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

```verilog
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
Introduction to Digital System Design

Module 2-A
Combinational Circuit Analysis and Synthesis
Reading Assignment:
"DDPP 4th Ed., pp. 196-210"

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
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
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

```
X1  →  \[ f \]  \rightarrow  f (X1, X2, \ldots, Xn)
X2  →     \[ f \]  \rightarrow  \cdots
Xn  →
```

No Feedback
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.
Examples

W, X, Y'  \textit{Literals}

W \cdot X \cdot Z  \textit{Product Term}

X \cdot Y' + W \cdot Z  \textit{Sum of Products Expression}

X + Y + Z'  \textit{Sum Term}

(X + Y) \cdot (W + Z')  \textit{Product of Sums Expression}
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.
## