1” output, such that it is possible for a momentary “0” output to occur during a transition in the differing input variable

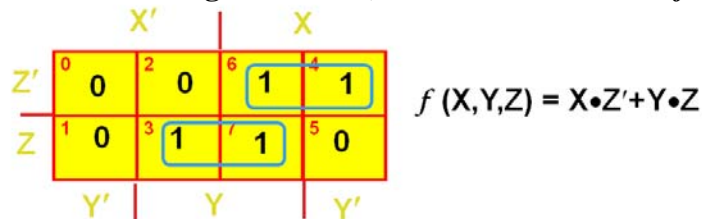


2:1 multiplexer circuit

- a static-0 hazard is a *pair of input combinations* that: (a) differ in only one input variable and (b) both produce a “0” output, such that it is possible for a momentary “1” output to occur during a transition in the differing input variable (the *dual* of a static-1 hazard)

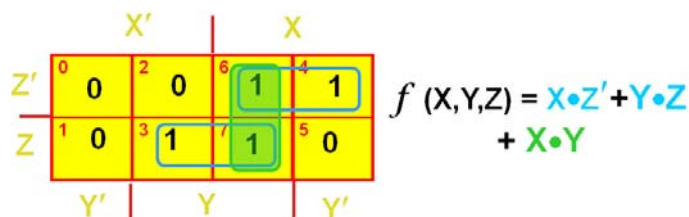
- prediction of static hazards from a K-map

- **important:** the existence or nonexistence of static hazards depends on the *circuit design* (i.e., *realization*) of a logic function
- cause of hazards: it is possible for the output to momentarily glitch to “0” if the AND gate that covers one of the combinations goes to “0” before the AND gate covering the other input combination goes to “1” (referred to as *break before make*)

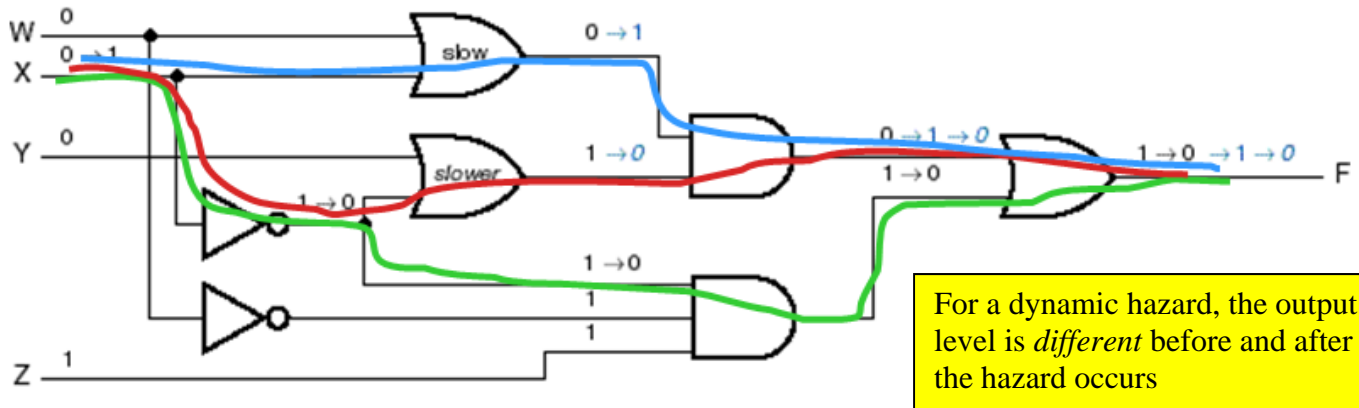


The extra product term is the *consensus* of the two original terms – in general, *consensus terms* must be added to *eliminate hazards*

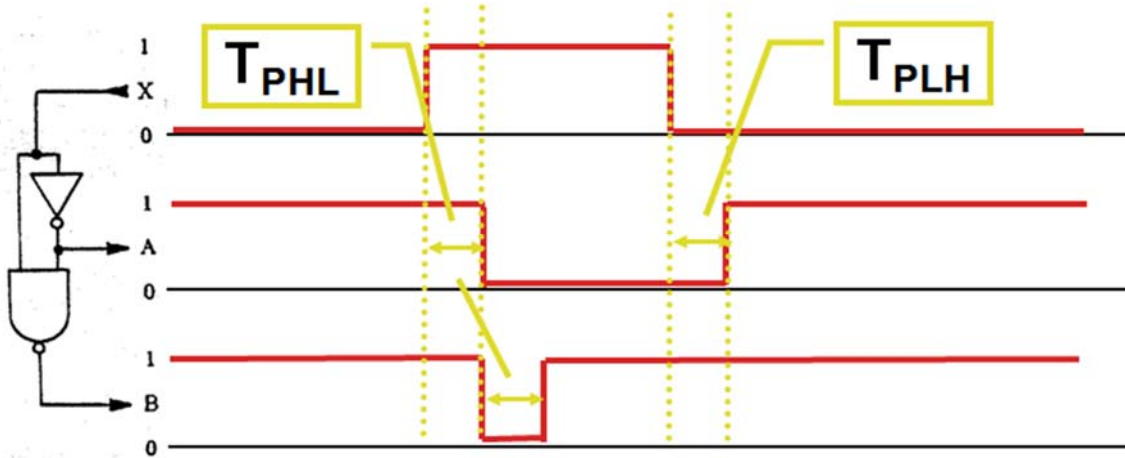
- use of consensus term(s) to eliminate hazards



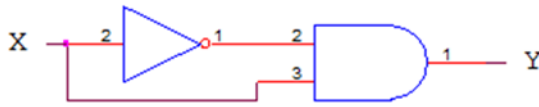
- A *dynamic hazard* is the possibility of an output changing *more than once* as the result of a single input transition (can occur if there are *multiple paths with different delays* from the changing input to the changing output)



- “useful” hazards – example: “leading edge” detector

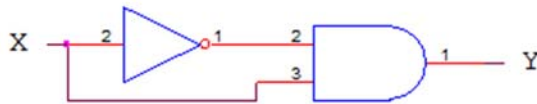


- designing hazard-free circuits
  - *very few* practical applications require the design of hazard-free combinational circuits (e.g., feedback sequential circuits)
  - techniques for finding hazards in arbitrary circuits are *difficult to use*
  - if cost is not a problem, then a “brute force” method of obtaining a hazard-free realization is to use the *complete sum* (i.e., all prime implicants)
  - functions that have non-adjacent product terms are *inherently hazardous* when subjected to simultaneous input changes



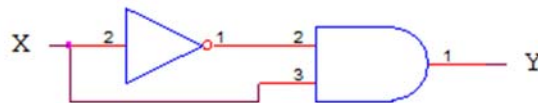
1. Steady state analysis of this circuit would predict that its output will always be:
  - A. 0
  - B. 1
  - C. 50% of  $V_{CC}$
  - D. *Les Déplorables*
  - E. none of the above

140



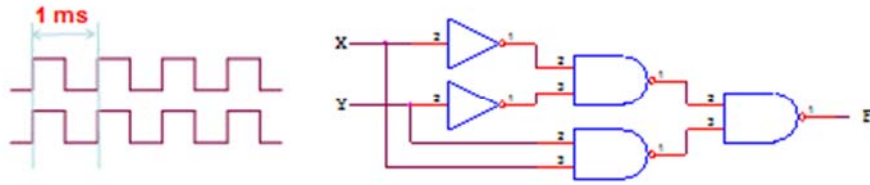
2. This circuit exhibits the following type of hazard when its input, X, transitions from low-to-high:
  - A. static-0
  - B. static-1
  - C. dynamic
  - D. *Les Déplorables*
  - E. none of the above

141



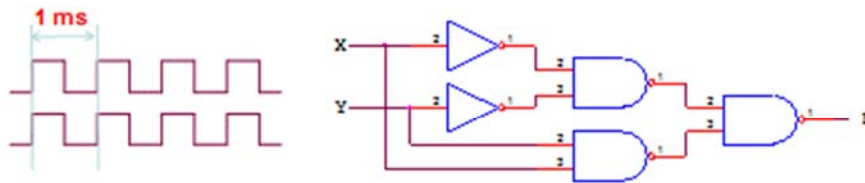
3. This circuit exhibits the following type of hazard when its input, X, transitions from high-to-low:
  - A. static-0
  - B. static-1
  - C. dynamic
  - D. *Les Déplorables*
  - E. none of the above

