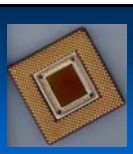


Purdue IM:PACT* Spring 2019 Edition

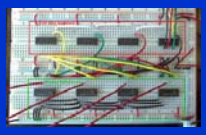
*Instruction Matters: Purdue Academic Course Transformation

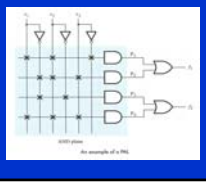
Introduction to Digital System Design

Module 2
Combinational Logic Circuits

Glossary of Common Terms

- **DISCRETE LOGIC** – a circuit constructed using small-scale integrated (SSI) and medium-scale integrated (MSI) logic devices (NAND gates, decoders, multiplexers, etc.)
- **PROGRAMMABLE LOGIC DEVICE (PLD)** – an integrated circuit onto which a generic logic circuit can be programmed (and subsequently erased and re-programmed)
- **GENERIC ARRAY LOGIC (GAL)** – a (legacy) flash memory based PLD, which is typically erased and re-programmed out-of-circuit
- **COMPLEX PLD (CPLD)** – large flash memory based PLD that is programmable in-circuit





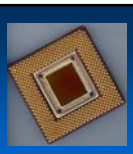
Glossary of Common Terms

- **isp (IN-SYSTEM PROGRAMMING)** – prefix used on CPLDs that can be erased and re-programmed in-circuit
- **FIELD PROGRAMMABLE GATE ARRAY (FPGA)** – an SRAM-based PLD that can be programmed in-circuit (no need to "erase" since SRAM-based)
- **ADVANCED BOOLEAN EXPRESSION LANGUAGE (ABEL)** – a "classic" hardware description language (HDL) for specifying the behavior of PLDs
- **VHDL** and **VERILOG** – advanced hardware simulation and description languages

```
module dff_d (q, d, clk);
    output q;
    input d, clk;
    reg q;

    always @(posedge(clk)) q=d;
endmodule
```

Module 2

- **Learning Outcome: "An ability to analyze and design combinational logic circuits"**
  - A. Combinational Circuit Analysis and Synthesis
  - B. Mapping and Minimization
  - C. Timing Hazards
  - D. XOR/XNOR Functions
  - E. Programmable Logic Devices
  - F. Hardware Description Languages
  - G. Combinational Building Blocks: Decoders
  - H. Combinational Building Blocks: Encoders and Tri-State Outputs
  - I. Combinational Building Blocks: Multiplexers
  - J. Top Level Modules



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-A
Combinational Circuit Analysis and Synthesis

Reading Assignment:
DDPP 4th Ed. pp. 196-210, 5th Ed, pp. 100-117

Learning Objectives:

- Identify minterms (product terms) and maxterms (sum terms)
- 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
- 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
- Transform a logic circuit from one set of symbols to another through graphical application of DeMorgan's Law
- Realize a combinational function directly using basic gates (NOT, AND, OR, NAND, NOR)

Outline

- Overview
- Definitions
- Minterm identification
- Maxterm identification
- ON Sets and OFF sets
- Combinational circuit analysis
- Equivalent symbols
- Combinational circuit synthesis

7

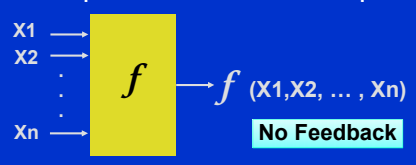
Overview

- We *analyze* a combinational logic circuit by obtaining a *formal description* of its logic function
- Once we have a description of the logic function, we can:
  - determine the behavior of the circuit for various input combinations
  - manipulate an algebraic description to suggest different circuit structures
  - transform an algebraic description into a standard form (e.g., sum-of-products for PLD implementation)
  - use an algebraic description of the circuit's functional behavior in the analysis of a larger system that includes the circuit

8

Definitions

- Definition: A *combinational logic circuit* is one whose output depend only on its *current combination of input values (or "input combination")*
- Definition: A *logic function* is the *assignment* of "0" or "1" to each possible combination of its input variables



9

Definitions

- Definition: A *literal* is a variable or the complement of a variable
- Definition: A *product term* is a single literal or a logical product of two or more literals
- Definition: A *sum-of-products expression* is a logical sum of product terms
- Definition: A *sum term* is a single literal or a logical sum of two or more literals
- Definition: A *product-of-sums expression* is a logical product of sum terms

10

Examples

W, X, Y′	*Literals*
W • X • Z	*Product Term*
X • Y′ + W • Z	*Sum of Products Expression*
X + Y + Z′	*Sum Term*
(X + Y) • (W + Z′)	*Product of Sums Expression*

11

Definitions

- Definition: A *normal term* is a product or sum term in which no variable appears more than once
- Definition: An n-variable *minterm* is a normal product term with n literals
- Definition: An n-variable *maxterm* is a normal sum term with n literals
- Definition: The *canonical* sum of a logic function is a sum of minterms corresponding to input combinations for which the function produces a "1" output
- Definition: The *canonical* product of a logic function is a product of maxterms corresponding to input combinations for which the function produces a "0" output

12

Minterm Identification

0 → complemented
1 → true

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)	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

13

Maxterm Identification

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)	$X \cdot Y \cdot Z$	$X' + Y' + Z'$

14

ON Sets and OFF Sets

- Definition: The minterm list that "turns on" an output function is called the *on set*
- Example: $\Sigma_{X,Y,Z}(0,1,2,3)$

  Indicates "sum" (of products)

  Rows of truth table that are "1"

- Definition: The maxterm list that "turns off" an output function is called the *off set*
- Example: $\Pi_{X,Y,Z}(4,5,6,7)$

  Indicates "product" (of sums)

  Rows of truth table that are "0"

15

Example

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

Based on the truth table, determine the following

F(X,Y,Z) expressed as:

an *on-set:* _____

an *off-set:* _____

a *sum of minterms:* _____

a *product of maxterms:* _____

16

Example

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

Based on the truth table, determine the following

F(X,Y,Z) expressed as:

an *on-set:* $\Sigma_{X,Y,Z}(0,3,6,7)$

an *off-set:* _____

a *sum of minterms:* _____

a *product of maxterms:* _____

17

Example

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

Based on the truth table, determine the following

F(X,Y,Z) expressed as:

an *on-set:* $\Sigma_{X,Y,Z}(0,3,6,7)$

an *off-set:* $\Pi_{X,Y,Z}(1,2,4,5)$

a *sum of minterms:* _____

a *product of maxterms:* _____

18

Example

Based on the truth table, determine the following

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

F(X,Y,Z) expressed as:

an *on-set:* $\Sigma_{X,Y,Z}(0,3,6,7)$

an *off-set:* $\Pi_{X,Y,Z}(1,2,4,5)$

a *sum of minterms:* X'•Y'•Z' + X'•Y•Z + X•Y•Z' + X•Y•Z

a *product of maxterms:* _____

Example

Based on the truth table, determine the following

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

F(X,Y,Z) expressed as:

an *on-set:* $\Sigma_{X,Y,Z}(0,3,6,7)$

an *off-set:* $\Pi_{X,Y,Z}(1,2,4,5)$

a *sum of minterms:* X'•Y'•Z' + X'•Y•Z + X•Y•Z' + X•Y•Z

a *product of maxterms:* _____

(X+Y+Z')•(X+Y'+Z)• (X'+Y+Z)•(X'+Y+Z')

Clicker Quiz

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)
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

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

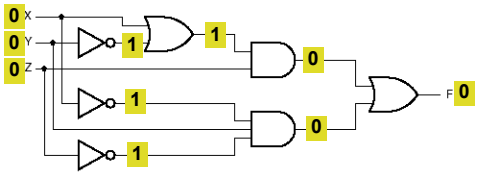
DUAL truth table rule: "flip and complement"

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

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

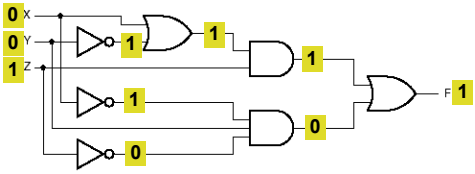
25

Example - Combinational Analysis



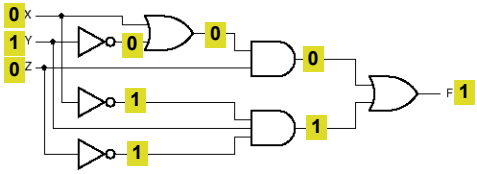
26

Example - Combinational Analysis



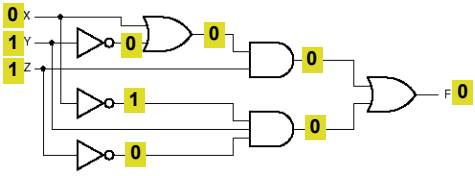
27

Example - Combinational Analysis



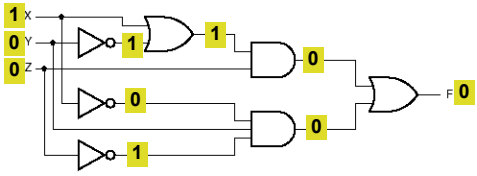
28

Example - Combinational Analysis



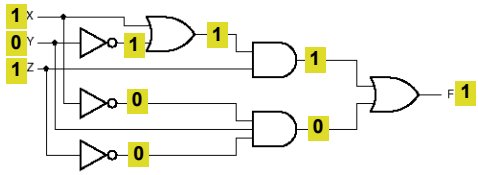
29

Example - Combinational Analysis



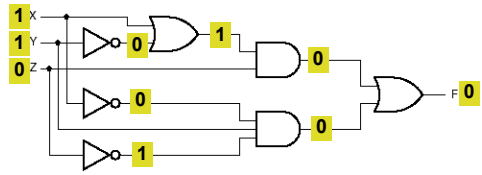
30

Example - Combinational Analysis



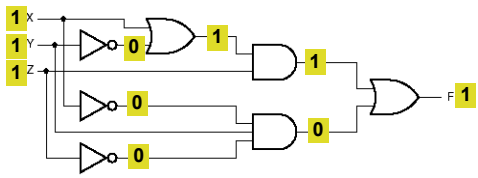
31

Example - Combinational Analysis



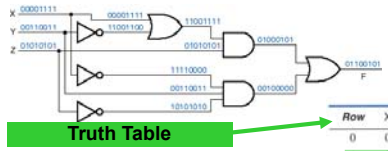
32

Example - Combinational Analysis



33

Example - Combinational Analysis



Truth Table

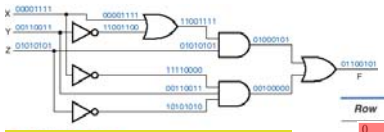
Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

The *"on set"* of this function is $f(X,Y,Z) = \Sigma_{X,Y,Z}(1,2,5,7)$

The *canonical sum* of this function is $f(X,Y,Z) = X'\bullet Y'\bullet Z + X'\bullet Y\bullet Z' + X\bullet Y'\bullet Z + X\bullet Y\bullet Z$

34

Example - Combinational Analysis



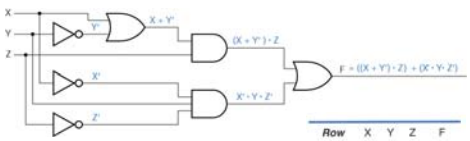
Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

The *"off set"* of this function is $f(X,Y,Z) = \Pi_{X,Y,Z}(0,3,4,6)$

The *canonical product* of this function is $f(X,Y,Z) = (X+Y+Z) \bullet (X+Y'+Z') \bullet (X'+Y+Z) \bullet (X'+Y'+Z)$

35

Example - Combinational Analysis

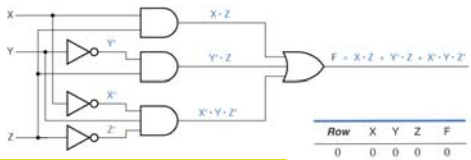


Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Writing the function implemented by this circuit "directly" yields $f(X,Y,Z) = ((X+Y')\bullet Z) + (X'\bullet Y\bullet Z') = X\bullet Z + Y'\bullet Z + X'\bullet Y\bullet Z'$

36

Example - Combinational Analysis

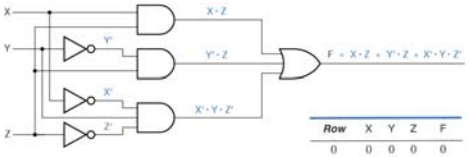


Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

The *expression*
f (X,Y,Z) = X•Z + Y′•Z + X′•Y•Z′
corresponds to a *different circuit* ("two-level AND-OR") for the *same logic function*

37

Example – Equivalent Symbols

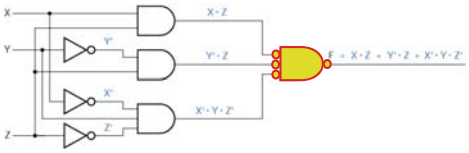


Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Recall that an *equivalent symbol* can be drawn for a gate by taking the *dual* of the operator and *complementing* all of its *inputs and outputs*

38

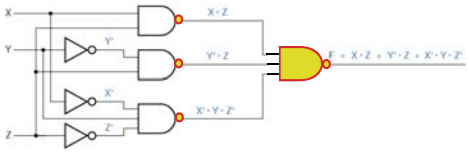
Example – Equivalent Symbols



Step 1: Starting at the "output end", replace the "OR" gate with an AND gate that has its inputs and outputs complemented

39

Example – Equivalent Symbols



Step 2: Migrate the "inversion bubbles", as appropriate, by applying involution

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

40

Summary

- There are numerous ways a combinational logic function can be represented
  - truth table
  - algebraic sum of minterms (sum-of-products expression)
  - minterm list (ON set)
  - algebraic product of maxterms (product-of-sums expression)
  - maxterm list (OFF set)

41

Clicker Quiz

42

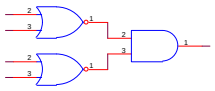
1. A **NOR** gate is **logically equivalent** to:
   - A. an AND gate with inverted inputs
   - B. an OR gate with inverted inputs
   - C. a NAND gate with inverted inputs
   - D. a NOR gate with inverted inputs
   - E. none of the above

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**
- **Such a circuit is called a *realization* of the function**
- **Typically there are *many possible realizations* of the *same function***
- **Circuit transformations can be made *algebraically* or *graphically***

46

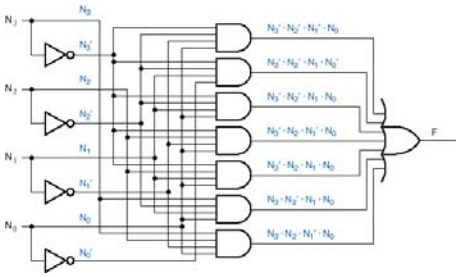
Combinational Synthesis

- **The starting point for designing a combinational logic circuit is usually a *word description* of a problem**
- **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*)**

$$f(N_3,N_2,N_1,N_0) = \Sigma N_3,N_2,N_1,N_0(1,2,3,5,7,11,13)$$

47

Example – Prime Number Detector

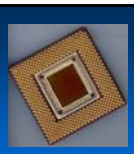


48

Thought Questions

- 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

?



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-B
Mapping and Minimization

Reading Assignment:
DDPP 4th Ed. pp. 210-222, 5th Ed. pp. 117-125

Learning Objectives:

- Draw a **Karnaugh Map** ("K-map") for a 2-, 3-, 4-, or 5-variable logic function
- List the assumptions underlying function minimization
- Identify the **prime implicants** ("PI"), essential PI, and non-essential PI of a function depicted on a K-map
- 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
- Use a **K-map to convert** a function from one standard form to another
- 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
- **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

Outline

- Overview
- Representation of logic functions using K-maps
- Minimization of logic functions using K-maps
- NAND-Wired AND configuration
- Incompletely specified functions
  - where they occur
  - how to minimize them

Overview

- Minimization is an important step in both ASIC *(application specific integrated circuit)* design and in PLD-based *(programmable logic device)* design
- Extra gates and gate inputs require *more chip area* ("real estate") and thereby *increase cost and power consumption*
- *Canonical* sum and product expressions (which can be determined "directly" from a truth table) are particularly expensive because the *number of minterms [maxterms] grows exponentially* with the number of variables

Overview

- 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 (T10 and T10′):

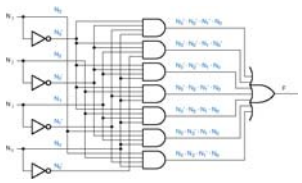
Expression • X + Expression • X′ = Expression

Takeaway: *The fundamental basis of logic minimization is the COMBINING THEOREM*

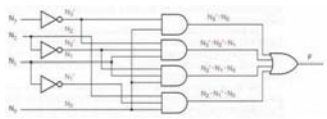
Minimization Motivation

4-bit Prime Number Detector

Minterm Form



Minimized Circuit Realization



55

Overview

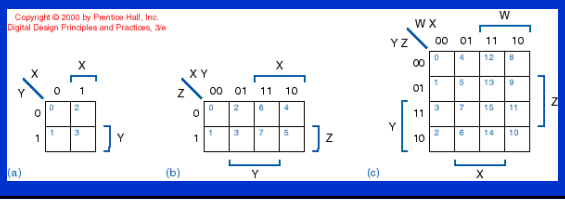
- Limitations of minimization methods
  - no restriction on *fan-in* is assumed (i.e., the total number of inputs a gate can have is assumed to be "infinite")
  - minimization of a function of more than 4 or 5 variables is *not practical* to do "by hand" (a computer program must be used!)
  - both *true and complemented* versions of all input variables are assumed to be readily available (i.e., the cost of input inverters is not considered)

This latter assumption is very appropriate for *PLD-based* design, but often violated in *gate-level* and *ASIC-based* design

56

Karnaugh Maps

- A Karnaugh map (or "K-map") is a graphical representation of a logic function's truth table
- The map for an n-variable logic function is an array with 2^n cells, one for each possible input combination (minterm)



57

Karnaugh Maps

- Several things to note concerning K-maps:
  - the small number in the corner of each square indicates the *minterm number*
  - the entries in the squares correspond to the "on set" of the function
  - the labels are placed in such a way that the minterms on any pair of adjacent squares *differ by only one literal*
  - the sides of the map are considered to be *contiguous*
  - *adjacent, like* squares may be combined in groups of 2^k to *reduce* the number of product terms in an expression (a grouping of 2^k squares will eliminate k variables)

58

Karnaugh Maps

- An alternate drawing for a 2-variable K-map

	X′	X
Y′	0	2
Y	1	3

59

Karnaugh Maps

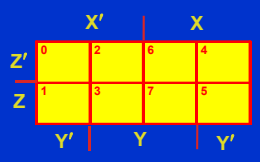
- Example: $f(X,Y) = X' + Y$

	X′	X
Y′	0: 1	2: 0
Y	1: 1	3: 1

60

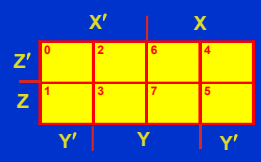
Karnaugh Maps

- An alternate drawing for a 3-variable K-map



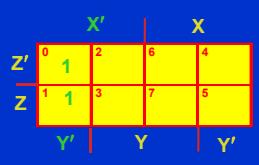
Karnaugh Maps

- Example: $f(X,Y,Z) = X' \bullet Y' + Y \bullet Z$



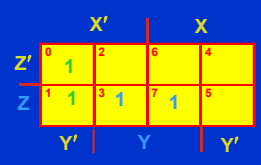
Karnaugh Maps

- Example: $f(X,Y,Z) = X' \bullet Y' + Y \bullet Z$



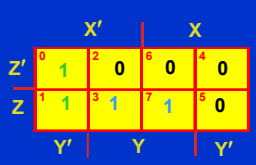
Karnaugh Maps

- Example: $f(X,Y,Z) = X' \bullet Y' + Y \bullet Z$



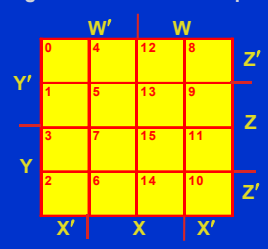
Karnaugh Maps

- Example: $f(X,Y,Z) = X' \bullet Y' + Y \bullet Z$



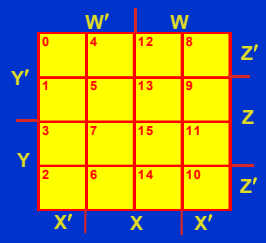
Karnaugh Maps

- Drawing for a 4-variable K-map



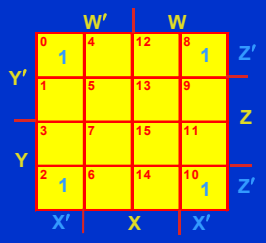
Karnaugh Maps

- Example: f(W,X,Y,Z) = X′•Z′ + W•Z + W′•X



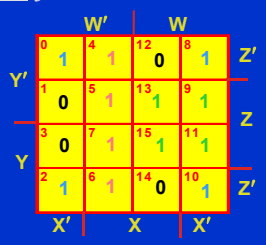
Karnaugh Maps

- Example: f(W,X,Y,Z) = X′•Z′ + W•Z + W′•X



Karnaugh Maps

- Example: f(W,X,Y,Z) = X′•Z′ + W•Z + W′•X



Minimization

- Definition: A *minimal sum* of a logic function f is a sum-of-products expression for f such that no sum-of-products expression for f has fewer product terms, and any sum-of-products expression with the same number of product terms has at least as many literals

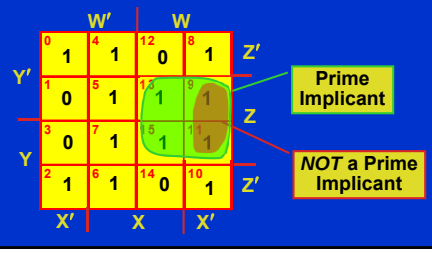
Translation: The *minimal sum* has the fewest possible product terms (first-level gates / second-level gate inputs) and the fewest possible literals (first-level gate inputs)

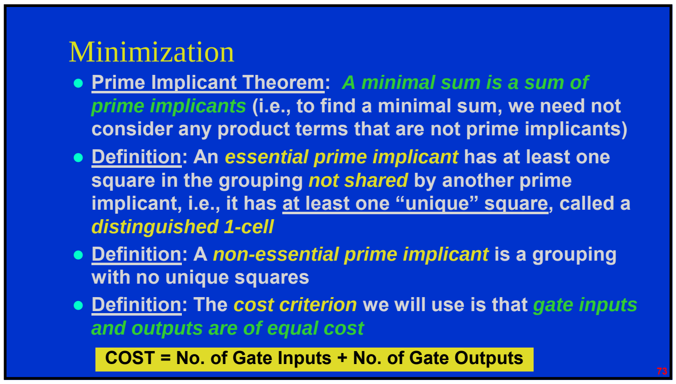
Minimization

- Definition: A logic function p *implies* a logic function f if for every input combination such that p = 1, then f = 1 also (i.e., if p implies f, then f is 1 for every input combination that p is 1, and maybe some more – or "f *covers* p")
- Definition: A *prime implicant* of an n-variable logic function f is a normal product term P that implies f, such that if any literal is removed from P, then the resulting product term does not imply f

Minimization

- Translation: A *prime implicant* is the *largest possible grouping* of size 2^k adjacent, like squares

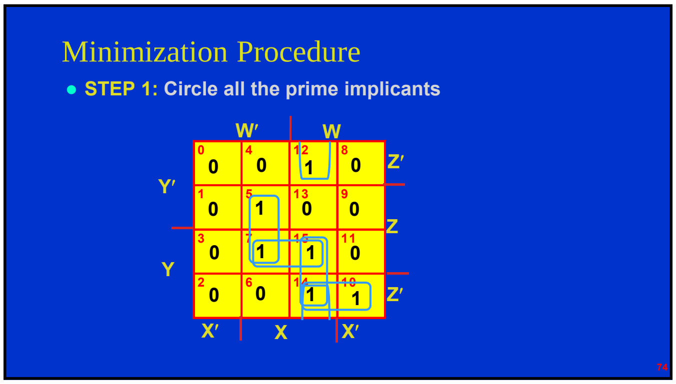




Minimization

- **Prime Implicant Theorem**: *A minimal sum is a sum of prime implicants* (i.e., to find a minimal sum, we need not consider any product terms that are not prime implicants)
- **Definition**: 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*
- **Definition**: A *non-essential prime implicant* is a grouping with no unique squares
- **Definition**: The *cost criterion* we will use is that *gate inputs and outputs are of equal cost*

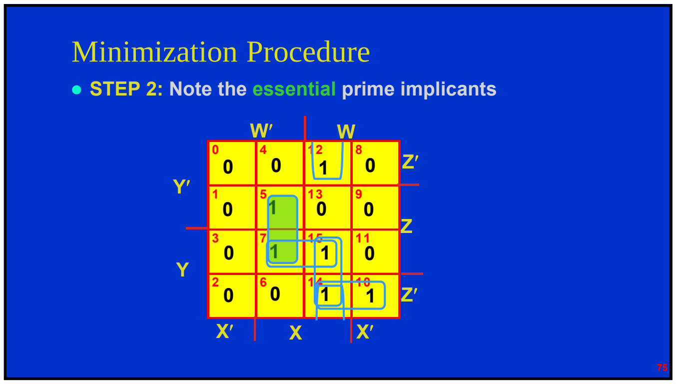
COST = No. of Gate Inputs + No. of Gate Outputs



Minimization Procedure

- **STEP 1:** Circle all the prime implicants

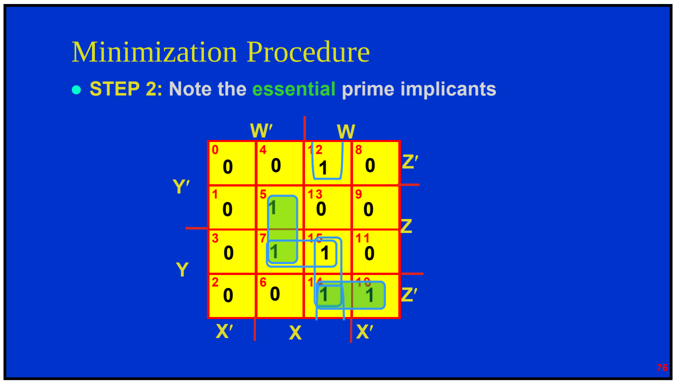
	W′	W′	W	W	
Y′	0	0	1	0	Z′
Y′	0	1	0	0	Z
Y	0	1	1	0	Z
Y	0	0	1	1	Z′
	X′	X	X	X′	



Minimization Procedure

- **STEP 2:** Note the essential prime implicants

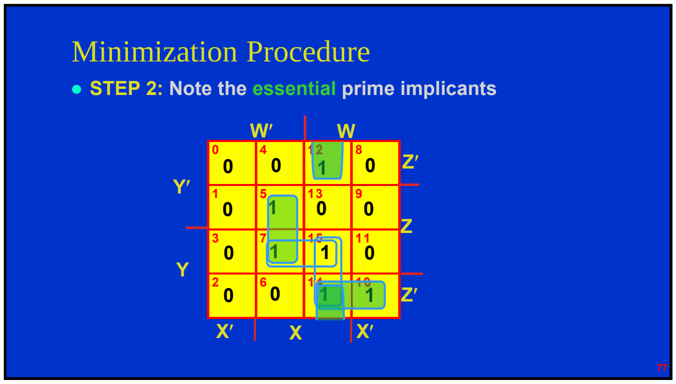
	W′	W′	W	W	
Y′	0	0	1	0	Z′
Y′	0	1	0	0	Z
Y	0	1	1	0	Z
Y	0	0	1	1	Z′
	X′	X	X	X′	



Minimization Procedure

- **STEP 2:** Note the essential prime implicants

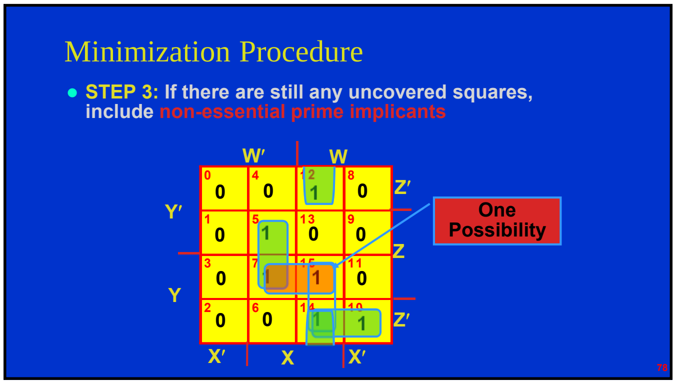
	W′	W′	W	W	
Y′	0	0	1	0	Z′
Y′	0	1	0	0	Z
Y	0	1	1	0	Z
Y	0	0	1	1	Z′
	X′	X	X	X′	



Minimization Procedure

- **STEP 2:** Note the essential prime implicants

	W′	W′	W	W	
Y′	0	0	1	0	Z′
Y′	0	1	0	0	Z
Y	0	1	1	0	Z
Y	0	0	1	1	Z′
	X′	X	X	X′	



Minimization Procedure

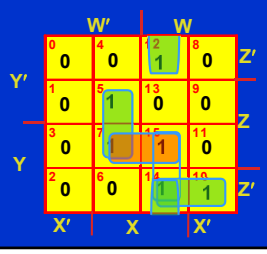
- **STEP 3:** If there are still any uncovered squares, include non-essential prime implicants

	W′	W′	W	W	
Y′	0	0	1	0	Z′
Y′	0	1	0	0	Z
Y	0	1	1	0	Z
Y	0	0	1	1	Z′
	X′	X	X	X′	

One Possibility

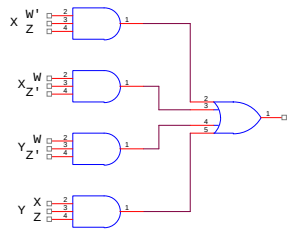
Minimization Procedure

- **STEP 3:** If there are still any uncovered squares, include non-essential prime implicants



f (W,X,Y,Z) = W′•X•Z + W•X•Z′ + W•Y•Z′ + X•Y•Z

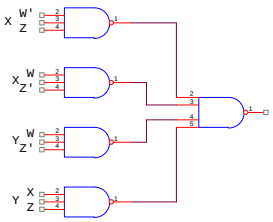
79



One possible circuit implementation (AND-OR):

COST is 16 inputs + 5 outputs = 21

80



EQUIVALENT circuit implementation, obtained through graphical application of DeMorgan's Law

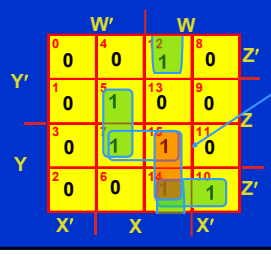
Note: AND-OR ≡ NAND-NAND

COST is 16 inputs + 5 outputs = 21 (same)

81

Minimization Procedure

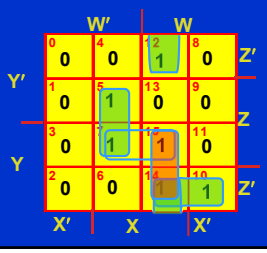
- REVISIT **STEP 3:** If there are still any uncovered squares, include non-essential prime implicants



82

Minimization Procedure

- REVISIT **STEP 3:** If there are still any uncovered squares, include non-essential prime implicants



f (W,X,Y,Z) = W′•X•Z + W•X•Z′ + W•Y•Z′ + W•X•Y

83

Clicker Quiz

84

	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

85

	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′	

2. The number of *essential prime implicants* is:
   - A. 1
   - B. 2
   - C. 3
   - D. 4
   - E. 5

86

	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′	

3. The number of *non-essential prime implicants* is:
   - A. 1
   - B. 2
   - C. 3
   - D. 4
   - E. 5

87

	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′	

4. The number of *product terms* in the *minimal sum* is:
   - A. 1
   - B. 2
   - C. 3
   - D. 4
   - E. 5

88

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

5. The ON SET for this function:
   - A. $\Sigma_{W,X,Y,Z}(2,4,5,6,9,10,11,12)$
   - B. $\Sigma_{W,X,Y,Z}(3,4,5,7,9,13,14,15)$
   - C. $\Sigma_{W,X,Y,Z}(3,4,5,7,9,10,11,13)$
   - D. $\Sigma_{W,X,Y,Z}(2,4,5,6,9,13,14,15)$
   - E. none of the above