142



4. Steady-state analysis of the function realized by this circuit for the input waveforms shown predicts that the output  $F(X,Y)$  should:
- A. should always be low
  - B. should always be high
  - C. should be identical to the input
  - D. should be the complement of the input
  - E. none of the above

143



5. Dynamic analysis of the output  $F(X,Y)$  reveals that:
- A. a static "0" hazard will be generated in response to low-to-high transitions of the input waveform
  - B. a static "1" hazard will be generated in response to low-to-high transitions of the input waveform
  - C. a static "0" hazard will be generated in response to high-to-low transitions of the input waveform
  - D. a static "1" hazard will be generated in response to high-to-low transitions of the input waveform
  - E. none of the above

144

## Lecture Summary – Module 2-D

### XOR/XNOR Functions

**Reference:** *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 447-448, 5<sup>th</sup> Ed. pp. 320-322

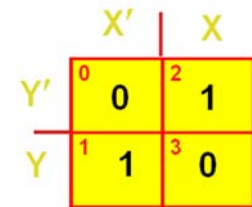
• **introduction**

- an *Exclusive-OR (XOR)* gate is a 2-input gate whose output is “1” if exactly one of its inputs is “1” (or, an XOR gate produces an output of “1” if its inputs are *different*)
- an *Exclusive-NOR (XNOR)* gate is the *complement* of an XOR gate – it produces an output of “1” if its inputs are the *same*
- an XNOR gate is also referred to as an *Equivalence* (or *XAND*) gate
- although XOR is not one of the basic functions of switching algebra, discrete XOR gates are commonly used in practice
- the “ring sum” operator  $\oplus$  is used to denote the XOR function:  $X \oplus Y = X \cdot Y' + X' \cdot Y$
- the XNOR function can be thought of as either the *dual* or the *complement* of XOR

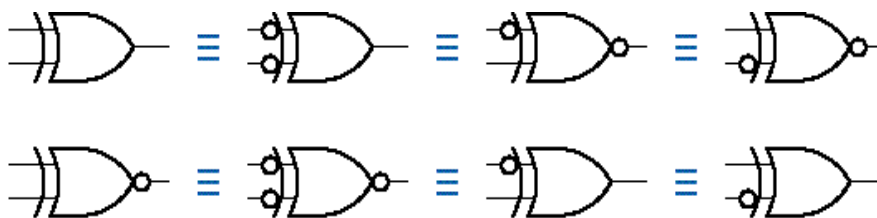
• **XOR properties**

- $X \oplus X = X' \cdot X + X \cdot X' = 0 + 0 = 0$
- $X' \oplus X' = X \cdot X' + X' \cdot X = 0 + 0 = 0$
- $X \oplus 1 = X' \cdot 1 + X \cdot 0 = X'$
- $X' \oplus 1 = X \cdot 1 + X' \cdot 0 = X$
- $(X \oplus Y)' = X \oplus Y \oplus 1$
- $X \oplus Y = Y \oplus X$
- $X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$
- $X \cdot (Y \oplus Z) = (X \cdot Y) \oplus (X \cdot Z)$

X	Y	$X \oplus Y$	$(X \oplus Y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

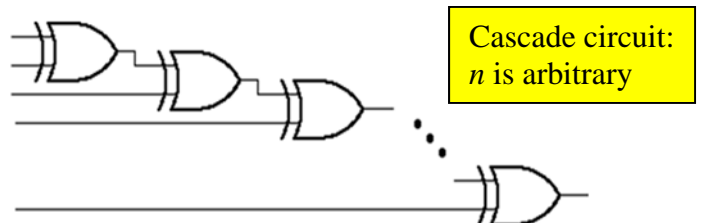
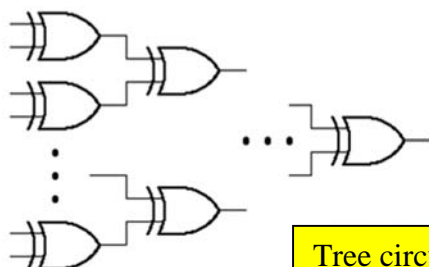


- “checkerboard” K-map → *non-reducible function*
- common/equivalent symbols



• **N-variable XOR/XNOR functions – tree and cascade circuits**

- the output of an n-variable XOR function is 1 if an odd number of inputs are 1
- the output of an n-variable XNOR function is 1 if an even number of inputs are 1
- realization of an n-variable XOR/XNOR function will require  $2^{n-1}$  P-terms





- simplification of special classes of functions using XOR/XNOR gates
  - functions that cannot be significantly reduced using conventional minimization techniques can sometimes be *simplified* by implementing them with XOR/XNOR gates
  - candidate functions that may be simplified this way have K-maps with “diagonal 1’s”
  - **Technique:** Write out function in SoP form, and “factor out” XOR/XNOR expressions

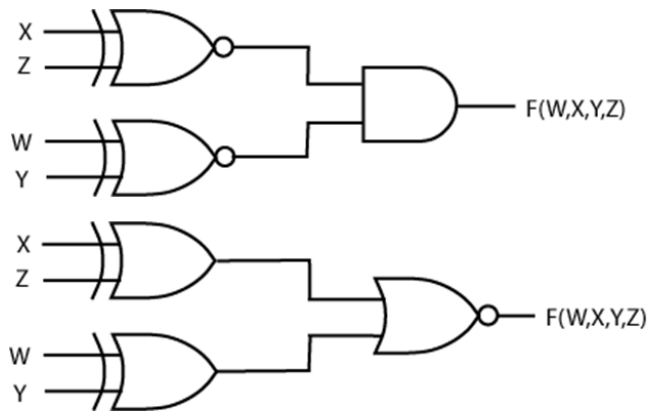
		W'		W	
	0	4		12	8
Y'	1	0		0	0
1	0	1		0	0
3	0	0		1	0
2	0	0		0	1
	X'			X	X'

**F(W,X,Y,Z) =**  
 $W' \cdot X' \cdot Y' \cdot Z' + W' \cdot X \cdot Y' \cdot Z$   
 $+ W \cdot X \cdot Y \cdot Z + W \cdot X' \cdot Y \cdot Z'$

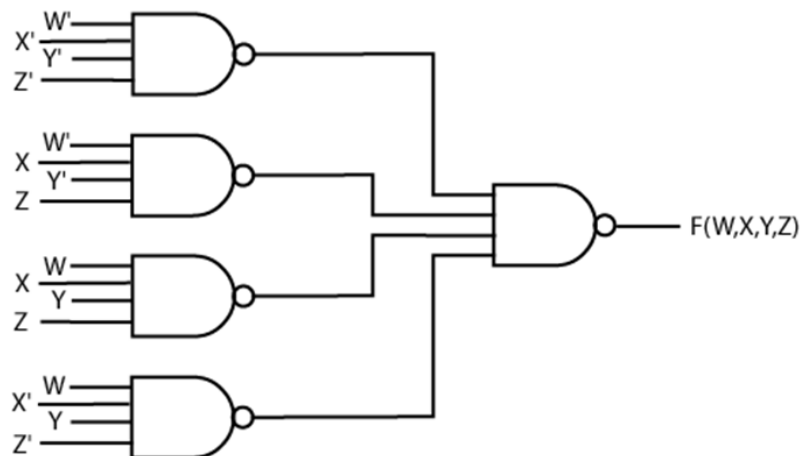
**Z =**  
 $X' \cdot Z' \cdot (W' \cdot Y' + W \cdot Y)$   
 $+ X \cdot Z \cdot (W' \cdot Y' + W \cdot Y)$

**Z' =**  
 $(X' \cdot Z' + X \cdot Z) \cdot (W' \cdot Y' + W \cdot Y)$

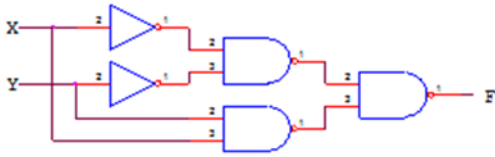
**=**  
 $(X \oplus Z)' \cdot (W \oplus Y)'$



**COST = 6 inputs + 3 outputs = 9**

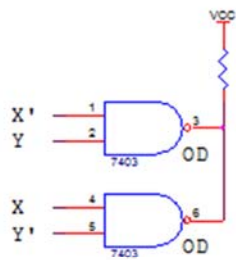


**COST = 20 inputs + 5 outputs = 25**



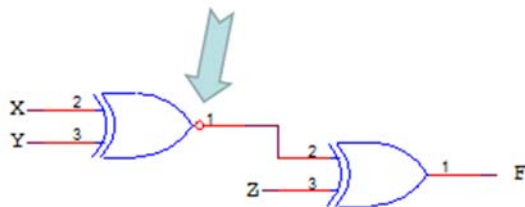
1. The function realized by this circuit is a:
- A. 2-input XOR
  - B. 2-input XNOR
  - C. 2-input AND
  - D. 2-input OR
  - E. none of the above

166



2. The ON set of the function realized by this circuit is:
- A.  $\Sigma_{X,Y}(0,2)$
  - B.  $\Sigma_{X,Y}(0,3)$
  - C.  $\Sigma_{X,Y}(1,2)$
  - D.  $\Sigma_{X,Y}(1,3)$
  - E. none of the above

167



3. The ON set of the function realized by this circuit is:
- A.  $\Sigma_{X,Y,Z}(0,3,4,7)$
  - B.  $\Sigma_{X,Y,Z}(1,2,5,6)$
  - C.  $\Sigma_{X,Y,Z}(0,3,5,6)$
  - D.  $\Sigma_{X,Y,Z}(1,2,4,7)$
  - E. none of the above

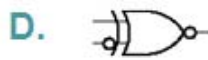
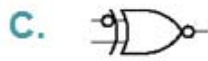
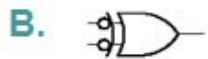
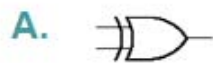
168

4. The XOR property listed below that is **NOT** true is:

- A.  $X \oplus 0 = X$
- B.  $X \oplus 1 = X'$
- C.  $X \oplus X = X$
- D.  $X \oplus X' = 1$
- E. none of the above

169

5. The following is **NOT** an equivalent symbol for an XOR gate:



- E. none of the above

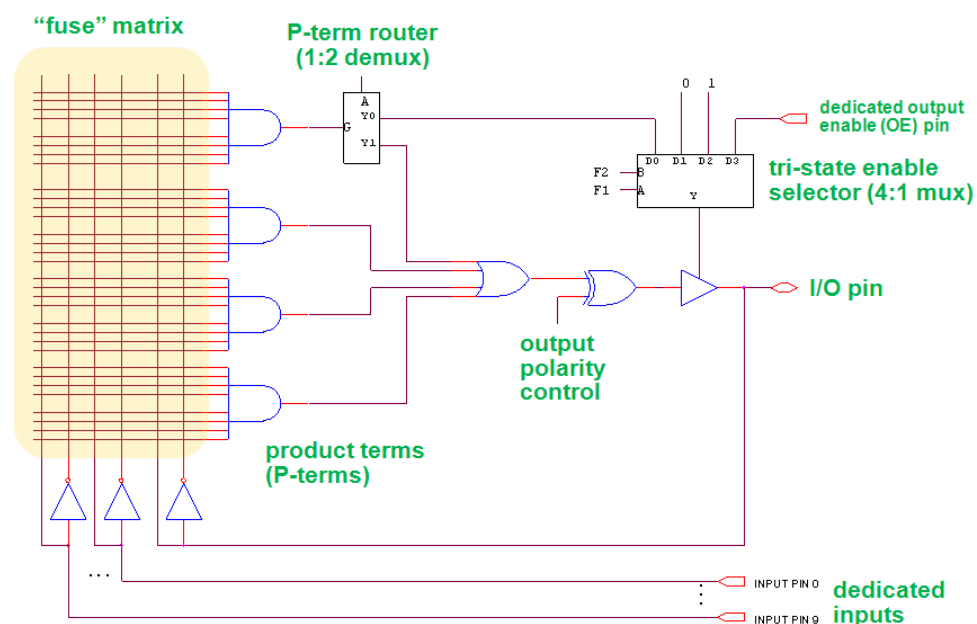
170

## Lecture Summary – Module 2-E

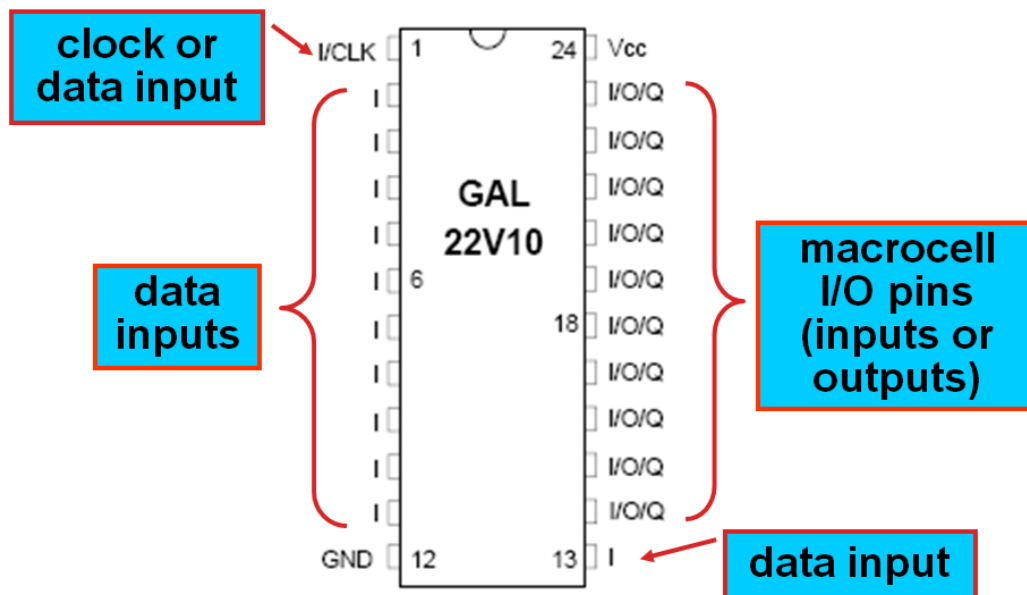
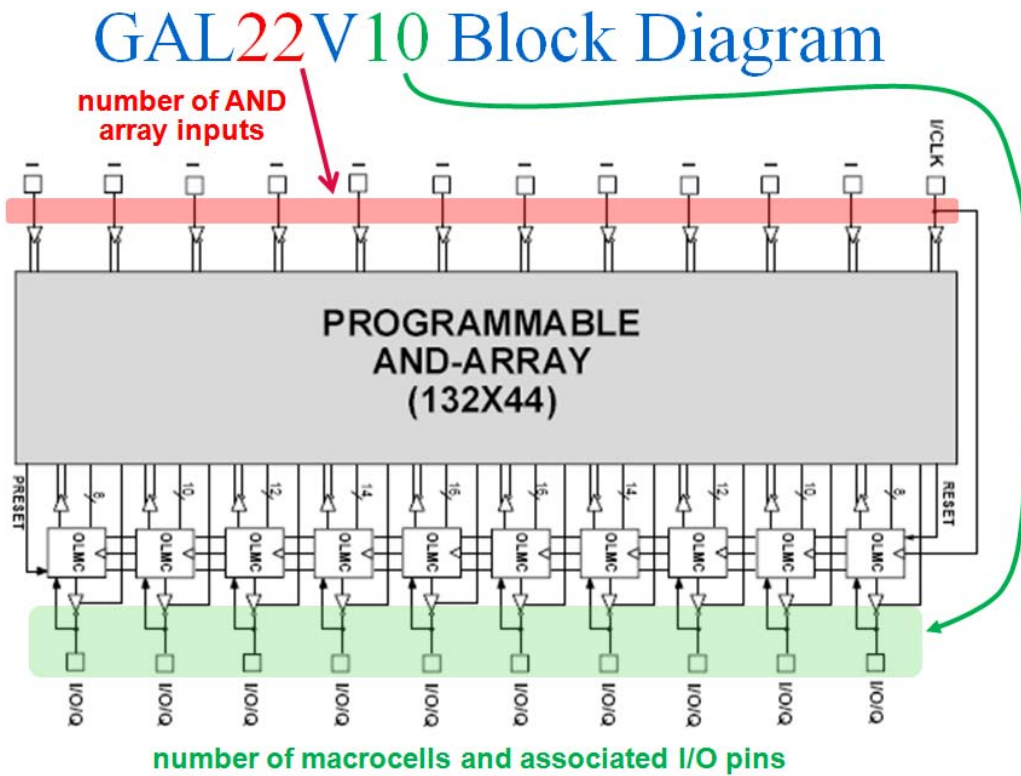
### Programmable Logic Devices