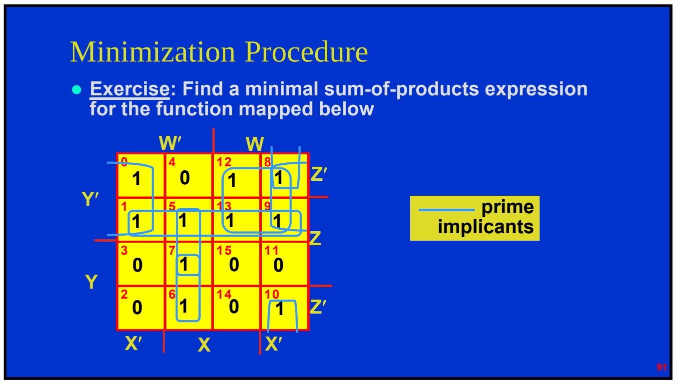
89

Minimization: Another Example

- **Exercise: Find a minimal sum-of-products expression for the function mapped below**

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

90

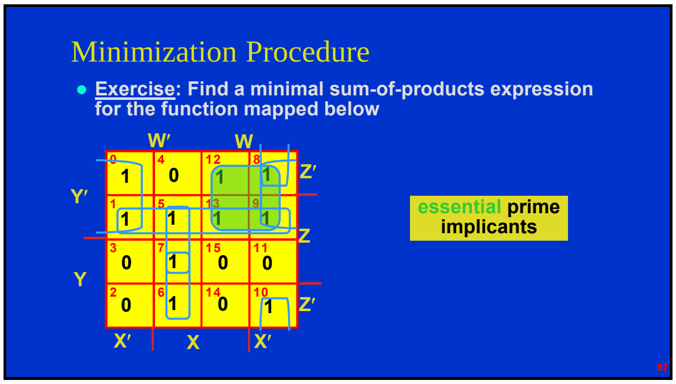


Minimization Procedure

- **Exercise**: Find a minimal sum-of-products expression for the function mapped below

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

prime implicants

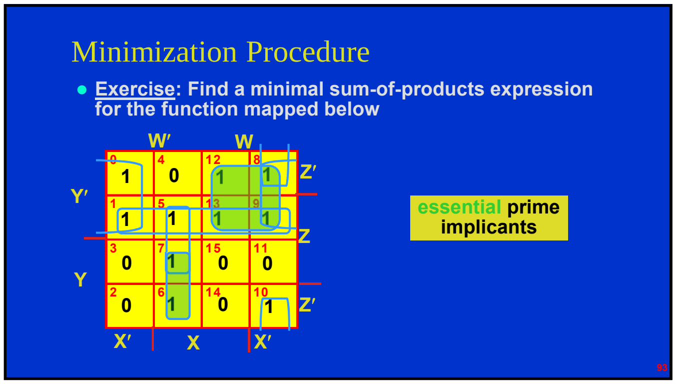


Minimization Procedure

- **Exercise**: Find a minimal sum-of-products expression for the function mapped below

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

essential prime implicants

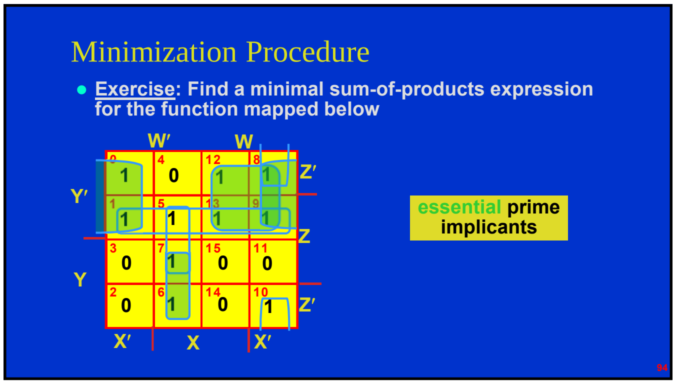


Minimization Procedure

- **Exercise**: Find a minimal sum-of-products expression for the function mapped below

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

essential prime implicants

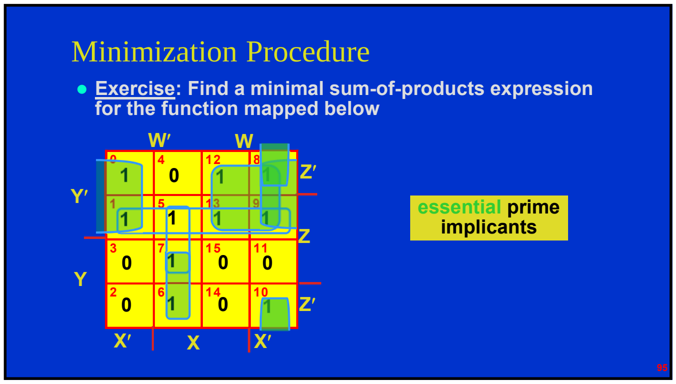


Minimization Procedure

- **Exercise**: Find a minimal sum-of-products expression for the function mapped below

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

essential prime implicants

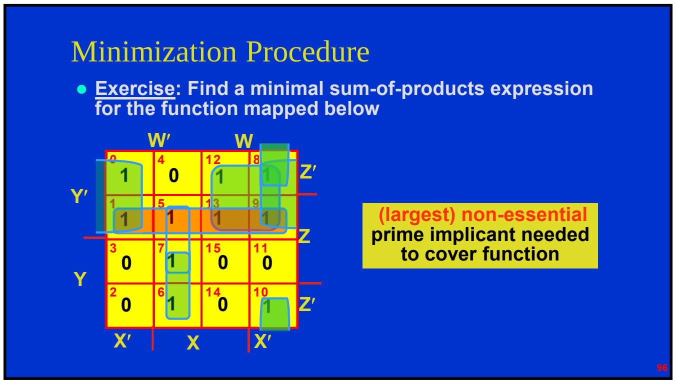


Minimization Procedure

- **Exercise**: Find a minimal sum-of-products expression for the function mapped below

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

essential prime implicants

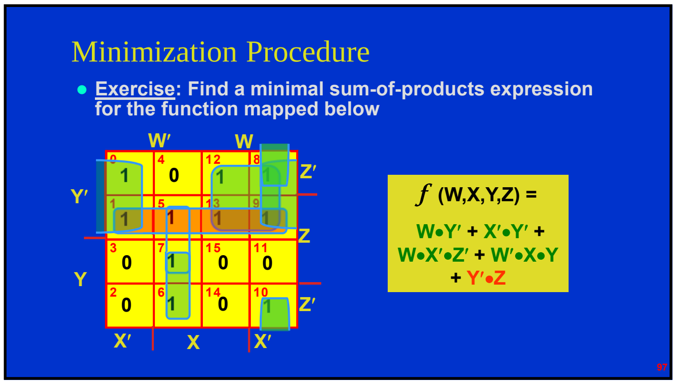


Minimization Procedure

- **Exercise**: Find a minimal sum-of-products expression for the function mapped below

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

(largest) non-essential prime implicant needed to cover function

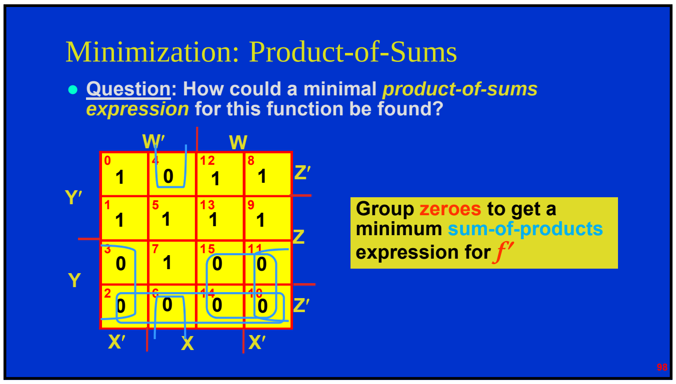


Minimization Procedure

- **Exercise**: Find a minimal sum-of-products expression for the function mapped below

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

f (W,X,Y,Z) =
W•Y′ + X′•Y′ +
W•X′•Z′ + W′•X•Y
+ Y′•Z

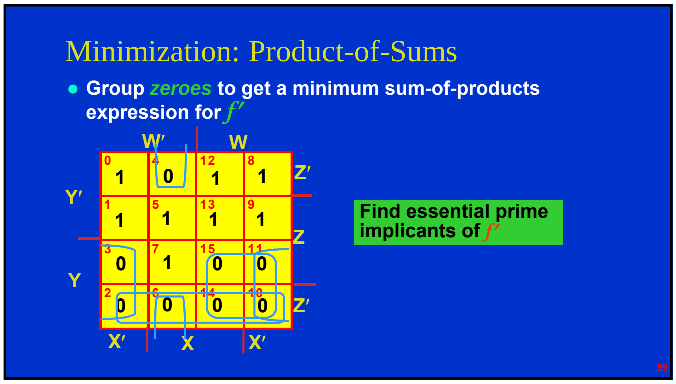


Minimization: Product-of-Sums

- **Question**: How could a minimal *product-of-sums expression* for this function be found?

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

Group zeroes to get a minimum sum-of-products expression for *f′*

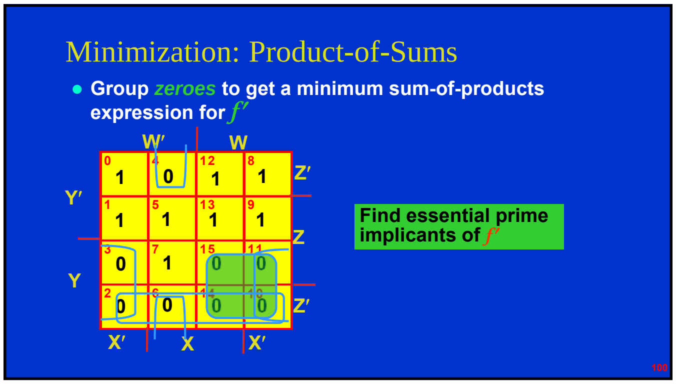


Minimization: Product-of-Sums

- Group *zeroes* to get a minimum sum-of-products expression for *f′*

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

Find essential prime implicants of *f′*

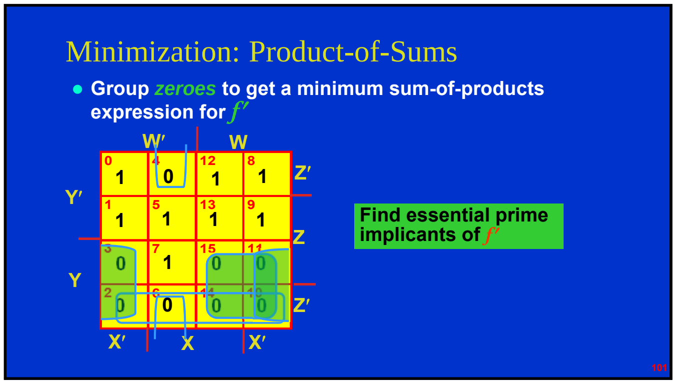


Minimization: Product-of-Sums

- Group *zeroes* to get a minimum sum-of-products expression for *f′*

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

Find essential prime implicants of *f′*

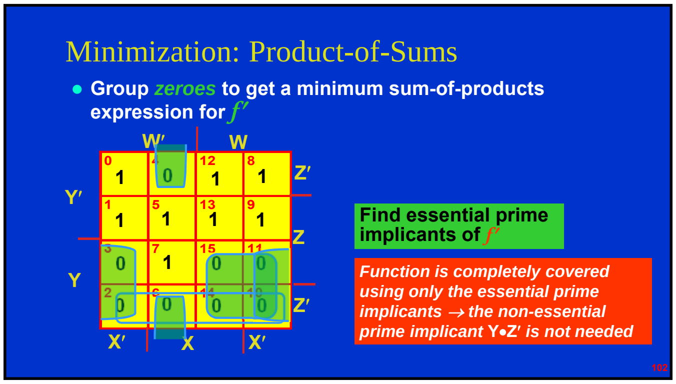


Minimization: Product-of-Sums

- Group *zeroes* to get a minimum sum-of-products expression for *f′*

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

Find essential prime implicants of *f′*



Minimization: Product-of-Sums

- Group *zeroes* to get a minimum sum-of-products expression for *f′*

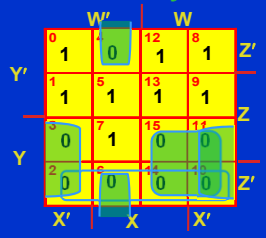
	W′ X′	W′ X	W X	W X′	
Y′	0: 1	4: 0	12: 1	8: 1	Z′
Y′	1: 1	5: 1	13: 1	9: 1	Z
Y	3: 0	7: 1	15: 0	11: 0	Z
Y	2: 0	6: 0	14: 0	10: 0	Z′

Find essential prime implicants of *f′*

Function is completely covered using only the essential prime implicants → the non-essential prime implicant Y•Z′ is not needed

Minimization: Product-of-Sums

- Group *zeroes* to get a minimum sum-of-products expression for *f′*

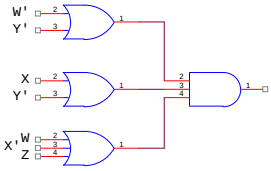


f′ = W•Y + X′•Y + W′•X•Z′

Apply DeMorgan's Law

f = (W′+Y′)•(X+Y′)•(W+X′+Z)

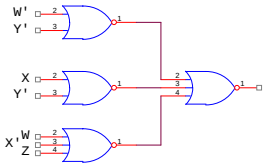
103



One possible circuit implementation (OR-AND):

COST is 10 inputs + 4 outputs = 14

104



EQUIVALENT circuit implementation, obtained through *graphical application* of DeMorgan's Law

Note: OR-AND ≡ NOR-NOR

COST is 10 inputs + 4 outputs = 14 (same)

105

More Minimization Examples

Assuming that only *true* variables are available, realize the function represented by $\Sigma_{X,Y,Z}(0,2,3,6)$ two different ways:

(a) using a single 7400 (quad 2-input NAND) plus a single 7410 (triple 3-input NAND)
(b) using a single 7403 (quad 2-input open-drain NAND)

Key to Solution: The "NAND-Wired AND" configuration realizes the *complement* of the NAND-NAND configuration → implement F′

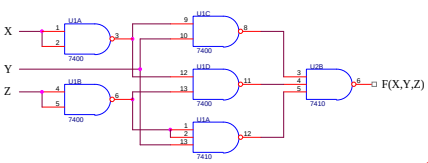
106

Solution to (a)

Given: $\Sigma_{X,Y,Z}(0,2,3,6)$

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

F(X,Y,Z) = X′•Y + X′•Z′ + Y•Z′



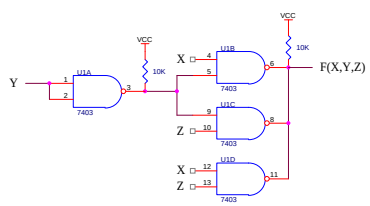
107

Solution to (b)

Given: $\Sigma_{X,Y,Z}(0,2,3,6)$

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

F′(X,Y,Z) = X•Y′ + X•Z + Y′•Z



108

"Conversion" Example

Express the *complement* of the following function in *minimal product-of-sums* form:

F(X,Y,Z) = (X + Y) · (X′ + Y + Z) · (X + Y + Z′)

➔ F′(X,Y,Z) = _____ ➔ Map _____

F(X,Y,Z) = _____ ➔
F′(X,Y,Z) in minimal POS form
= _____

109

"Conversion" Example

Express the *complement* of the following function in *minimal product-of-sums* form:

F(X,Y,Z) = (X + Y) · (X′ + Y + Z) · (X + Y + Z′)