Reference: *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 370-383, 840-859; 5<sup>th</sup> Ed. pp. 246-252

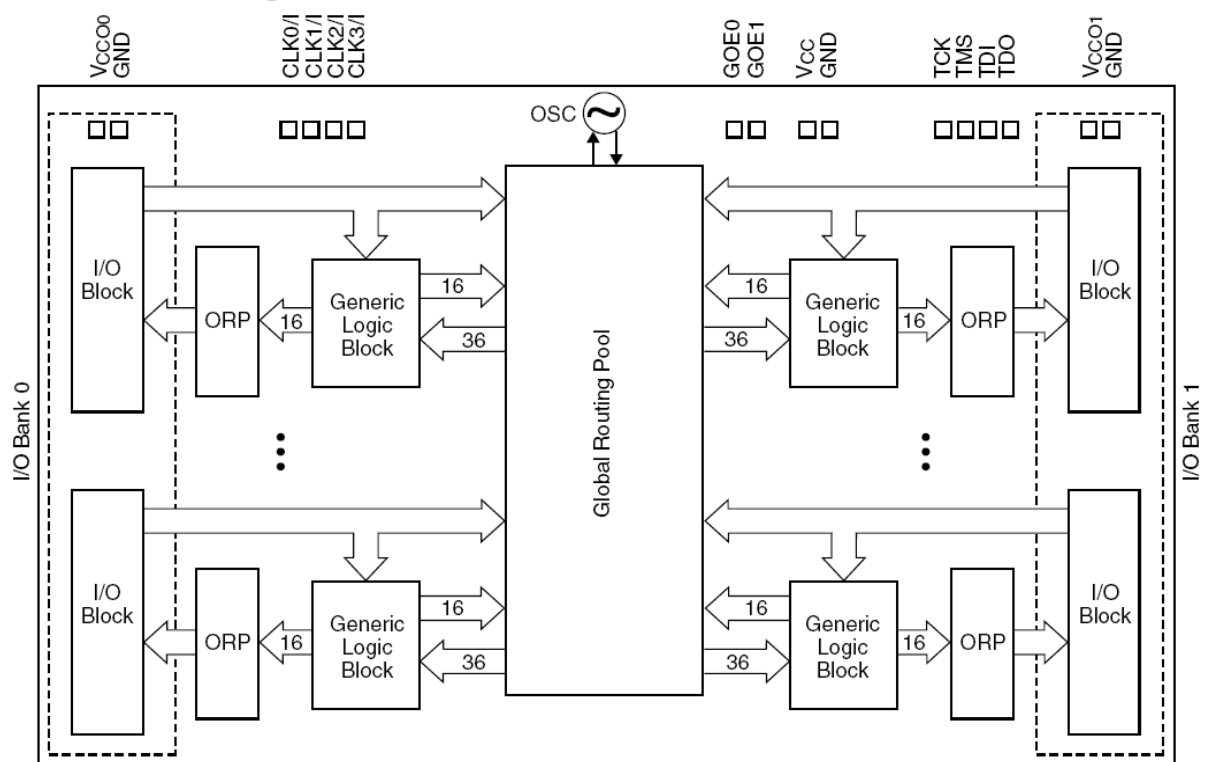
- overview -- programmable logic devices
  - first were programmable logic arrays (PLAs)
    - two-level, AND-OR, SoP
    - limitations: inputs, outputs, P-terms
    - both true and complemented version of each input available
    - connections made by “fuses” (non-volatile memory cells)
    - each AND gate’s inputs any subset of true/complemented input variables
    - each OR gate’s inputs any subset of AND gate outputs
  - special case of PLA is programmable array logic (PAL)
    - fixed OR array (AND gates cannot be shared)
    - each output includes (inverting) tri-state buffer
    - some pins may be used for either input or output (“I/O pins”)
  - generic array logic (GAL) devices (essentially PALs)
    - generic array logic (GAL) devices can be configured to emulate the AND-OR, register (flip-flop), and output structure of combinational and sequential PAL devices
    - an *output logic macrocell* (“OLMC”) is associated with each I/O pin to provide configuration control
    - OLMCs include *output polarity control* (important because it allows minimization software to “choose” *either* the SoP *or* PoS realization of a given function)
    - erasable/reprogrammable GAL devices use floating gate technology (flash memory) for “fuses” and are *non-volatile* (i.e., retain programming without power)
    - GAL devices require a “universal programmer” to erase and reprogram their so-called “fuse maps” (means that they must be *removed* for reprogramming and subsequently reinstalled – *requires a socket*)
    - GAL combinational macrocell structure



- **example** – GAL22V10



- **complex PLDs (CPLDs)**
  - **modern complex PLDs (CPLDs) contain hundreds of macrocells and I/O pins, and are designed to be erased/reprogrammed *in-circuit* (called “isp”)**
  - **because CPLDs typically contain significantly more macrocells than I/O pins, capability is provided to use macrocell resources “internally” (called a *node*)**
  - **a global routing pool (GRP) is used to connect generic logic blocks (GLBs)**
  - **output routing pools (ORPs) connect the GLBs to the I/O blocks (IOBs), which contain multiple I/O cells**
  - **example: ispMACH4000ZE-series**



- **field programmable gate arrays (FPGAs)**
  - **a field programmable gate array (FPGA) is “kind of like a CPLD turned inside-out”**
  - **logic is broken into a large number of programmable blocks called *look-up tables* (LUTs) or *configurable logic blocks* (CLBs)**
  - **programming configuration is stored in SRAM-based memory cells and is therefore *volatile*, meaning the FPGA configuration is lost when power is removed**
  - **programming information must therefore be loaded into an FPGA (typically from an external ROM chip) *each time it is powered up* (“initialization/boot” cycle)**
  - **LUTs/CLBs are inherently less capable than PLD macrocells, but *many more of them* will fit on a comparably sized FPGA (than macrocells on a CPLD)**

## Lecture Summary – Module 2-F

### Hardware Description Languages

**Reference:** *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 237-255, 5<sup>th</sup> Ed. pp. 177-233

- overview
  - hardware description languages (HDLs) are the most common way to describe the programming configuration of a CPLD or an FPGA
  - the first HDL to enjoy widespread use was PALASM (“PAL Assembler”) from Monolithic Memories, Inc. (inventors of the PAL device)
  - early HDLs only supported equation entry
  - next generation languages such as CUPL (Compiler Universal for Programmable Logic) and ABEL (Advanced Boolean Expression Language) added more advanced capabilities:
    - truth tables and clocked operator tables
    - logic minimization
    - high-level constructs such as *when-else-then* and *state diagram*
    - test vectors
    - timing analysis
  - VHDL and Verilog
    - started out as *simulation languages* (later developments in these languages allowed actual hardware design)
    - support modular, hierarchical coding and support a wide variety of high-level programming constructs – represents a *higher level of abstraction*
      - arrays
      - procedures
      - function calls
      - conditional and iterative statements
    - potential pitfall – because VHDL and Verilog have their genesis as simulation languages, it is possible to create *non-synthesizable HDL code* using them (i.e., code that can *simulate* a digital system, but *not actually realize* it)
- concentration on use of Verilog
  - You will use Verilog to program legacy PLDs (like the 22V10) as well as current generation CPLDs (like the ispMACH 4256ZE)
  - We will use the **Lattice ispLever Classic 1.8** software package in lab, which includes support for Verilog – **free copy** of software package available at [latticesemi.com](http://latticesemi.com)
  - Verilog program contents
    - documentation (program name, comments)
    - declarations that identify the inputs and outputs of the logic functions to be performed
    - statements that specify the logic functions to be performed

- Verilog program semantics
  - *identifiers* (module names, signal/variables names) must begin with a letter or underscore and can include digits and dollar signs (\$)
  - *identifiers* are case sensitive
  - single line *comments* begin with //
  - */\* comments* can also be done this way \*/
  - *input* and *output declarations* tell the compiler about symbolic names associated with the external pins of the device
  - each *assign* statement describes a small piece of logic circuitry
  - Constant values can be described as n'bxxxx where n is the bit-width of the signal and x is 0 or 1
  
- Verilog wire type
  - *wire* is a basic data type in Verilog
  - Similar to an actual wire, these variables cannot store logic value and are used to connect signals between inputs, outputs and logic elements such as gates
  - *wire* is used to model Combinational Logic
  - *wire* can take on four basic values
    - 0 – logical zero
    - 1 – logical one
    - X – Unknown value
    - Z – High-Impedance state
  
- Verilog Bitwise Operators
  - &        AND
  - |        OR
  - ^        XOR
  - ~^ or ^~ XNOR
  - ~        NOT

(similar to bitwise operators in C - you will learn about logical Verilog operators later, and the difference between the two)
  
- ispLEVER Operators
  - ispLEVER reports use different notation for some operators
  - &    AND
  - #    OR
  - !    NOT
  - \$    XOR
  - !\$   XNOR
  
- Verilog assign statements
  - *assign* statements are used to continuously assign the value of the expression on the right of the “=” to the signal on the left
  - assigning constant bits to a variable
    - assign A = 3'b110;
    - assign B = 3'b101;
  - assigning logic to a variable
    - assign X = A & B;
    - assign Y = A | B;



- Verilog Module Structure
  - *synthesis loc* is a compiler directive that tells ispLEVER to use specific pins for input or output.
  - input wire Sel, A, B /\* synthesis loc="4,5,6" \*/; tells compiler to connect Sel, A, and B, to pins 4, 5 and 6, respectively on the PLD
- example Verilog program with equations

```
/* Verilog Combinational Example for GAL22V10 */
```

```
module verilog_exA(A,B,C,D,X,Y,Z);
```

```
  input A,B,C,D /* synthesis loc="2,3,4,5" */;
  output X,Y,Z /* synthesis loc="14,15,16" */;
```

```
  // dataflow style logic equations
```

```
  assign X = (A & B) | ~(C & D);
```

```
  assign Y = ~(B & D) | ~(A & B & D);
```

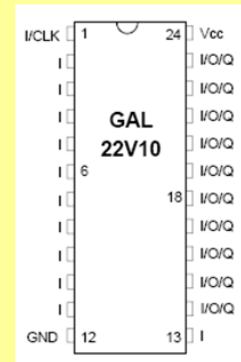
```
  assign Z = A & ~(B & C & ~D);
```

```
  // use parenthesis for readability
```

```
  // and to make sure order of operations
```

```
  // (precedence) are as intended
```

```
endmodule
```



**Note:** Explicit pin declarations can be omitted and automatically assigned by the “fitter” program (part of ispLever)

- reg data types
  - similar to *wire*, but *reg* can be used to store information like registers
  - unlike *wire*, “*reg*” can be used to model both Combinational and Sequential logic
  - For behavioral code using an *always* block, the output must be type *reg*
  - For dataflow code with *assign* statements, the outputs must be of type *wire*
- always block
  - always block lets you write “behavioral” style code, similar to C
  - should have a sensitivity list associated with it, all statements in the always block will be evaluated when the conditions in this list are triggered
  - conditions may be any change to the signal or rising or falling edges of the signals
  - examples of always block:
    - always @ (A,B,C) begin
      - all statements will be evaluated whenever A, B, or C change their values
    - always @ (posedge CLK) begin
      - all statements will be evaluated on the positive (rising) edge of the CLK signal
    - always @ (\*) begin
      - all statements will be evaluated whenever any input signal in the always block changes

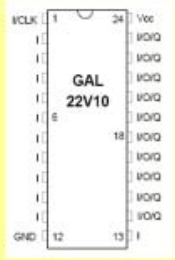
- case syntax
  - similar to the case structure in C
  - compares expression to a set of cases and evaluates the statement(s) associated with first matching case
  - all cases defined between *case* (signal) .. *endcase*
  - multiple statements for a *case* must be enclosed in a *begin* and *end* block
  - multiple comparison signals can be concatenated as *case* ({signal1, signal2 ...signaln}) and compared against values of their total bit width
  - If the logic does not cover all possible bit combinations of the comparison signal(s), a default case must be added.
- example Verilog program with truth table

### Example Verilog Module #2

```

/* Truth table example */
module ttex(E,R,S,T,A,B,C,D,F);
input  E,R,S,T /* synthesis loc="2,3,4,5" */;
output A,B,C,D,F /* synthesis loc="14,15,16,17,18" */;
reg [4:0] abcdf /* bit vector to assign to output pins */;

always @(E,R,S,T) begin
    case ({E,R,S,T})
        4'b0000: abcdf = 5'b010000;
        4'b0001: abcdf = 5'b000100;
        4'b0010: abcdf = 5'b001000;
        4'b0011: abcdf = 5'b000100;
        4'b0100: abcdf = 5'b100000;
        4'b0101: abcdf = 5'b100000;
        4'b0110: abcdf = 5'b001000;
        4'b0111: abcdf = 5'b100000;
        4'b1000: abcdf = 5'b010000;
        4'b1001: abcdf = 5'b010000;
        4'b1010: abcdf = 5'b001000;
        4'b1011: abcdf = 5'b000001;
        4'b1100: abcdf = 5'b100000;
        4'b1101: abcdf = 5'b100000;
        4'b1110: abcdf = 5'b001000;
        4'b1111: abcdf = 5'b100000;
    endcase
end
assign {A,B,C,D,F} = abcdf;
endmodule
                
```



Compares each case against a concatenated 4-bit vector with E at bit position 3, R at position 2, S at position 1 and T at position 0  
 e.g. 4'b1011 matches E=1,R=0,S=1,T=1

assign A = abcdf[4], B = abcdf[3], etc.

- example Verilog program with multi-input XOR

### Example Verilog Program #3C

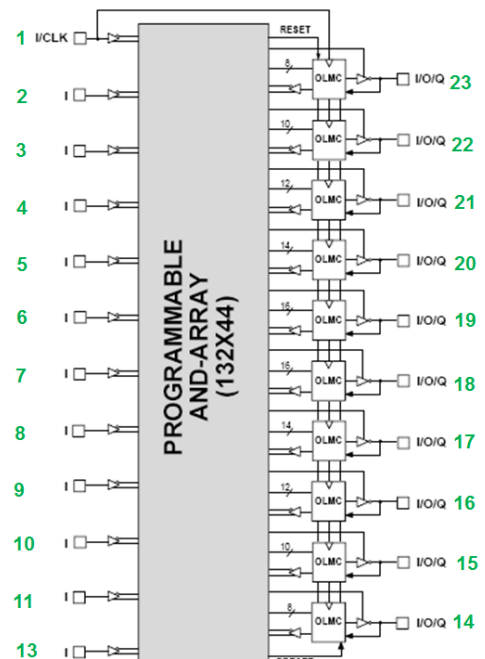
```

/* 10 variable XOR on 22V10 */
module xor_exC(I,X,Y,Z)
input [9:0] I;
output X,Y,Z /* synthesis loc="18,19,23" */;
wire Xi,Yi;

// notice the index ranges
assign Xi= ^I[4:0]; // 16 P-terms
assign Yi= ^I[9:5]; // 16 P-terms
assign Z = Xi^Yi; // 2 P-terms

// outputs can't be directly used
// like an input inside the code
assign X = Xi;
assign Y = Yi;
endmodule
                
```

NOTE: Requires two "passes" through the PLD (which doubles the propagation delay)



- **structural code**

- relies on instantiating every module and connecting their inputs and outputs manually
- logic can be described without the use of boolean operators, logical constructs (if-else, case), always blocks or assign statements
  - `module_name instance_name (signal_list);` will instantiate a module of type `module_name` called `instance_name` (the `signal_list` corresponds to the inputs and outputs, also called the port list)
  - `and AND2 (XY, X, Y);` will instantiate an AND gate with inputs X and Y with output XY
  - `xor OR (X_Y,X,Y);` will instantiate a 2-input XOR gate
- built-in primitives: `and`, `or`, `nand`, `nor`, `xor`, `xnor`, `not`, `buf`, `bufif0`, `bufif1`, `notif0`, `notif1`
- example:

```

module structural_ex(A,B,C,D,X,Y);
  input wire A, B, C, D;
  output wire X, Y;
  wire AB, CD;
  and AND2a (AB, A, B); // AB = A & B
  and AND2b (CD, C, D); // CD = C & D
  or OR2a (X, AB, CD); // X = AB | CD
  assign Y = (A & B) | (C & D);
endmodule

```

*X and Y evaluate the same function*

**X : Structural style/code**

**Y : Dataflow style/code**

1. Which of the following is **not** a valid Verilog identifier?

- A. X2
- B. 2X
- C. XY
- D. \_XY
- E. none of the above

2. Which of the following specifies a range of bits within a bit vector X in Verilog?

- A. X3..X1
- B. X(3:1)
- C. [3:1]
- D. X[3:1]
- E. none of the above

3. For **input or output port declarations**, which of the following statements is **not** true?

- A. "synthesis loc" declarations associate the device's physical pins with symbolic port names
- B. pin numbers are optional
- C. if pin numbers are not specified, the pin numbers are assigned by the "fitter" program based on the PLD characteristics
- D. the pin may be declared active high or active low
- E. none of the above

4. The **order** in which different **assign expressions** are placed in the body of a Verilog module **does not matter**.