➔ F′(X,Y,Z) = X′·Y′ + X·Y′·Z′ + X′·Y′·Z ➔ Map zeroes

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

F(X,Y,Z) = _____ ➔
F′(X,Y,Z) in minimal POS form
= _____

110

"Conversion" Example

Express the *complement* of the following function in *minimal product-of-sums* form:

F(X,Y,Z) = (X + Y) · (X′ + Y + Z) · (X + Y + Z′)

➔ F′(X,Y,Z) = X′·Y′ + X·Y′·Z′ + X′·Y′·Z ➔ Map zeroes

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

F(X,Y,Z) = Y + X•Z ➔
F′(X,Y,Z) in minimal POS form
= Y′ • (X′ + Z′)

111

Incompletely Specified Functions

- There are some logic functions that do not assign a specific binary output value (0/1) to each of the 2^n input combinations
- Since there are essentially some *unused combinations*, these functions are referred to as *incompletely specified functions*
- The unused combinations are often called *don't cares* or the *d-set*
- Example: Binary Coded Decimal (BCD), where 4 binary digits are used to represent a decimal digit $(0 - 9)_{10}$ – here there are 6 *unused combinations* $(1010 - 1111)_2$

112

Incompletely Specified Functions

- Application: Determine a logic function that will be "1" if the BCD digit input satisfies the following inequality:
  $1 < N_{10} < 9$

$F = \Sigma_{W,X,Y,Z}(2,3,4,5,6,7,8) + d(10,11,12,13,14,15)$

On Set

d-Set

113

BCD Inequality Detector Example

N_{10}	W X Y Z	F(W,X,Y,Z)
0	0 0 0 0	0
1	0 0 0 1	0
2	0 0 1 0	1
3	0 0 1 1	1
4	0 1 0 0	1
5	0 1 0 1	1
6	0 1 1 0	1
7	0 1 1 1	1
8	1 0 0 0	1
9	1 0 0 1	0

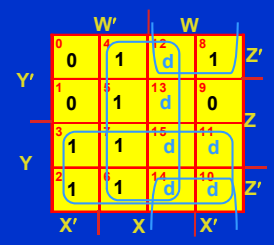
114

Incompletely Specified Functions

- To minimize an incompletely specified function, we modify the procedure for circling sets of 1's (prime implicants) as follows:
  - 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

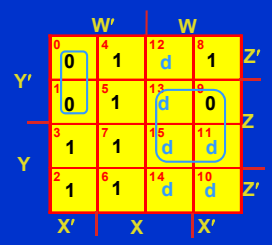
Most hardware description languages (HDL) provide a means for the designer to specify *don't care* inputs

BCD Inequality Detector Example: SOP



Minimum SP: $f(W,X,Y,Z) = X + Y + W \bullet Z'$
Cost: 5 gate inputs + 2 gate outputs = 7

BCD Inequality Detector Example: POS



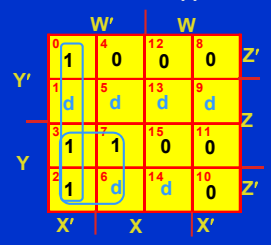
Minimum PS:
$f'(W,X,Y,Z) =$
$W \bullet Z + W' \bullet X' \bullet Y'$
$\rightarrow f(W,X,Y,Z) =$
$(W' + Z') \bullet (W + X + Y)$

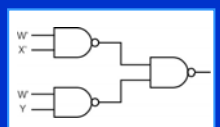
Cost: 7 gate inputs
+ 3 gate outputs = 10

Conclusion: The SP implementation *costs less*

Incompletely Specified Functions

- Example: Find a minimal *sum-of-products* expression for the function mapped below

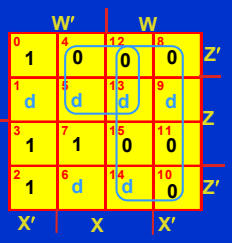


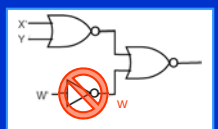


$f(W,X,Y,Z) = W' \bullet X' + W' \bullet Y$
Cost: 6 gate inputs +
3 gate outputs
= 9 cost units

Incompletely Specified Functions

- Example: Find a minimal *product-of-sums* expression for the function mapped below





$f'(W,X,Y,Z) = W + X \bullet Y'$
$f(W,X,Y,Z) = W' \bullet (X' + Y)$
Cost: 4 gate inputs +
2 gate outputs
= 6 cost units

Clicker Quiz

	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′

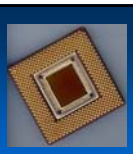
5. Assuming the availability of **only true** input variables, the **fewest number of 2-input open-drain NAND gates** that are needed to realize this function is:
   A. 6 **B.** 7 **C.** 8 **D.** 9 **E.** none of the above

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



Purdue IM:PACT* Spring 2018 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-C
Timing Hazards

Reading Assignment:
DDPP 4th Ed. pp. 224-229, 5th Ed. pp. 122-126

Learning Objectives:
- Define and identify static-0, static-1, and dynamic hazards
- Describe how a static hazard can be eliminated using consensus terms
- Describe a circuit that takes advantage of the existence of hazards and analyze its behavior
- Draw a timing chart that depicts the input-output relationship of a combinational circuit

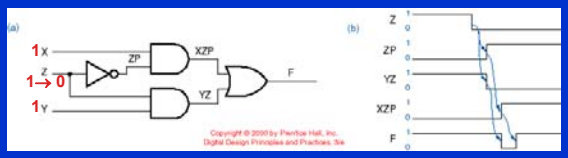
---

Outline
- Timing hazards
  - Static
  - Dynamic
- Elimination of timing hazards
- Clever utilization of timing hazards
- Designing hazard-free circuits

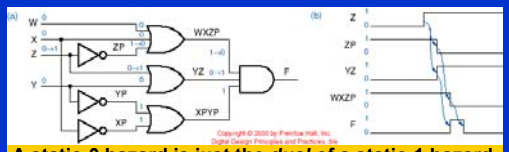
Timing Hazards
- The combinational circuit analysis methods described thus far *ignore* propagation delay and predict only the *steady state behavior*
- Gate propagation delay may cause the transient behavior of logic circuit to *differ* from that predicted by steady state analysis
- A circuit's output may produce a *short pulse* (often called a *glitch*) at time when steady state analysis predicts the output should not change
- A *hazard* is said to exist when a circuit has the possibility of producing such a glitch

---

Timing Hazards: Static 1
- Definition: 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



Timing Hazards: Static 0
- Definition: 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



A static-0 hazard is just the *dual* of a static-1 hazard

Timing Hazards

- A **K-map** can be used to **detect** static hazards in a two-level sum-of-products or product-of-sums circuit
- Important: The existence or nonexistence of static hazards depends on the *circuit design* (i.e., *realization*) of a logic function
- A properly designed two-level sum-of-products (AND-OR) circuit has no **static-0** hazards but *may* have **static-1** hazards
- Existence of static-1 hazards can be *predicted* from a K-map

Timing Hazards

- Using a K-map to graphically detect the possibility of a static-1 hazard:

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

f (X,Y,Z) = X•Z′+Y•Z

Note: 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"

Timing Hazards

- Solution: Include an extra product term (AND gate) to cover the hazardous input pair

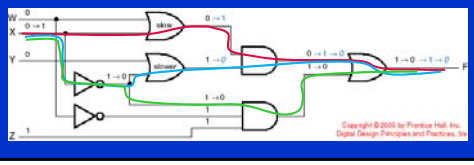
	X′		X	
Z′	0	0	1	1
Z	0	1	1	0
	Y′	Y	Y	Y′

f (X,Y,Z) = X•Z′+Y•Z + X•Y

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

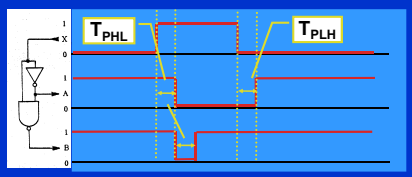
Timing Hazards

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



Timing Hazards

- Important: Not all hazards are hazardous – in fact, some can be quite useful! Consider the case in which we would like to detect a low-to-high transition (the "leading edge") of a logic signal

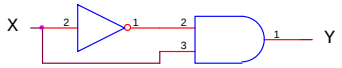


Designing Hazard-Free Circuits

- *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

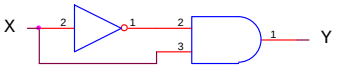
Clicker Quiz

139



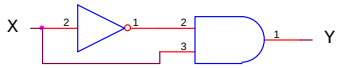
1. **Steady state analysis of this circuit would predict that its output will always be:**
   - A. **0**
   - B. **1**
   - C. **50% of V_{CC}**
   - D. ***Les Déplorables***
   - E. none of the above

140



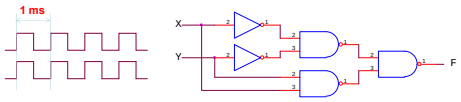
2. **This circuit exhibits the following type of hazard when its input, X, transitions from low-to-high:**
   - A. **static-0**
   - B. **static-1**
   - C. **dynamic**
   - D. ***Les Déplorables***
   - E. none of the above

141



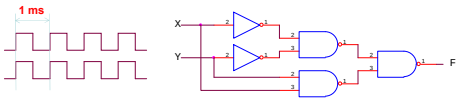
3. **This circuit exhibits the following type of hazard when its input, X, transitions from high-to-low:**
   - A. **static-0**
   - B. **static-1**
   - C. **dynamic**
   - D. ***Les Déplorables***
   - E. none of the above

142



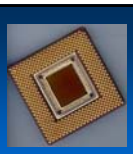
4. **Steady-state analysis of the function realized by this circuit for the input waveforms shown predicts that the output F(X,Y) should:**
   - A. **should always be low**
   - B. **should always be high**
   - C. **should be identical to the input**
   - D. **should be the complement of the input**
   - 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



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-D
XOR/XNOR Functions

Reading Assignment:
DDPP 4th Ed. pp. 447-448, 5th Ed. pp. 320-322

Learning Objectives:
- Identify properties of XOR/XNOR functions
- Simplify an otherwise non-minimizable function by expressing it in terms of XOR/XNOR operators

148

Outline
- XOR and XNOR functions
- XOR operator properties
- XOR "checkerboard" K-map
- XOR N-variable functions
- Realization of "non-reducible" functions using XOR/XNOR gates

147

XOR/XNOR Functions
- 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

148

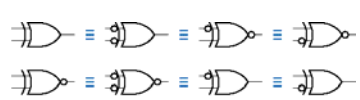
XOR/XNOR Functions
- The *"ring sum" operator* ⊕ is often used to denote the XOR function: $X \oplus Y = X' \cdot Y + X \cdot Y'$
- The XNOR function can be thought of as either the *dual* or the *complement* of the XOR function

$$(X \oplus Y)' = (X \oplus Y)^D = X' \cdot Y' + X \cdot Y$$

X	Y	X⊕Y	(X⊕Y)′
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

149

XOR Operator ⊕ Properties
- $X \oplus X = X' \cdot X + X \cdot X' = 0 + 0 = 0$
- $X' \oplus X' = X \cdot X' + X' \cdot X = 0 + 0 = 0$
- $X \oplus 1 = X' \cdot 1 + X \cdot 0 = X'$
- $X' \oplus 1 = X \cdot 1 + X' \cdot 0 = X$
- $(X \oplus Y)' = X \oplus Y \oplus 1$
- $X \oplus Y = Y \oplus X$
- $X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$
- $X \cdot (Y \oplus Z) = (X \cdot Y) \oplus (X \cdot Z)$



XOR and XNOR Equivalent Symbols

150

XOR K-Map

- K-map of 2-variable XOR function

$$X \oplus Y = X' \cdot Y + X \cdot Y'$$

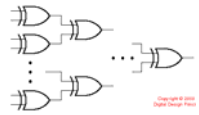
	X′	X
Y′	0	1
Y	1	0

Leads to a "checkerboard" K-map, that cannot be reduced (in either SoP or PoS form)

151

XOR N-Variable Functions

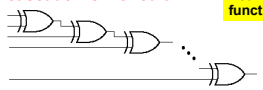
- The XOR (or XNOR) of *N* variables can be realized with *tree* or *cascade* circuits
  - tree XOR circuit (*N* is a power of 2)



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

- cascade XOR circuit



Realization of an n-variable XOR or XNOR function will require 2^{n-1} P-terms

152

Non-Reducible Functions

- 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

153

Example – "Diagonal" K-map

	W′		W		
Y′	1	0	0	0	Z′
	0	1	0	0	Z
Y	0	0	1	0	
	0	0	0	1	Z′
	X′	X		X′	

154

Example – "Diagonal" K-map

- Minimize function to the extent possible

	W′		W		
Y′	1	0	0	0	Z′
	0	1	0	0	Z
Y	0	0	1	0	
	0	0	0	1	Z′
	X′	X		X′	

F(W,X,Y,Z) =
W′•X′•Y′•Z′ + W′•X•Y′•Z
+ W•X•Y•Z + W•X′•Y•Z ′

155

Example – "Diagonal" K-map

- Factor out XOR/XNOR expressions

	W′		W		
Y′	1	0	0	0	Z′
	0	1	0	0	Z
Y	0	0	1	0	
	0	0	0	1	Z′
	X′	X		X′	

F(W,X,Y,Z) =
W′•X′•Y′•Z′ + W′•X•Y′•Z
+ W•X•Y•Z + W•X′•Y•Z ′

= X′•Z′ • (W′•Y′ + W•Y)
+ X•Z • (W′•Y′ + W•Y)

156

Example – "Diagonal" K-map

- Factor out XOR/XNOR expressions

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

F(W,X,Y,Z) =
W′•X′•Y′•Z′ + W′•X•Y′•Z
+ W•X•Y•Z + W•X′•Y•Z ′

= X′•Z′ • (W′•Y′ + W•Y)
+ X•Z • (W′•Y′ + W•Y)

= (X′•Z′ +X•Z)•(W′•Y′+W•Y)

157

Example – "Diagonal" K-map

- Write function in terms of XOR/XNOR operators

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

F(W,X,Y,Z) =
W′•X′•Y′•Z′ + W′•X•Y′•Z
+ W•X•Y•Z + W•X′•Y•Z ′

= X′•Z′ • (W′•Y′ + W•Y)
+ X•Z • (W′•Y′ + W•Y)

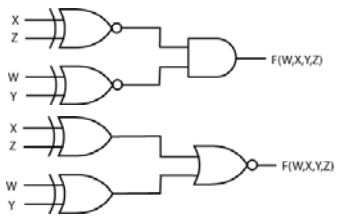
= (X′•Z′ +X•Z)•(W′•Y′+W•Y)

= (X ⊕ Z)′ • (W ⊕ Y)′

158

Example – "Diagonal" K-map

- Realize using XOR/XNOR gates

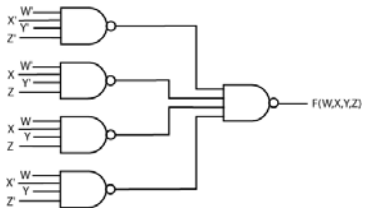


COST = 6 inputs + 3 outputs = 9

159

Example – "Diagonal" K-map

- Compare with minimal SoP realization



COST = 20 inputs + 5 outputs = 25

160

Example – "X"-map

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

161

Example – "X"-map

- Minimize function to the extent possible

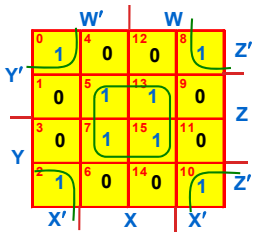
	W′		W		
Y′	0: 1	4: 0	12: 0	8: 1	Z′
	1: 0	5: 1	13: 1	9: 0	Z
Y	3: 0	7: 1	15: 1	11: 0	
	2: 1	6: 0	14: 0	10: 1	Z′
	X′	X		X′	

F(W,X,Y,Z) =
X′•Z′ + X•Z

162

Example – "X"-map

- **Write function in terms of XOR/XNOR operators**

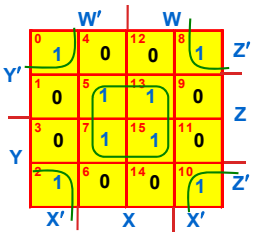


F(W,X,Y,Z) =
X′•Z′ + X•Z
= (X ⊕ Z)′

163

Example – "X"-map

- **Compare costs**

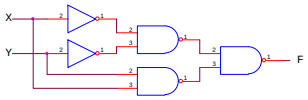


F(W,X,Y,Z) =
X′•Z′ + X•Z Cost=9
= (X ⊕ Z)′ Cost=3

164

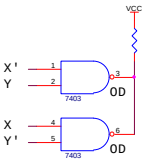
Clicker Quiz

165



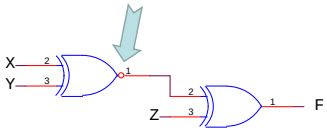
1. The function realized by this circuit is a:
   - A. **2-input XOR**
   - B. **2-input XNOR**
   - C. **2-input AND**
   - D. **2-input OR**
   - E. none of the above

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:

A. 