- A. true
- B. false

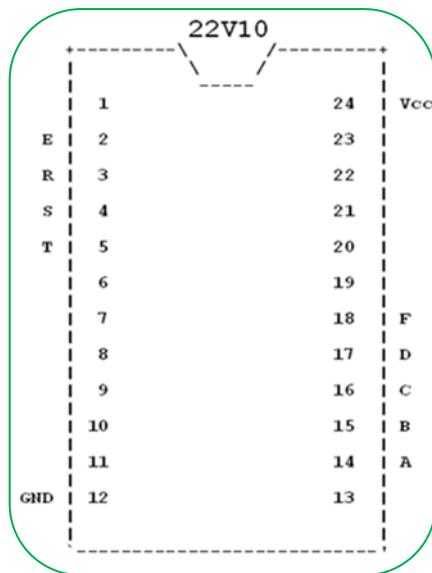
- detailed example – “crazy grader” (a.k.a. “arbitrary uniformed grading hack” –or– “AUGH”)
  - four input variables (E, R, S, T)
  - five output functions (A, B, C, D, F)
  - “stick built” (using SSI parts) vs. GAL22V10 (programmed using Verilog)

```

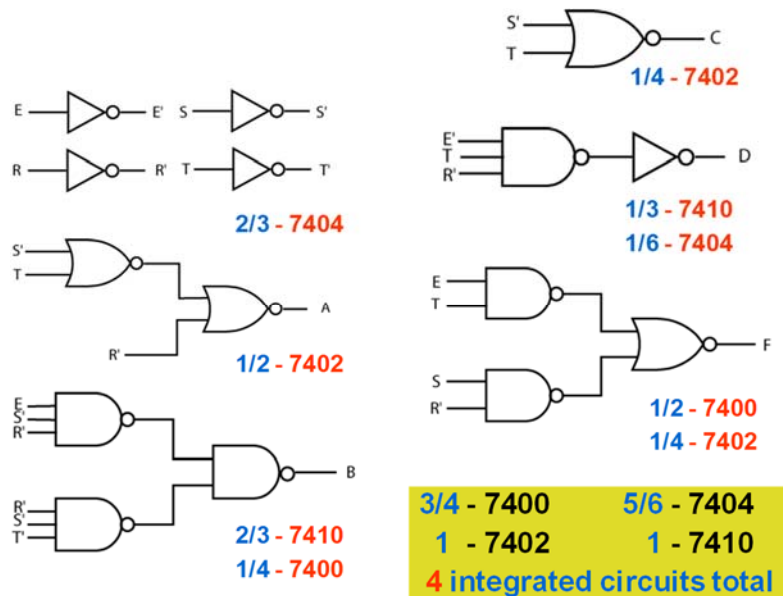
/* Who Wants to be a Digijock */
module gameshow(E,R,S,T,A,B,C,D,F);
  input wire E,R,S,T /* synthesis loc="2,3,4,5" */;
  output wire A,B,C,D,F /* synthesis loc="14,15,16,17,18" */;
  /* Quick and easy way in Verilog */
  /* ..."by inspection" from problem statement */
  assign A = (R & T) | (R & ~S);
  assign B = (E & ~R & ~S) | (~R & ~T & ~S);
  assign C = S & ~T;
  assign D = T & ~E & ~R;
  assign F = ~A & ~B & ~C & ~D;
  // or assign F = E & S & T & ~R;
endmodule
    
```

		E'		E	
	0	4	12	8	
S'	B	A	A	B	T'
	1	5	13	9	
	D	A	A	B	T
	3	7	15	11	
S	D	A	A	F	T'
	2	6	14	10	
	C	C	C	C	T'
	R'		R		R'

VS. ×5



VS.



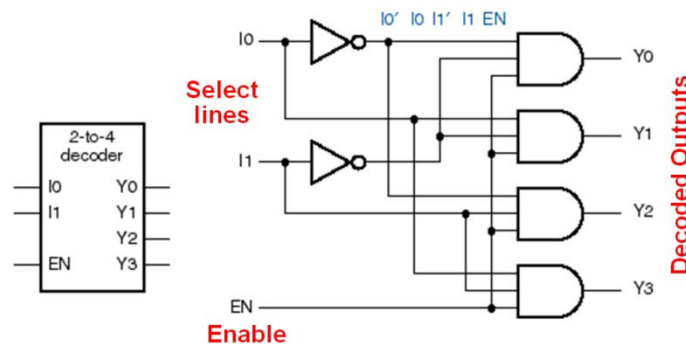
## Lecture Summary – Module 2-G

### Combinational Building Blocks: Decoders/Demultiplexers

**Reference:** *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 384-398, 5<sup>th</sup> Ed. pp. 250-256, 260-278

- overview

- a decoder is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs (in a one-to-one mapping, each input code word produces a different output code word)
- n-bit binary input code most common
- 1-out-of-m output code most common (note: output code bits are *mutually exclusive*)
- binary decoder: n to 2<sup>n</sup> (n-bit binary input code, 1-out-of-2<sup>n</sup> output code)
- example: 2-to-4 (2:4) binary decoder *or* 1-to-4 (1:4) demultiplexer

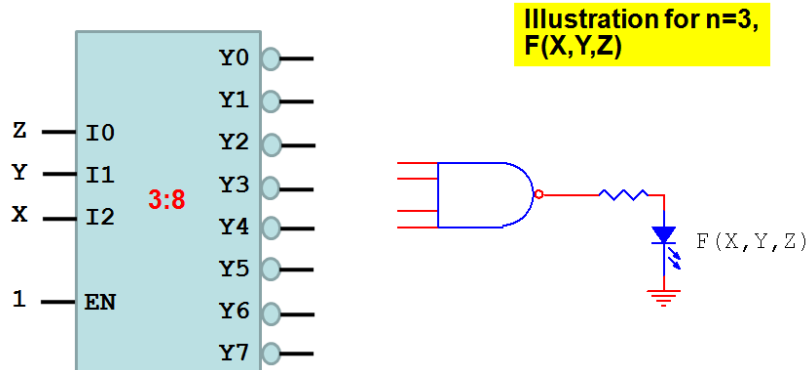


Note that **EN** can also be construed as a **digital input** that is routed to the **selected output**, in which case the circuit would be referred to as a **demultiplexer**

- key observations

- each output of an n to 2<sup>n</sup> binary decoder represents a minterm of an n-variable Boolean function; therefore, any arbitrary Boolean function of n-variables can be realized with an n-input binary decoder by simply “OR-ing” the needed outputs
- if the decoder outputs are active low, a NAND gate can be used to “OR” the minterms of the function (representing its ON set)
- if the decoder outputs are active low, an AND gate can be used to “OR” the minterms of the complement function (representing its OFF set)
- a NAND gate (or AND gate) with at most 2<sup>n-1</sup> inputs is needed to implement an arbitrary n-variable function using an n to 2<sup>n</sup> binary decoder (that has active low outputs)

- general circuit for implementing an arbitrary n-variable function using a decoder, for case where **ON set has ≤ 2<sup>n-1</sup>** members



• decoders/demultiplexers in Verilog

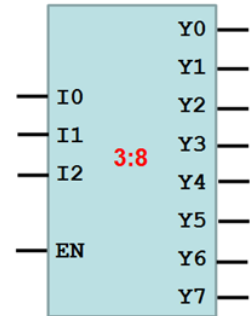
```
/* 3:8 Decoder / 1:8 Demultiplexer with Active-High Outputs */
```

```
module dec38H(EN, I, Y);

    input wire EN;           // Enable input pin
    input wire [2:0] I;     // Select input pins
    output reg [7:0] Y;     // Active-high output pins

    always @* begin // @* instead of listing inputs
        Y = 8'b0;     // assign all bits of Y to 0
        Y[I]= EN;    // overwrite the Ith bit with EN
    end

endmodule
```