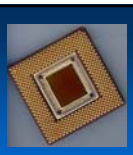
B. 

C. 

D. 

E. none of the above

170



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-E
Programmable Logic Devices

Reading Assignment:
DDPP 4th Ed. pp. 370-383, 840-859; 5th Ed. pp. 246-252

Learning Objectives:

- Describe the genesis of programmable logic devices
- List the differences between complex programmable logic devices (CPLDs) and field programmable gate arrays (FPGAs) and describe the basic organization of each

172

Outline

- Overview
- Programmable Logic Arrays (PLAs)
- Programmable Array Logic (PALs)
- Generic Array Logic (GALs)
- Complex PLDs
- Field Programmable Gate Arrays (FPGAs)
- Summary

173

Overview

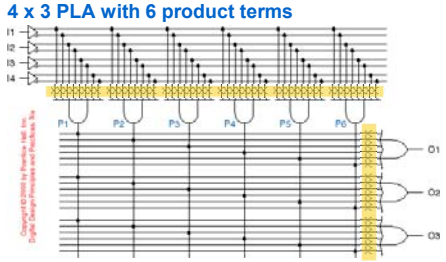
- The first programmable logic devices (PLDs) were programmable logic arrays (PLAs)
- PLAs are combinational, **two-level AND-OR devices** that can be programmed to realize and sum-of-products expression
- Limitations
  - number of inputs (*n*)
  - number of outputs (*m*)
  - number of product ("P") terms (*p*)

Such a device might be described as an *n* x *m* PLA with *p* product terms

174

Programmable Logic Array

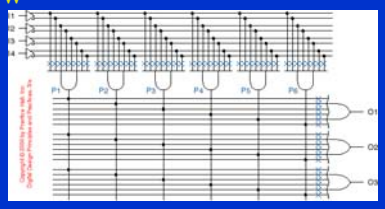
- **4 x 3 PLA with 6 product terms**



Potential connections indicated by "X"

175

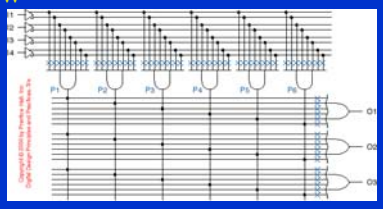
Overview



- Each input is connected to a ***buffer*** that produces ***both a true and a complemented*** version of the signal for use in the array
- Connections are made by ***fuses***, which are actual ***fusible links*** (one-time programmable devices) or ***non-volatile memory cells*** (erasable, re-programmable devices)

176

Overview

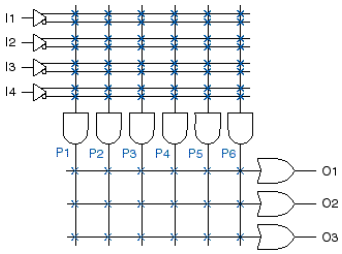


- Each AND gate's inputs can be any subset of the primary input signals and their complements
- Each OR gate's inputs can be any subset of the AND gate outputs

177

Programmable Logic Array

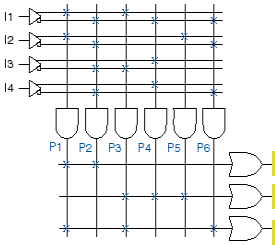
- **Compact view of 4 x 3 PLA with 6 P-terms**



178

Programmable Logic Array

- **4 x 3 PLA programmed to implement three logic equations**



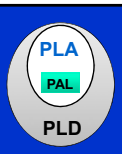
I1•I2 + I1′•I2′•I3′•I4′

I1•I3′ + I1′•I3•I4 + I2

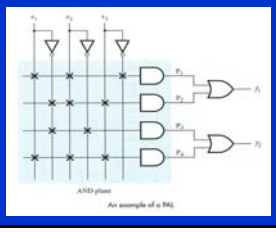
I1•I2 + I1•I3′ + I1′•I2′•I4′

179

Programmable Array Logic



- A special case of PLA is the ***programmable array logic*** (PAL)
- Unlike a PLA, a PAL device has a *fixed* OR array (i.e. AND gates can not be shared)
- Each output has an individual tri-state enable, controlled by a dedicated AND gate
- There is an inverter between the output of the OR gate and the external pin
- Some of the output pins may also be used as inputs (called "I/O pins")
  - tri-state buffer OFF, *input only*
  - tri-state buffer ON, either *output-only*, output *cascaded* to another function input, or *feedback* to create a sequential circuit



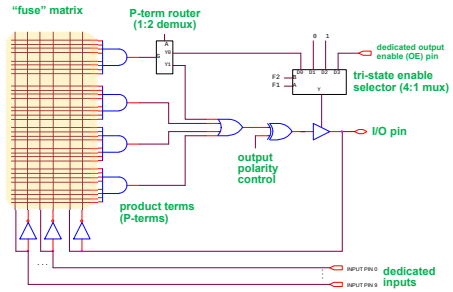
180

Generic Array Logic

- 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*)
- A legacy GAL device (22V10) is included in your digital parts kit to provide an introduction to PLDs

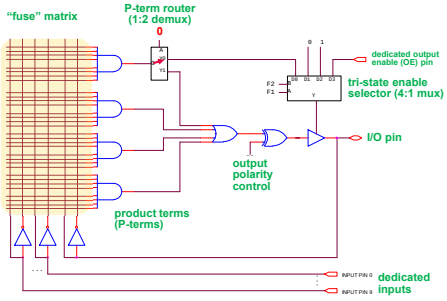
181

GAL Combinational Macrocell



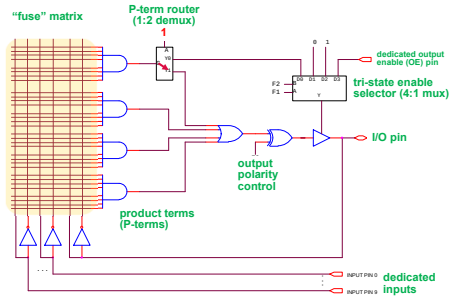
182

GAL Combinational Macrocell



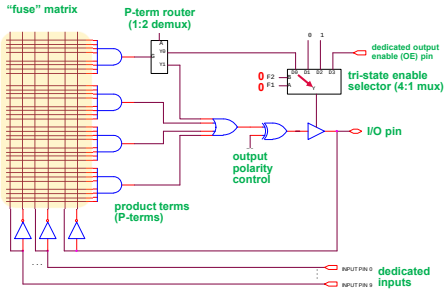
183

GAL Combinational Macrocell



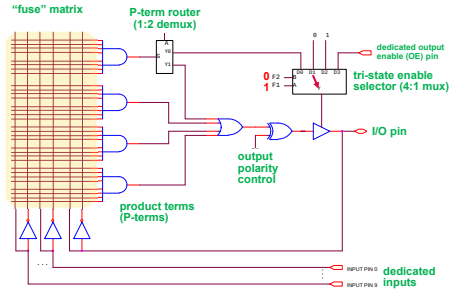
184

GAL Combinational Macrocell



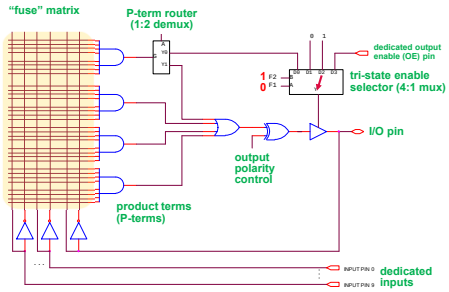
185

GAL Combinational Macrocell



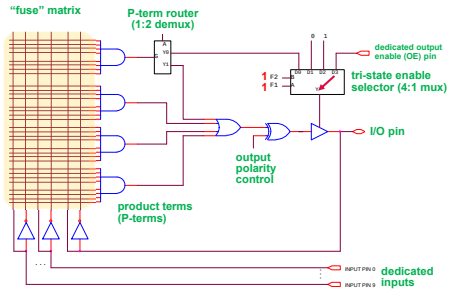
186

GAL Combinational Macrocell



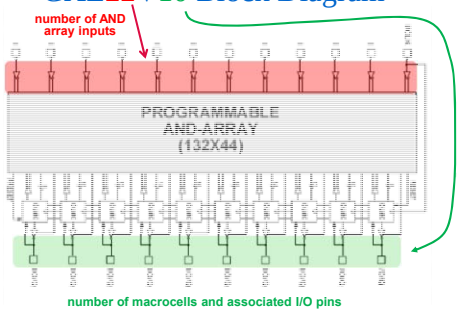
187

GAL Combinational Macrocell



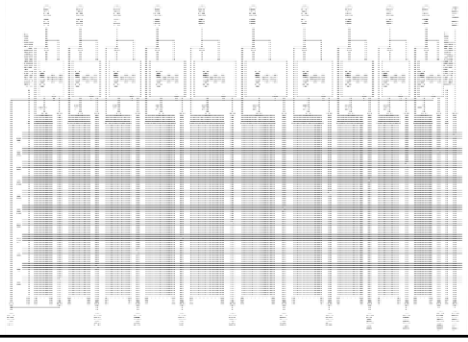
188

GAL22V10 Block Diagram



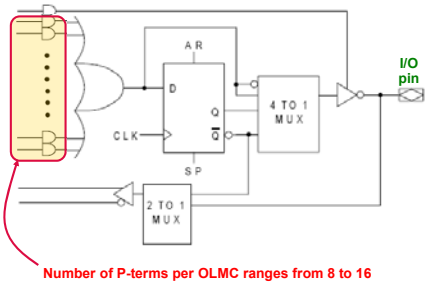
189

GAL22V10 AND Array ("Fuse Matrix")



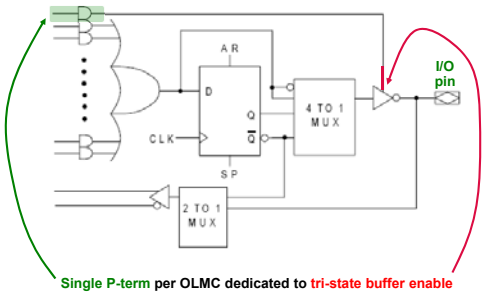
190

GAL22V10 Output Logic Macrocell ("OLMC")



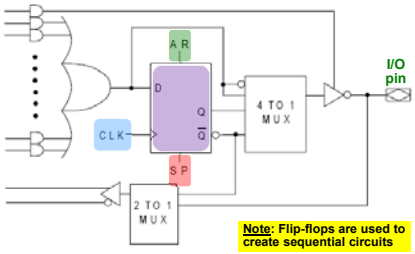
191

GAL22V10 Output Logic Macrocell ("OLMC")



192

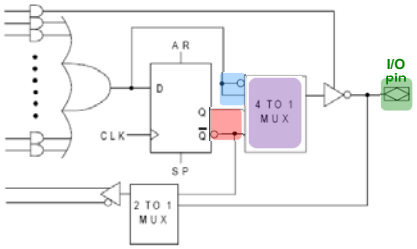
GAL22V10 Output Logic Macrocell ("OLMC")



All OLMC edge-triggered D flip-flops utilize common clock (CLK), asynchronous reset (AR), and asynchronous preset (SP) signals

193

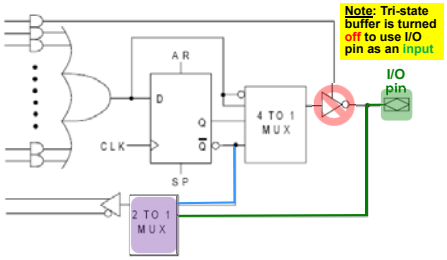
GAL22V10 Output Logic Macrocell ("OLMC")



4:1 multiplexer selects (routes) true/complemented combinational or true/complemented registered function to the I/O pin

194

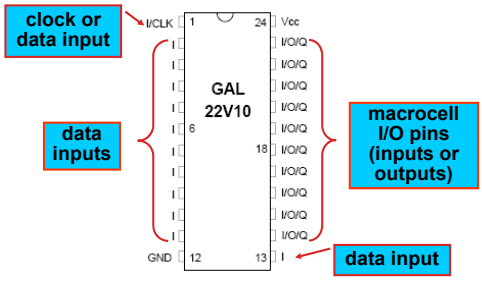
GAL22V10 Output Logic Macrocell ("OLMC")



2:1 multiplexer selects (routes) true/complemented I/O pin or true/complemented registered feedback to the P-term array

195

GAL22V10 Pinout



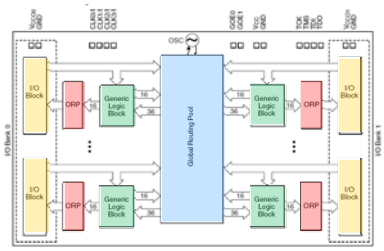
196

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*)
- Example: The Lattice ispMACH 4000 series CPLDs feature 36-input, 16-macrocell GLBs
- A "breakout board" utilizing an ispMACH 4256ZE device (with 256 macrocells and 144 pins) will be used for the second half of the lab experiments

197

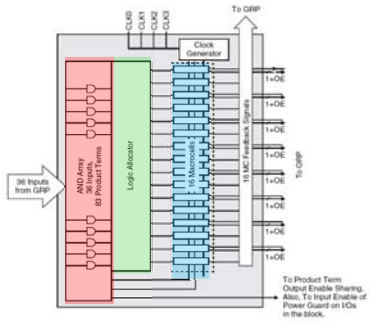
ispMACH 4000ZE Block Diagram



- 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

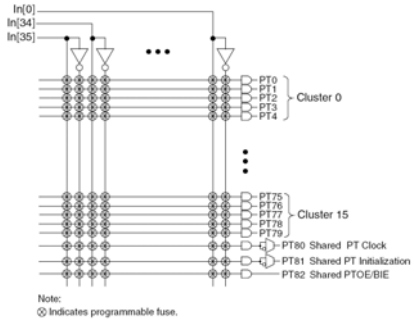
198

ispMACH 4000ZE Generic Logic Block



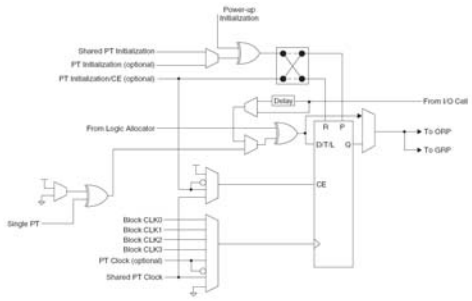
199

ispMACH 4000ZE 36-Input AND Array



200

ispMACH 4000ZE Macrocell



201

Field Programmable Gate Arrays

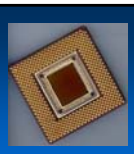
- A field programmable gate array (FPGA) is "kind of like a CPLD turned inside-out"
- Logic is broken into a large number of programmable blocks called ***look-up tables*** **(LUTs)** or ***configurable logic blocks*** **(CLBs)**
- Programming configuration is stored in **SRAM-based memory cells** and 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)

202

Summary

- There are currently two types of programmable logic devices in common use:
  - CPLDs
    - in-circuit programmable
    - non-volatile (retains configuration information when powered down)
    - "instant on" (no external configuration ROM or boot sequence required)
    - less dense (fewer programmable logic blocks) than comparably sized FPGA
  - FPGAs
    - in-circuit programmable
    - volatile (loses configuration when powered down)
    - requires external configuration ROM and "boot" sequence to initialize
    - more dense (greater number programmable logic blocks) than comparably sized CPLD

203



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-F
Hardware Description Languages

Reading Assignment:
DDPP 4th Ed. pp. 237-243, 290-335; 5th Ed. pp. 177-233

Learning Objectives:

- List the basic features and capabilities of a hardware description language
- List the syntactic elements of a Verilog module
- Identify operators and keywords used to create Verilog modules
- Write equations using Verilog dataflow syntax
- Define functional behavior by creating truth tables with the casez construct in Verilog

Outline

- Overview
- Verilog and ispLever™
- Verilog coding semantics
- Verilog module structure
- Verilog symbols for logical operations
- Sample Verilog modules
- Structural code in Verilog

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

Overview

- Both VHDL and Verilog started out as *simulation languages* (later developments in these languages allowed actual hardware design)
- Both languages 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)
- Advantage – VHDL and Verilog are much better adapted to large scale system design Verilog has become the most common language for IC design and verification.

"Deep into levels of abstraction, your descent will be"

Verilog and ispLever™

- Because Verilog is so commonly used in industry and you will need it in future classes, you will be introduced to Verilog in this course
- 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 ABEL, Verilog, and VHDL as well as schematic entry
- You can obtain your own free copy of this software from the Lattice Semiconductor web site (www.latticesemi.com)

Verilog and ispLever™

- A Verilog module is a text file containing:
  - 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
- Because you need to be able to program a PLD or CPLD, your Verilog code must be strictly limited to syntax that translates neatly into logic circuitry
- Verilog source files are transformed into a fuse map file by the compiler integrated into ispLever
- A universal programmer is used to burn the fuse map file into a legacy PLD device (an isp device can be programmed directly from the integrated ispVM tool via a USB cable)

Verilog Program Semantics

- *identifiers* (module names, signal/variable 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

211

Verilog WIRE Type

- wire is a basic data type in Verilog
- *Similar to an actual wire*, these variables cannot store logic values 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

212

Verilog BITWISE Operators

&	and
|	or
~	not
^	exclusive or
~^ *or* ^~	exclusive nor

You will learn about logical vs. bitwise operators later (similar to C)

213

ispLEVER Operators

Reports generated by ispLever use a *different notation* for some of the bitwise operators

Logical operation	Verilog	ispLEVER
AND	&	&
OR	|	#
NOT	~	!
XOR	^	$

214

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

```
wire [2:0]  A,B,X,Y;
assign  A = 3'b110;
assign  B = 3'b101;
assign  X = A & B;
assign  Y = A | B;
```

3-bit wires A, B, X, and Y

A is assigned the constant 3-bit value of 110

B is assigned the constant 3-bit value of 101

wire X is assigned the value of A bitwise AND-ed with B i.e. 100

wire Y is assigned the value of A bitwise OR-ed with B i.e. 111

215

Verilog MODULE Structure (Example 1)

```
// comments start with double slash, keywords highlighted in red
/* or they can be bounded with slash star as in C */
module nand_nor(Sel,A,B,Y);
  input wire Sel, A, B;
  output wire Y;

  wire Y1, Y2, Y3, Y4;

  assign Y = Y3 | Y4;
  assign Y1 = A & B ;
  assign Y2 = A | B;
  assign Y3 = (~Y1) & Sel;
  assign Y4 = (~Y2) & (~Sel);
endmodule
```

Describes a circuit called nand_nor with inputs Sel, A, B, and output Y

4 individual wire names Y1 .. Y4

Each assign statement describes a separate piece of logic with the output on the left and operations on inputs on the right

216

Verilog MODULE Structure (Example 1)

```
module nand_nor(Sel,A,B,Y);
  input wire Sel, A, B /* synthesis loc="4,5,6" */;

  output wire Y /* synthesis loc="7" */;

  wire Y1, Y2, Y3, Y4;

  assign Y = Y3 | Y4;
  assign Y1 = A & B ;
  assign Y2 = A | B;
  assign Y3 = (~Y1) & Sel;
  assign Y4 = (~Y2) & (~Sel);
endmodule
```

synthesis loc is a compiler directive that tells ispLever to connect Sel, A, B, and Y to pins 4, 5, 6, and 7 (respectively) on the PLD

wire is one of several signal variable types you will learn to use

assign statements are not the only way of describing your logic, but they are the *simplest* for very small combinational logic designs

217

Verilog MODULE Structure (Example 1)

```
module nand_nor(Sel,A,B,Y);
  input wire Sel, A, B /* synthesis loc="4,5,6" */;

  output wire [1:0] Y /* synthesis loc="7,8" */;

  wire Y1, Y2, Y3,Y4;

  assign Y = {Y3,Y4};
  assign Y1 = A & B ;
  assign Y2 = A | B;
  assign Y3 = (~Y1) & Sel;
  assign Y4 = (~Y2) & (~Sel);
endmodule
```

The index range [1:0] makes Y into a 2-bit vector
Y[1] assigned to pin 7, Y[0] pin 8

The concatenation operator { } makes a bit vector out of multiple wires

218

Verilog BIT Literals

```
wire a,b;
wire [2:0] Y;

assign a = 1'b0;
assign b = 1'b1;
assign Y = 3'b100;
```

1 bit equal to binary 0

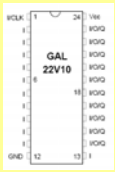
1 bit equal to binary 1

3 bits equal to 100_2
Y[2]=1'b1 Y[1]=1'b0 Y[0]=1'b0

219

Example Verilog Module #1A

```
/* 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
```

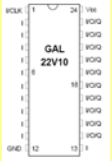


Note: Explicit pin declarations can be omitted and automatically assigned by the "fitter" program (part of ispLever)

220

Example Verilog Module #1B

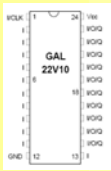
```
/* Verilog Combinational Example for GAL22V10
   with active low inputs */

// "n" prefix is just a naming convention
module verilog_exA(nA,nB,nC,nD,X,Y,Z);
input nA,nB,nC,nD /* synthesis loc="2,3,4,5" */;
output X,Y,Z      /* synthesis loc="14,15,16" */;

wire A,B,C,D;
assign A = ~nA;  // to treat inputs as
assign B = ~nB;  // active low, you must
assign C = ~nC;  // invert them
assign D = ~nD;

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

endmodule
```



221

Example Verilog Module #1C

```
/* Verilog Combinational Example for GAL22V10
   with active low inputs and outputs */

module verilog_exA(nA,nB,nC,nD,nX,nY,nZ);
input nA,nB,nC,nD   /* synthesis loc="2,3,4,5" */;
output nX,nY,nZ     /* synthesis loc="14,15,16" */;

wire A,B,C,D;
assign A = ~nA;  // to treat inputs as
assign B = ~nB;  // active low, you must
assign C = ~nC;  // invert them
assign D = ~nD;

// to make outputs active low, invert the
// value assigned to the output
assign nX = ~( (A & B) | ~(C & D) );
assign nY = ~( ~(B & D) | ~(A & B & D) );
assign nZ = ~( A & ~(B & C & ~D) );

endmodule
```



222

Verilog REG Data Types

- Similar data type to wire, but reg can be used to store information
- 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
- Examples:

```
reg myvar;          // one bit variable called myvar
reg [7:0] myvec;    // 8-bit variable called myvec
```

223

ALWAYS Block in Verilog

- An 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

224

ALWAYS Block in Verilog

Example always blocks:

```
always @ (A,B,C) begin
…
end
```

All statements will be evaluated *whenever A, B, or C change their values*

```
always @ (posedge CLK) begin
…
end
```

All statements will be evaluated on the *positive (rising) edge of CLK signal* (use *negedge* for falling edge of CLK)

```
always @ (*) begin
…
end
```

All statements will be evaluated *whenever any input signal in the always block changes*

225

Verilog 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. e.g. a 3-bit signal for comparison will need a default case if 8 cases are not provided

226

Verilog MODULE Structure (Example 2)

```
module nand_nor(Sel,A,B,Y);
  input Sel, A, B /* synthesis loc="4,5,6" */;

  output reg Y    /* synthesis loc="7" */;

always @ (Sel,A,B) begin
  case ({Sel,A,B})
    3'b000: Y = 1'b1;  // row 0
    3'b001: Y = 1'b1;  // row 1
    3'b010: Y = 1'b1;  // row 2
    // (remaining combinations)
   default:  Y = 1'b0;  // or use a default case
  endcase
endmodule
```

Y must be declared as reg type to be an output of an always block

This is the closest structure available in Verilog to a traditional "truth table"

@(Sel,A,B) is a sensitivity list

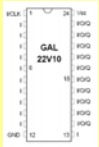
For combinational logic, list all inputs

Compares each case against a concatenated 3-bit vector with Sel at bit position 2, A at position 1 and B at position 0 and evaluates value of Y based on matching case, e.g. 3'b001 matches Sel=0,A=0,B=1

227

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'b01000;
      4'b0001: abcdf = 5'b00010;
      4'b0010: abcdf = 5'b00100;
      4'b0011: abcdf = 5'b00010;
      4'b0100: abcdf = 5'b10000;
      4'b0101: abcdf = 5'b10000;
      4'b0110: abcdf = 5'b00100;
      4'b0111: abcdf = 5'b10000;
      4'b1000: abcdf = 5'b01000;
      4'b1001: abcdf = 5'b01000;
      4'b1010: abcdf = 5'b00100;
      4'b1011: abcdf = 5'b00001;
      4'b1100: abcdf = 5'b10000;
      4'b1101: abcdf = 5'b10000;
      4'b1110: abcdf = 5'b00100;
      4'b1111: abcdf = 5'b10000;
    endcase
  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

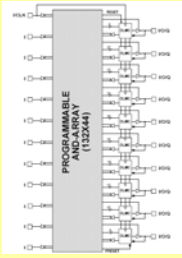
e.g. 4'b1011 matches E=1,R=0,S=1,T=1

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

228

Example Verilog Module #3A

```
/* 4 variable XOR on 22V10 */
module xor_exA (I, X);
  input [3:0] I;
  output X;
  // if synthesis_loc not given,
  // then ISPlever will choose pin #

  // bitwise &, |, ^ can be used as
  // reduction operators on a vector

  assign X = ^I;

  // you could also write this as
  // X = I[3]^I[2]^I[1]^I[0];

endmodule
```



NOTE: Each XOR gate increases P-terms by a factor of 2 (number of P-terms = 2^{n-1})

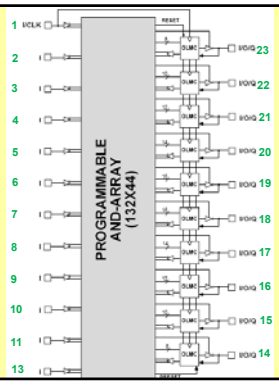
Equation requires 8 P-terms → can be realized on any 22V10 macrocell (any I/O pin)

229

Example Verilog Module #3B

```
/* 5 variable XOR on 22V10 */
module xor_exA (I, X);
  input [4:0] I;
  output X;

  assign X = ^I;

endmodule
```

Equation requires 16 P-terms → can be realized on macrocells associated with I/O pins 18 & 19

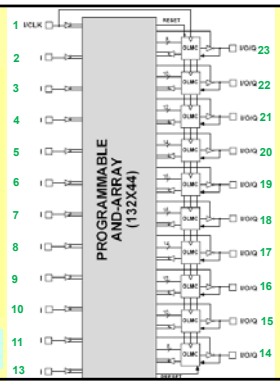


230

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)



231

Structural Code in Verilog

- 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

232

Verilog Built-in Primitives

- and
- or
- nand
- nor
- xor
- xnor
- not
- buf
- bufif0
- bufif1
- notif0
- notif1

Usage of built-in primitives is illustrated in the next slide. The same syntax can be used for user-defined modules as well.

For more information, refer to Section 5.7 in the Wakerly text.

233

Structural Code in Verilog

- Example illustrating multiple modules connected

```
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

234

Clicker Quiz

235

1. Which of the following is **not** a valid Verilog identifier?
   A. **X2**
   B. **2X**
   C. **XY**
   D. **_XY**
   E. none of the above

236

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

237

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

238

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

239

Example – Your BFFAM's "Crazy Grader"

Your "best friend from another major" (BFFAM) has been asked to design a circuit that determines grades based on the characters (E,R,S,T) in a student's last name, as follows:

- **Give a grade of "A" if name contains an R *and* a T *-or-* an R *and not* an S**
- **Give a grade of "B" if name contains an E *and not* an R *and not* a S *-or-* does *not* contain an R *and not* a T *and not* an S**
- **Give a grade of "C" if name contains an S *and not* a T**
- **Give a grade of "D" if name contains a T *and not* an E *and not* an R**
- **Give a grade of "F" if none of the above (name contains an E *and* an S *and* a T *and not* an R)**

240

K-Map of "Grade Distribution"

	E′		E		
S′	B	A	A	B	T′
	D	A	A	B	T
S	D	A	A	F	
	C	C	C	C	T′
	R′	R		R′	

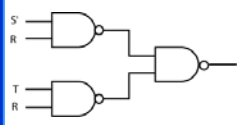
Options

- Map and minimize all 5 functions, implement with several discrete CMOS ICs, subject to the following limitations:
  - only "true" variables are available
  - only SSI chips in digital kit can be used
    - 7400 quad 2-input NAND
    - 7402 quad 2-input NOR
    - 7404 hex inverter
    - 7410 triple 3-input NAND
- Create a Verilog file that specifies the desired functionality using a truth table, implement with a single 22V10 PLD

Working K-Map for "A" – SoP

	E′		E		
S′	0	1	1	0	T′
	0	1	1	0	T
S	0	1	1	0	
	0	0	0	0	T′
	R′	R		R′	

A = S′•R + T•R

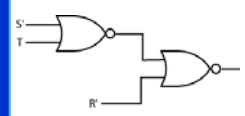


COST = 6 inputs + 3 outputs = 9

Working K-Map for "A" – PoS

	E′		E		
S′	0	1	1	0	T′
	0	1	1	0	T
S	0	1	1	0	
	0	0	0	0	T′
	R′	R		R′	

A′ = R′ + S•T′
A = R • (S′ + T)



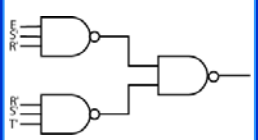
COST = 4 inputs + 2 outputs = 6

Cheaper than SoP

Working K-Map for "B" – SoP

	E′		E		
S′	1	0	0	1	T′
	0	0	0	1	T
S	0	0	0	0	
	0	0	0	0	T′
	R′	R		R′	

B = E•S′•R′ + R′•S′•T′

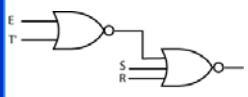


COST = 8 inputs + 3 outputs = 11

Working K-Map for "B" – PoS

	E′		E		
S′	1	0	0	1	T′
	0	0	0	1	T
S	0	0	0	0	
	0	0	0	0	T′
	R′	R		R′	

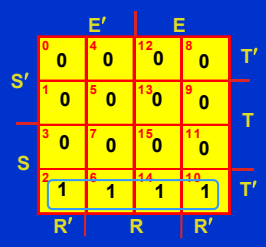
B′ = S + R + E′•T
B = S′• R′•(E+T′)



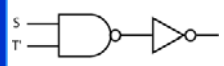
COST = 5 inputs + 2 outputs = 7

Cheaper than SoP

Working K-Map for "C" – SoP

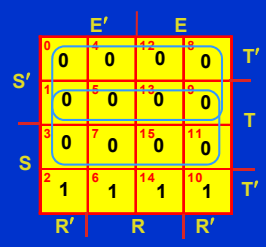


C = S•T′

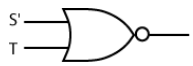


COST = 3 inputs
+ 2 outputs = 5

Working K-Map for "C" – PoS



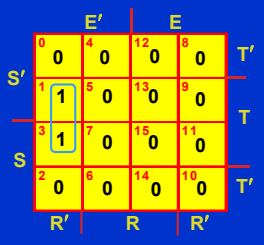
C′ = S′ + T
C = S•T′



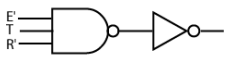
COST = 2 inputs
+ 1 output = 3

Cheaper than SoP

Working K-Map for "D" – SoP

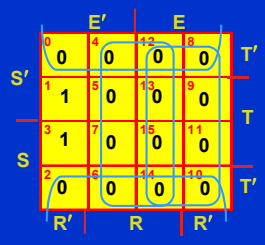


D = E′•T•R′

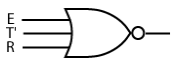


COST = 4 inputs
+ 2 outputs = 6

Working K-Map for "D" – PoS



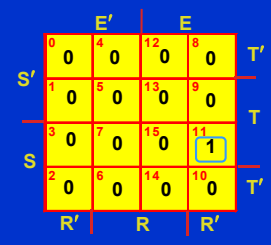
D = E′•T•R′



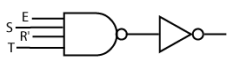
COST = 3 inputs
+ 1 output = 4

Cheaper than SoP

Working K-Map for "F" - SoP

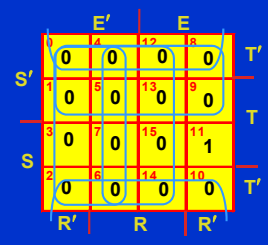


F = E•S•R′•T

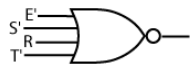


COST = 5 inputs
+ 2 outputs = 7

Working K-Map for "F" - PoS



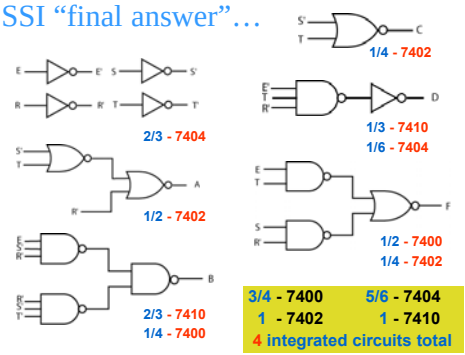
F′ = E′+S′+R+T′
F = E•S•R′•T



COST = 4 inputs
+ 1 output = 5

Cheaper than SoP

SSI "final answer"…

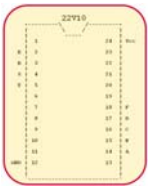


3/4 - 7400 5/6 - 7404
1 - 7402 1 - 7410
4 integrated circuits total

253

Verilog "final answer"…

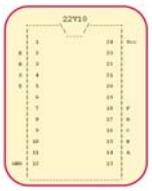