1. The OFF set realized by this decoder-based circuit is:

- A.  $\Pi_{X,Y,Z}(0,2,5,7)$
- B.  $\Pi_{X,Y,Z}(1,3,4,6)$
- C.  $\Pi_{X,Y,Z}(1,2,4,5)$
- D.  $\Pi_{X,Y,Z}(0,3,4,6)$
- E. none of the above

2. The ON set realized by this decoder-based circuit is:

- A.  $\Sigma_{X,Y,Z}(0,2,5,7)$
- B.  $\Sigma_{X,Y,Z}(1,3,4,6)$
- C.  $\Sigma_{X,Y,Z}(1,2,4,5)$
- D.  $\Sigma_{X,Y,Z}(0,3,4,6)$
- E. none of the above

• special purpose decoders (e.g., 7-segment display)

```
/* Hexadecimal 7-Segment Decoder for 22V10 */

module hexadec(I, A, B, C, D, E, F, G);

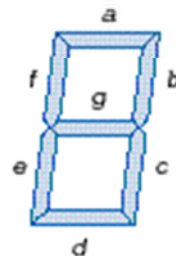
    input wire [3:0] I /*synthesis loc="2,3,4,5"*/;
    output wire A, B, C, D, E, F, G;

    reg [6:0] SEG7;

    always @ (I) begin
        case (I)
            4'b0000: SEG7 = 7'b1111110;
            4'b0001: SEG7 = 7'b0110000;
            4'b0010: SEG7 = 7'b1101101;
            4'b0011: SEG7 = 7'b1111001;
            4'b0100: SEG7 = 7'b0110011;
            4'b0101: SEG7 = 7'b1011011;
            4'b0110: SEG7 = 7'b1011111;
            4'b0111: SEG7 = 7'b1110000;
            4'b1000: SEG7 = 7'b1111111;
            4'b1001: SEG7 = 7'b1111011;
            4'b1010: SEG7 = 7'b1110111;
            4'b1011: SEG7 = 7'b0011111;
            4'b1100: SEG7 = 7'b1001110;
            4'b1101: SEG7 = 7'b0111101;
            4'b1110: SEG7 = 7'b1001111;
            4'b1111: SEG7 = 7'b1000111;
        endcase
    end

    assign {A,B,C,D,E,F,G} = SEG7;

endmodule
```



## Lecture Summary – Module 2-H

### Combinational Building Blocks: Encoders and Tri-State Outputs

**Reference:** *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 408-415, 430-432; 5<sup>th</sup> Ed. 279-280, 308-310

- overview
  - an encoder is an “inverse decoder” – the role of inputs and outputs is reversed, and there are more input code bits than output code bits
  - common application: encode “device number” associated with service request
  - problem: more than one device may be requesting service at a given instant – motivation for “priority encoder”
- priority encoders
  - inputs are numbered, priority is assigned based on number (usually lowest number → lowest priority, etc., but *not always*)
  - easiest way to do this in Verilog is using casez construct
  - example – 8:3 priority encoder with “strobe output” to indicate if any of the encoder inputs have been asserted
- Verilog casez construct
  - use ? as wild card
  - beware of non-unique expressions: 1<sup>st</sup> matching expression wins

```

module pri_enc(I, E, G);

  input wire [7:0] I; // Input 0 - lowest priority, Input 7 - highest priority
  output wire [2:0] E; // Encoded output
  output wire G; // Strobe output (asserted if any input is asserted)

  reg [3:0] EG;

  always @ (I) begin
    casez (I)
      8'b00000000: EG = 4'b0000; // No inputs asserted
      8'b00000001: EG = 4'b0001; // Input 0 wins
      8'b0000001?: EG = 4'b0011; // Input 1 wins
      8'b000001??: EG = 4'b0101; // Input 2 wins
      8'b00001???: EG = 4'b0111; // Input 3 wins
      8'b0001????: EG = 4'b1001; // Input 4 wins
      8'b001?????: EG = 4'b1011; // Input 5 wins
      8'b01??????: EG = 4'b1101; // Input 6 wins
      8'b1??????: EG = 4'b1111; // Input 7 wins
    endcase
  end

  assign {E,G} = EG;

endmodule

```





1. The **highest priority input** is:

- A. A
- B. B
- C. C
- D. D
- E. none of the above

2. The **lowest priority input** is:

- A. A
- B. B
- C. C
- D. D
- E. none of the above

3. If input **A is asserted**,  
the outputs will be:

- A. E1=0, E0=0, G=0
- B. E1=0, E0=0, G=1
- C. E1=1, E0=1, G=0
- D. E1=1, E0=1, G=1
- E. none of the above

4. When inputs **B and C are asserted simultaneously**  
(and A is negated) the outputs will be:

- A. E1=0, E0=0, G=1
- B. E1=0, E0=1, G=1
- C. E1=1, E0=0, G=1
- D. E1=1, E0=1, G=1
- E. none of the above

```

/* Different Priority Encoder */
module diff_pri(A,B,C,D,E,G);

    input wire A, B, C, D;
    output wire [1:0] E;
    output wire G;

    reg [2:0] EG;

    assign E = EG[2:1];
    assign G = EG[0];

    always @ (A, B, C, D) begin
        casez ({A,B,C,D})
            4'b0000: EG = 3'b000;
            4'b0001: EG = 3'b111;
            4'b001?: EG = 3'b101;
            4'b01???: EG = 3'b011;
            4'b1????: EG = 3'b001;
        endcase
    end

endmodule

```

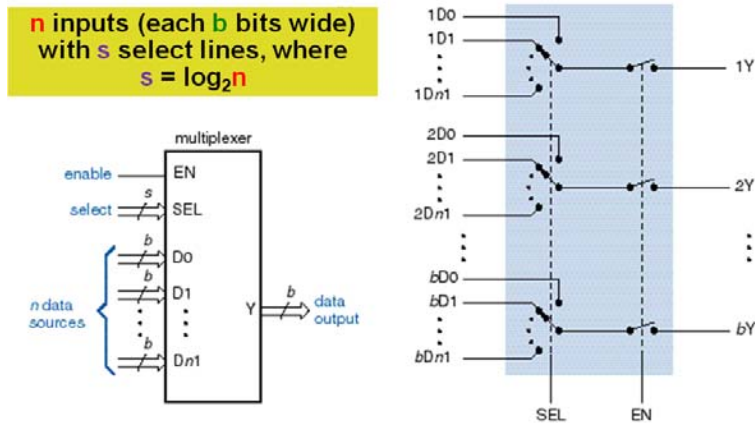
## Lecture Summary – Module 2-I

### Combinational Building Blocks: Multiplexers

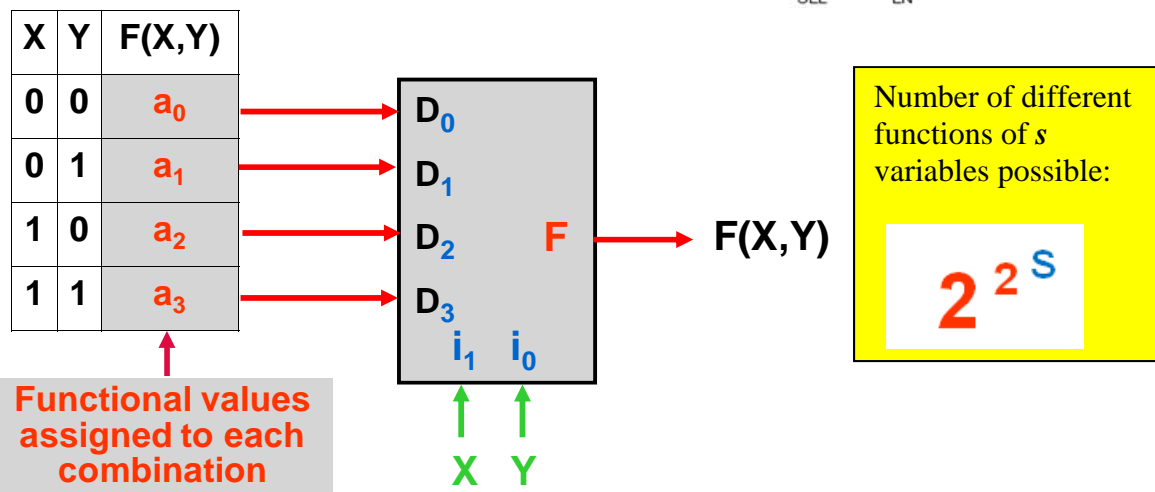
**Reference:** *Digital Design Principles and Practices* 4<sup>th</sup> Ed. pp. 432-440, 445-446; 5<sup>th</sup> Ed. pp. 281-289, 290-291

- overview

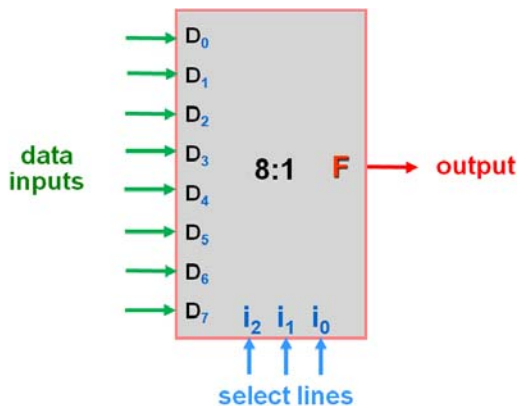
- o a **multiplexer** is a digital switch that uses  $s$  select lines to determine which of  $n = 2^s$  inputs is connected to its output
- o each of the input paths may be  $b$  bits wide
- o equation implemented by  $s$ -select line mux is the SoP form of a **general  $s$ -variable Boolean function**:  $F(X,Y) = a_0 \cdot X' \cdot Y' + a_1 \cdot X' \cdot Y + a_2 \cdot X \cdot Y' + a_3 \cdot X \cdot Y$
- o general structure



- truth table analogy



- example: 8-to-1 (8:1) multiplexer

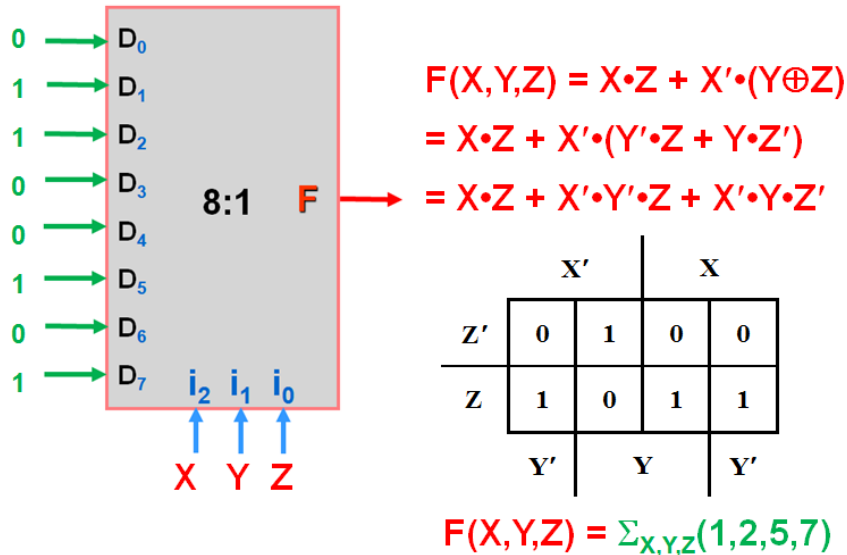


```

module mux811(D, EN, S, Y);
  input wire [7:0] D;      // Data inputs
  input wire EN;         // Function enable
  input wire [2:0] S;    // Select lines
  output wire Y;        // Output
  assign Y = EN & (!S[2] & !S[1] & !S[0] & D[0] |
                 !S[2] & !S[1] & S[0] & D[1] |
                 !S[2] & S[1] & !S[0] & D[2] |
                 !S[2] & S[1] & S[0] & D[3] |
                 S[2] & !S[1] & !S[0] & D[4] |
                 S[2] & !S[1] & S[0] & D[5] |
                 S[2] & S[1] & !S[0] & D[6] |
                 S[2] & S[1] & S[0] & D[7] );
endmodule
    
```

- example – multiplexer function realization

Determine the multiplexer data input values for realizing the function  $F(X,Y,Z) = X \cdot Z + X' \cdot (Y \oplus Z)$



- multiplexers in Verilog (8-bit wide 4:1 mux example)

```

module mux418a(EN, S, A, B, C, D, Y_z);
  input wire EN; // Tri-state output enable line
  input wire [1:0] S; // Select inputs
  input wire [7:0] A, B, C, D; // 8-bit input buses
  output tri [7:0] Y_z; // 8-bit output bus
  wire [7:0] Y;
  assign Y = S[1] & !S[0] & A |
            !S[1] & S[0] & B |
            S[1] & !S[0] & C |
            S[1] & S[0] & D;
  assign Y_z = EN ? Y : 8'bZZZZZZZZ;
endmodule

```

```

module mux418b(EN, S, A, B, C, D, Y_z);
  input wire EN; // Tri-state output enable line
  input wire [1:0] S; // Select inputs
  input wire [7:0] A, B, C, D; // 8-bit input buses
  output tri [7:0] Y_z; // 8-bit output bus
  reg [7:0] Y;
  assign Y_z = EN ? Y : 8'bZZZZZZZZ;
  always @ (S) begin
    // Y = 8'b00000000;
    if (S == 2'b00) Y = A;
    else if (S == 2'b01) Y = B;
    else if (S == 2'b10) Y = C;
    else if (S == 2'b11) Y = D;
    // else Y = 8'b00000000;
  end
endmodule

```

1. The number of equations generated by this program (that would be burned into a PLD that realized this design) is:
- A. 2
  - B. 8
  - C. 9
  - D. 16
  - E. none of the above

```

/* Big Multiplexer */
module bigmux(EN, S, A, B, C, D, Y_z);
  input wire EN;
  input wire [1:0] S;
  input wire [7:0] A, B, C, D;
  output tri [7:0] Y_z;
  wire [7:0] Y;
  assign Y_z = EN ? Y : 8'bZZZZZZZZ;
  assign Y = ~S[1] & ~S[0] & A |
             ~S[1] & S[0] & B |
             S[1] & ~S[0] & C |
             S[1] & S[0] & D;
endmodule

```

2. When  $EN=0$ ,  $S[1]=1$ , and  $S[0]=1$ , the output  $Y_z$ :
- A. will all be Hi-Z
  - B. will all be zero
  - C. will all be one
  - D. will be equal to the inputs D
  - E. none of the above

3. When  $EN=1$ ,  $S[1]=1$ , and  $S[0]=1$ , the output  $Y$
- A. will all be Hi-Z
  - B. will all be zero
  - C. will all be one
  - D. will be equal to the input D
  - E. none of the above

## Lecture Summary – Module 2-J

### Top Level (Hierarchical) Models

Reference: *Digital Design Principles and Practices* (4<sup>th</sup> Ed.), pp. 306-308

- **definition:** A *top level module* is the highest level module in a design hierarchy that instantiates other modules and connects them
- separating logic across multiple modules serves the advantage of reusability for modules and removing redundant logic
- example: If two modules use a 4-to-1 mux, create a separate module for the mux, and simply *instantiate* it in the other modules
- follows structural style of instantiation: `module_name instance_name (signal_list);`
- signals in `signal_list` will be connected in the order of that module's portlist – this is called *port mapping by order*
- alternatively, *port mapping by name* can be used, which is a more error-free method – here, each signal passed to the instantiated module uses the name of the signal in the module's port list to indicate where it is connected

```
module and_or(A,B,C,D);
  input wire A, B;
  output wire C, D;

  assign C = A & B;
  assign D = A | B;
endmodule
```

```
module top_order(w,x,y,z);
  input wire w, x;
  output wire y, z;

  assign a = 1'b0;
  assign b = 1'b1;
  and_or DUT1(w, x, y, z);

endmodule
```

Port mapping by **order** assigns **A = w, B = x, C = y, D = z** based on how they are ordered in the instantiation

```
module top_name(w,x,y,z);
  input wire w, x;
  output wire y, z;

  assign a = 1'b0;
  assign b = 1'b1;
  and_or DUT1(.B(x), .A(w), .D(z), .C(y));

endmodule
```

Port mapping by **name** allows the signals to be listed in any order with **A = w, B = x, C = y, D = z**