```
/* Who Wants to be a Digijock */

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" */;

  reg [4:0] ABCDF;

  always @ (E, R, S, T) begin
    case ({E,R,S,T})
      4'b0000:  ABCDF = 5'b01000;
      4'b0001:  ABCDF = 5'b00010;
      4'b0010:  ABCDF = 5'b00100;
      4'b0011:  ABCDF = 5'b00010;
      4'b0100:  ABCDF = 5'b10000;
      4'b0101:  ABCDF = 5'b10000;
      4'b0110:  ABCDF = 5'b00100;
      4'b0111:  ABCDF = 5'b10000;
      4'b1000:  ABCDF = 5'b01000;
      4'b1001:  ABCDF = 5'b01000;
      4'b1010:  ABCDF = 5'b00100;
      4'b1011:  ABCDF = 5'b00001;
      4'b1100:  ABCDF = 5'b10000;
      4'b1101:  ABCDF = 5'b10000;
      4'b1110:  ABCDF = 5'b00100;
      4'b1111:  ABCDF = 5'b10000;
    endcase
  end

  assign {A,B,C,D,F} = ABCDF;

endmodule
```



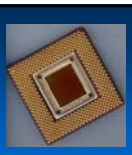
254

Are you sure that's your final answer?


```
/* 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
```



255



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-G

Combinational Building Blocks: Decoders / Demultiplexers

Reading Assignment:

DDPP 4th Ed. pp. 384-390, 403-409; 5th Ed. pp. 250-256, 260-278

Learning Objectives:

- Define the function of a decoder (demultiplexer) and describe how it can be used as a combinational building block
- Illustrate how a decoder can be used to realize an arbitrary Boolean function

257

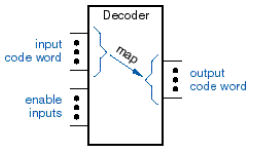
Outline

- Overview
- Binary decoders
- Decoders in Verilog
- Special purpose decoders

258

Overview

- **Definition: A decoder is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs**
- **The input code generally has fewer bits than the output code**
- **In a one-to-one mapping, each input code word produces a different output code word**



259

Overview

- **The most commonly used *input code* is an n-bit binary code, where an n-bit word represents one of 2^n different coded values**
- **Sometimes an n-bit binary code is *truncated* to represent fewer than 2^n values (e.g., BCD)**
- **The most commonly used *output code* is a 1-out-of-m code, which contains m bits, where only one bit is asserted at any time (the output code bits are *mutually exclusive*)**

260

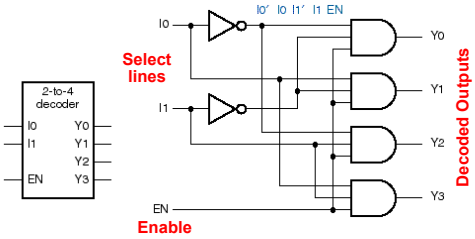
Binary Decoders

- **The most common decoder circuit is an n-to-2^n decoder or *binary decoder***
- **Binary decoders have an n-bit binary input code and a 1-out-of-2^n output code**
- **Application: Used to activate exactly one of 2^n outputs based on an n-bit value**
- **Analogy: Electronically-controlled rotary selector switch**

A device that routes an input to one of 2^n outputs is typically referred to as a (1-to-2^n) **demultiplexer**

261

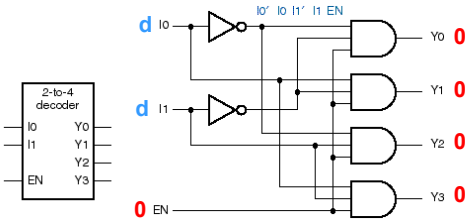
Example: 2-to-4 (2:4) Decoder



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 (1:4) **demultiplexer**

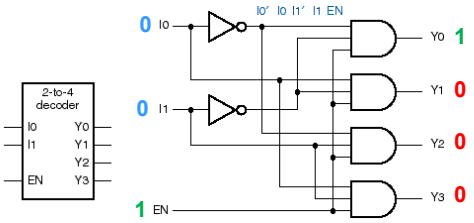
262

Example: 2-to-4 (2:4) Decoder



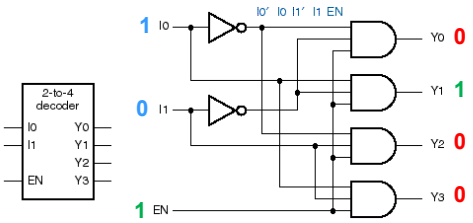
263

Example: 2-to-4 (2:4) Decoder



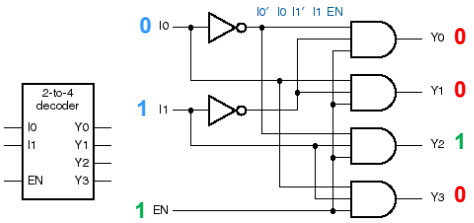
264

Example: 2-to-4 (2:4) Decoder



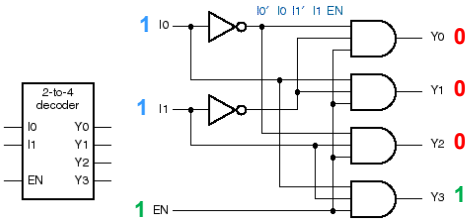
265

Example: 2-to-4 (2:4) Decoder



266

Example: 2-to-4 (2:4) Decoder



267

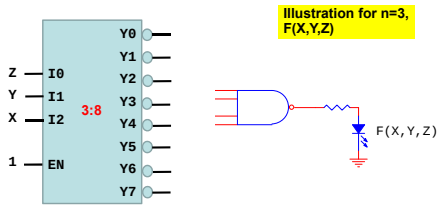
Key Observations

- **Key Observation #1**: each output of an n to 2^n 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
- **Key Observation #2**: if the decoder outputs are active low, a NAND gate can be used to "OR" the minterms of the function (representing its ON set)
- **Key Observation #3**: 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)
- **Key Observation #4**: a NAND gate (or AND gate) with at most 2^{n-1} inputs is needed to implement an arbitrary n-variable function using an n to 2^n binary decoder (that has active low outputs)

268

Example – Arbitrary Function Realization

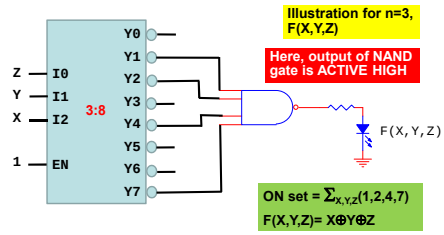
General circuit for implementing an arbitrary n-variable function using a decoder with active low outputs and a NAND gate with 2^{n-1} inputs, for case where the ON set has $\leq 2^{n-1}$ members



269

Example – Arbitrary Function Realization

General circuit for implementing an arbitrary n-variable function using a decoder with active low outputs and a NAND gate with 2^{n-1} inputs, for case where the ON set has $\leq 2^{n-1}$ members



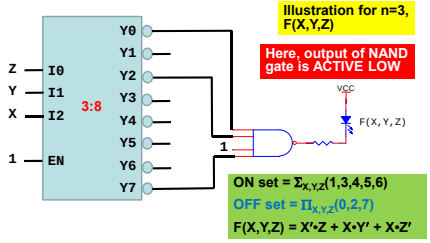
270

Example – Arbitrary Function Realization

General circuit for implementing an arbitrary n-variable function using a decoder with active low outputs and a NAND gate with 2^{n-1} inputs, for case where the ON set has > 2^{n-1} members

Illustration for n=3, F(X,Y,Z)

Here, output of NAND gate is ACTIVE LOW



ON set = $\Sigma_{X,Y,Z}(1,3,4,5,6)$

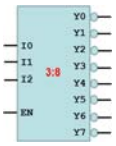
OFF set = $\Pi_{X,Y,Z}(0,2,7)$

F(X,Y,Z) = X'•Z + X•Y' + X•Z'

271

Decoders in Verilog

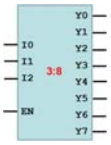
```
/* 3:8 Decoder / 1:8 Demultiplexer with Active-Low Outputs */

module dec38L(EN, I, nY);

  input wire EN;            // Enable input pin
  input wire [2:0] I;       // Select input pins
  output wire [7:0] nY;     // Active-low output pins

  wire [7:0] Y;

  assign nY = ~Y;      // Active low assignment

  assign Y[0] = EN & ~I[2] & ~I[1] & ~I[0];
  assign Y[1] = EN & ~I[2] & ~I[1] &  I[0];
  assign Y[2] = EN & ~I[2] &  I[1] & ~I[0];
  assign Y[3] = EN & ~I[2] &  I[1] &  I[0];
  assign Y[4] = EN &  I[2] & ~I[1] & ~I[0];
  assign Y[5] = EN &  I[2] & ~I[1] &  I[0];
  assign Y[6] = EN &  I[2] &  I[1] & ~I[0];
  assign Y[7] = EN &  I[2] &  I[1] &  I[0];

endmodule
```



272

Decoders/Demultiplexers in Verilog

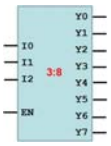
```
/* 3:8 Decoder / 1:8 Demultiplexer with Active-High Outputs */

module dec38H(EN, I, Y);

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

  assign Y[0] = EN & ~I[2] & ~I[1] & ~I[0];
  assign Y[1] = EN & ~I[2] & ~I[1] &  I[0];
  assign Y[2] = EN & ~I[2] &  I[1] & ~I[0];
  assign Y[3] = EN & ~I[2] &  I[1] &  I[0];
  assign Y[4] = EN &  I[2] & ~I[1] & ~I[0];
  assign Y[5] = EN &  I[2] & ~I[1] &  I[0];
  assign Y[6] = EN &  I[2] &  I[1] & ~I[0];
  assign Y[7] = EN &  I[2] &  I[1] &  I[0];

endmodule
```



273

Decoders/Demultiplexers in Verilog

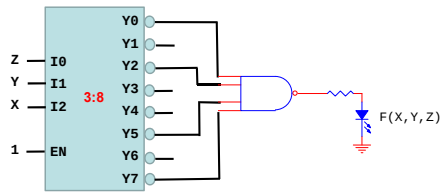
```
/* 3:8 Decoder / 1:8 Demultiplexer with Active-High Outputs */

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
```



274

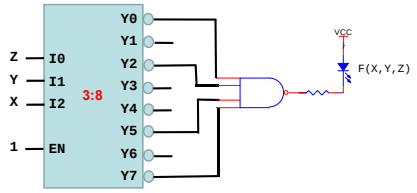
Clicker Quiz

275



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

276



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. $\Sigma_{X,Y,Z}(1,2,4,5)$

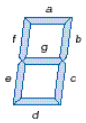
D. $\Sigma_{X,Y,Z}(0,3,4,6)$

E. none of the above

277

Special Purpose Decoders

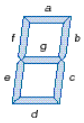
- **A seven-segment decoder has 4-bit BCD or hexadecimal data as its input code and "seven-segment code" as its output code**



278

Example: Hexadecimal 7-Segment Decoder

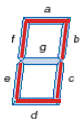
```
/* Hexadecimal 7-Segment Decoder for 22V10 */

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
```



279

Example: Hexadecimal 7-Segment Decoder

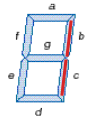
```
/* Hexadecimal 7-Segment Decoder for 22V10 */

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
```



280

Example: Hexadecimal 7-Segment Decoder

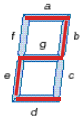
```
/* Hexadecimal 7-Segment Decoder for 22V10 */

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
```



281

Example: Hexadecimal 7-Segment Decoder

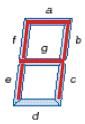
```
/* Hexadecimal 7-Segment Decoder for 22V10 */

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
```



282

Example: Hexadecimal 7-Segment Decoder

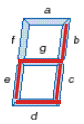
```
/* Hexadecimal 7-Segment Decoder for 22V10 */

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
```



283

Example: Hexadecimal 7-Segment Decoder

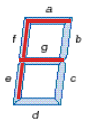
```
/* Hexadecimal 7-Segment Decoder for 22V10 */

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
```



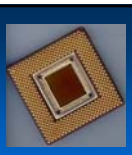
284

Example: Hexadecimal 7-Segment Decoder

```
/* 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
```



285



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-H

Combinational Building Blocks: Encoders and Tri-State Outputs

Reading Assignment:

DDPP 4th Ed. pp. 408-412, 430-432; 5th Ed. 279-280, 308-310

Learning Objectives:

- Define the function of an encoder and describe how it can be used as a combinational building block
- Discuss why the inputs of an encoder typically need to be prioritized

287

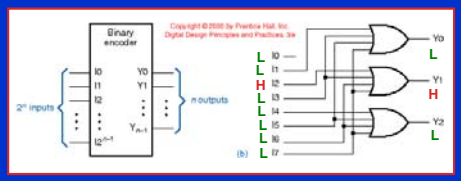
Outline

- Overview
- Priority Encoders
- Tri-State Outputs
- Keypad Encoders

288

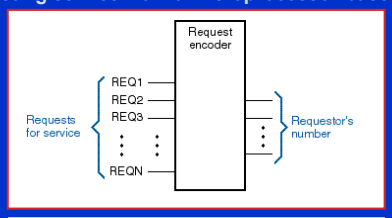
Overview

- **Definition**: 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
- The simplest encoder to build is a 2^n-to-n or binary encoder



Priority Encoders

- A common application is to encode the number of a device requesting service from a microprocessor-based system



Problem: More than one device may be requesting service at any given time

Priority Encoders

- **Solution**: Assign *priority* to the input lines, such that when multiple inputs are asserted simultaneously, the *highest priority* (i.e. *highest numbered*) input "wins" – such a device is called a *priority encoder*
- An easy way to specify this functionality in Verilog is to use the **casez** construct
- **Example**: An 8-to-3 encoder with active high inputs and outputs, including a "strobe" output (G) to indicate if any input has been asserted

Verilog CASEZ Construct

- use ? as "wild card"
- beware of non-unique expressions – first matching expression wins

```
casez  ({Sel,A,B})
    3'b00?: Y = 1'b1;
    3'b010: Y = 1'b1;
    3'b011: Y = 1'b0;
    // etc.
endcase
```

000 or 001 both yield Y = 1'b1

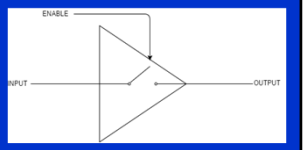
```
/* 8-to-3 Priority Encoder Using a GAL22V10 */

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
```

```
Title: 8-to-3 Priority Encoder Using GAL 22V10 (ispLever Reduced Equation Report)

 P-Terms   Fan-in  Fan-out  Type  Name (attributes)
---------  ------  -------  ----  ----------------
   4/1        4        1    Pin-  E2
   8/1        8        1    Pin-  G
   4/3        6        1    Pin-  E1
   4/4        7        1    Pin   E0
=========
  20/9          Best P-Term Total: 9
                       Total Pins: 12
                      Total Nodes: 0
             Average P-Term/Output: 2

Positive-Polarity (SoP) Equations:
E2 = (I7 # I6 # I5 # I4);
G  = (I7 # I6 # I5 # I4 # I3 # I2 # I1 # I0);
E1 = (I7 # I6 # !I5 & !I4 & I3 # !I5 & !I4 & I2);
E0 = (I7 # !I6 & I5 # !I6 & !I4 & I3 # !I6 & !I4 & !I2 & I1);

Reverse-Polarity Equations:
!E2 = (!I7 & !I6 & !I5 & !I4);
!G  = (!I7 & !I6 & !I5 & !I4 & !I3 & !I2 & !I1 & !I0);
!E1 = (!I7 & !I6 & I5 # !I7 & !I6 & I4 # !I7 & !I6 & !I3 & !I2);
!E0 = (!I7 & I6 # !I7 & !I5 & I4 # !I7 & !I5 & !I3 & I2 # !I7 & !I5 & !I3 & !I1);
```

ispLEVER operators:
AND - &, OR - #,
NOT - !, XOR - $

Tri-State Outputs

- Tri-state outputs can be assigned one of three values: logical 1, logical 0 or Hi-Z (high impedance)
- Hi-Z is a state that is not driven to any value and can be seen as an open circuit
- Example: ENABLE asserted will allow the input (logic 1 or 0) to be seen on the OUPUT (ENABLE negated will float OUTPUT to Hi-Z)



295

Tri-State Outputs

- In Verilog, an output value of **'bZ** (high- impedance or Hi-Z) assigned to an output port disables ("floats") the output
- **tri** is a *wire type* used for tri-state values
- Can use the conditional operator **? :** to implement a tri-state buffer
  - `output tri D_z;  input wire D,EN;`
  - `assign D_z = EN ? D : 1'bZ` *(ternary operator)*
    - `If EN == 1, D_z = D`
    - `If EN == 0, D_z=1'bZ (disabled)`
- Example: Create a Verilog module that implements a 4:2 priority encoder with tri-state encoded outputs (E1, E0). This design should include an active high output strobe (G) that is asserted when any input is asserted

296

Example: 4-to-2 Priority Encoder with Tri-State Outputs

```
/* 4-to-2 Priority Encoder With Tri-State Enable */

module prienc42(I, E_z, G, EN);

  input wire [3:0] I;   // input 0 - lowest priority,
                        // input 3 - highest priority
  input wire EN;        // tri-state enable control input
  output tri [1:0] E_z; // encoded tri-state enabled output
  output wire G;        // strobe "go" output (high if any input is asserted)

  reg [2:0] EG;        // EG = {E,G}

  always @ (I) begin
    casez (I)
      4'b0000:  EG = 3'b000; // No inputs active
      4'b0001:  EG = 3'b001; // Input 0 wins
      4'b001?:  EG = 3'b011; // Input 1 wins
      4'b01??:  EG = 3'b101; // Input 2 wins
      4'b1???:  EG = 3'b111; // Input 3 wins
    endcase
  end

  assign G = EG[0];
  assign E_z = EN ? EG[2:1] : 2'bzz;

endmodule
```

297

Keypad Encoders

- Another common use for encoders is to encode keypads and keyboards
- Example: Design a 10-to-4 priority encoder for encoding a BCD keypad using a 22V10
- Solution: Modify the 8-to-3 priority encoder Verilog file described previously (include tri-state output capability)

298

```
/* 10-to-4 BCD Priority Keypad Encoder */

module bcd_enc(K, EN, E_z, KS);

  input wire EN;            // Tri-state enable
  input wire [9:0] K;       // Key inputs (0 - lowest priority, 9 - highest)
  output tri [3:0] E_z;     // 4-bit encoded tri-state enabled BCD output
  output wire G;           // Key strobe (asserted high when any key pressed)

  reg [4:0] KG;

  assign G = KG[0];
  assign E_z = EN ? KG[4:1] : 4'bZZZZ;

  always @ (K) begin
    casez (K)
      10'b0000000000:  KG = 5'b00000;
      10'b0000000001:  KG = 5'b00001;
      10'b0000000017:  KG = 5'b00011;
      10'b0000000177:  KG = 5'b00101;
      10'b0000001777:  KG = 5'b00111;
      10'b0000017777:  KG = 5'b01001;
      10'b0000177777:  KG = 5'b01011;
      10'b0001777777:  KG = 5'b01101;
      10'b0017777777:  KG = 5'b01111;
      10'b0177777777:  KG = 5'b10001;
      10'b1777777777:  KG = 5'b10011;
    endcase
  end

endmodule
```

299

Clicker Quiz

300

```
/* 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
```

301

1. The highest priority input is:
   A. A
   B. B
   C. C
   D. D
   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
```

302

2. The lowest priority input is:
   A. A
   B. B
   C. C
   D. D
   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
```

303

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

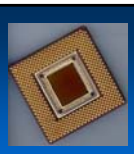
```
/* Different Priority Encoder */

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
```

304

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
```

305



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-I

Combinational Building Blocks: Multiplexers

Reading Assignment:
DDPP 4th Ed. pp. 432-440, 445-446; 5th Ed. pp. 281-289, 290-291

Learning Objectives:
- Define the function of a multiplexer and describe how it can be used as a combinational building block
- Illustrate how a multiplexer can be used to realize an arbitrary Boolean function

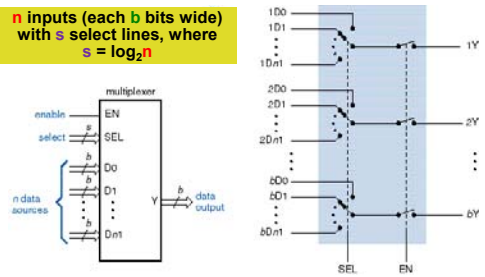
Outline
- Overview
- General multiplexer structure
- Multiplexer truth table analogy
- Multiplexer function generation
- Multiplexers in Verilog

Overview
- Definition: A ***multiplexer*** is a digital switch that uses ***s*** select lines to determine which of $n = 2^s$ inputs is connected to its output
- It is often called a ***mux*** for short
- Each of the input paths may be ***b*** bits wide
- An overall enable signal (EN) is usually provided (if EN negated, all outputs are "0")
- The equation implemented by an ***s*** select line multiplexer is the sum-of-products form of a general ***s***-variable function

$$F(X,Y) = a_0 \cdot X' \cdot Y' + a_1 \cdot X' \cdot Y + a_2 \cdot X \cdot Y' + a_3 \cdot X \cdot Y$$

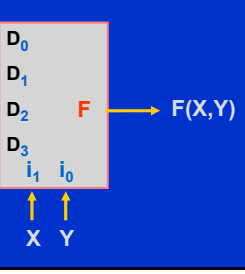
General Multiplexer Structure

n inputs (each b bits wide) with s select lines, where $s = \log_2 n$



Multiplexer Truth Table Analogy

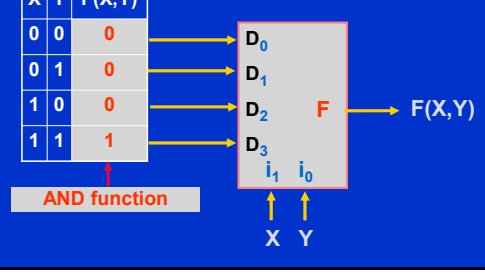
X	Y	F(X,Y)
0	0	a_0
0	1	a_1
1	0	a_2
1	1	a_3



Functional values assigned to each combination

Multiplexer Truth Table Analogy

X	Y	F(X,Y)
0	0	0
0	1	0
1	0	0
1	1	1

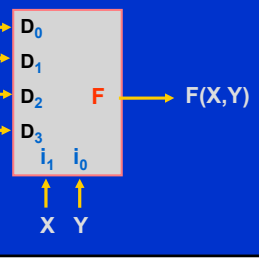


AND function

Multiplexer Truth Table Analogy

X	Y	F(X,Y)
0	0	0
0	1	1
1	0	1
1	1	1

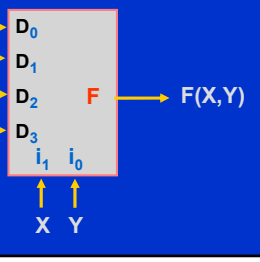
OR function



Multiplexer Truth Table Analogy

X	Y	F(X,Y)
0	0	0
0	1	1
1	0	1
1	1	0

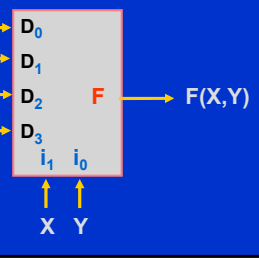
XOR function



Multiplexer Truth Table Analogy

X	Y	F(X,Y)
0	0	1
0	1	0
1	0	0
1	1	1

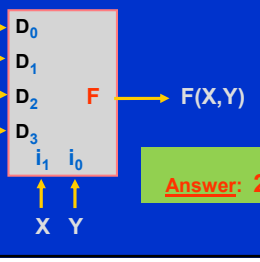
XNOR function



Multiplexer Truth Table Analogy

This is very similar to the look-up tables (LUTs) used in FPGAs

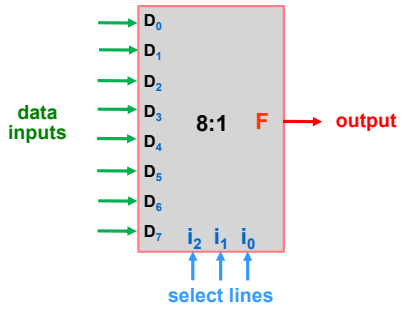
X	Y	F(X,Y)
0	0	a_0
0	1	a_1
1	0	a_2
1	1	a_3



Question: How many *different functions* of S variables are possible?

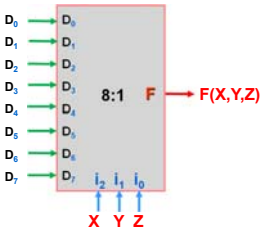
Answer: 2^{2^S}

Example: 8-to-1 (8:1) Multiplexer



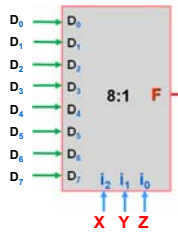
Example: Multiplexer Function Realization

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



Example: Multiplexer Function Realization

Determine the multiplexer data input values for realizing the function F(X,Y,Z) = X•Z + X′•(Y⊕Z)

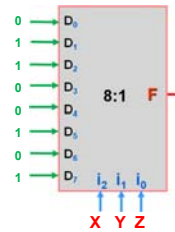


F(X,Y,Z) = X•Z + X′•(Y⊕Z)
= X•Z + X′•(Y′•Z + Y•Z′)
= X•Z + X′•Y′•Z + X′•Y•Z′

319

Example: Multiplexer Function Realization

Determine the multiplexer data input values for realizing the function F(X,Y,Z) = X•Z + X′•(Y⊕Z)



F(X,Y,Z) = X•Z + X′•(Y⊕Z)
= X•Z + X′•(Y′•Z + Y•Z′)
= X•Z + X′•Y′•Z + X′•Y•Z′

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

F(X,Y,Z) = $\Sigma_{X,Y,Z}(1,2,5,7)$

320

Multiplexers in Verilog

- Multiplexer functionality can be expressed in Verilog in several different ways:
  - using conventional sum-of-products expressions
  - using case structures
  - using if-else constructs or ternary operators
- Example: 8-to-1 X 1-bit multiplexer using a 22V10 PLD (conventional SoP)
- Example: 4-to-1 X 8-bit multiplexer using a CPLD (two advanced methods)

321

Example: 8-to-1 × 1-bit Multiplexer

```
/* 8-to-1 X 1-bit Multiplexer Using 22V10 */

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
```

322

Example: 4-to-1 × 8-bit Multiplexer – Method 1

```
/* 4-to-1 X 8-bit Multiplexer Using CPLD */

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
```

Similar to case statements, a default value for the signal should be provided in an else statement or above the if-else if block as needed

323

Example: 4-to-1 × 8-bit Multiplexer – Method 2

```
/* 4-to-1 X 8-bit Multiplexer Using CPLD */

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;
    case (S)
      2'd0: Y = A;    // d stands for decimal
      2'd1: Y = B;
      2'd2: Y = C;
      2'd3: Y = D;
      // default: Y = 8'b00000000;
    endcase
  end
endmodule
```

324

Clicker Quiz

325

```
/* 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'bZZZZZZZZZ;
  assign Y = ~S[1] & ~S[0] & A |
             ~S[1] &  S[0] & B |
              S[1] & ~S[0] & C |
              S[1] &  S[0] & D;

endmodule
```

326

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 wire [7:0] Y_z;

  wire [7:0] Y;

  assign Y_z = EN ? Y : 8'bZZZZZZZZZ;
  assign Y = ~S[1] & ~S[0] & A |
             ~S[1] &  S[0] & B |
              S[1] & ~S[0] & C |
              S[1] &  S[0] & D;

endmodule
```

327

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

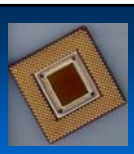
```
/* Big Multiplexer */

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 wire [7:0] Y_z;

  wire [7:0] Y;

  assign Y_z = EN ? Y : 8'bZZZZZZZZZ;
  assign Y = ~S[1] & ~S[0] & A |
             ~S[1] &  S[0] & B |
              S[1] & ~S[0] & C |
              S[1] &  S[0] & D;

endmodule
```

328

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

```
/* 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 wire [7:0] Y_z;

  wire [7:0] Y;

  assign Y_z = EN ? Y : 8'bZZZZZZZZZ;
  assign Y = ~S[1] & ~S[0] & A |
             ~S[1] &  S[0] & B |
              S[1] & ~S[0] & C |
              S[1] &  S[0] & D;

endmodule
```

329



Purdue IM:PACT* Spring 2019 Edition

*Instruction Matters: Purdue Academic Course Transformation

Introduction to Digital System Design

Module 2-J
Top Level (Hierarchical) Modules

Reading Assignment:
DDPP 4th Ed. pp. 306-308, 5th Ed. 198-201

Learning Objectives:
- Understand the need for using top level (hierarchical) modules
- Understand how top level modules are created in Verilog using structural Verilog syntax

331

Outline
- Overview
- Instantiating modules
- Example top level modules

332

Overview
- 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

333

Instantiating 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

334

Example Top Level Modules

```
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**

335

Module 2 Combinational Logic Circuits
- Learning Outcome: "An ability to analyze and design combinational logic circuits"
  - A. Combinational Circuit Analysis and Synthesis
  - B. Mapping and Minimization
  - C. Timing Hazards
  - D. XOR/XNOR Functions
  - E. Programmable Logic Devices
  - F. Hardware Description Languages
  - G. Combinational Building Blocks: Decoders
  - H. Combinational Building Blocks: Encoders and Tri-State Outputs
  - I. Combinational Building Blocks: Multiplexers
  - J. Top Level Modules

336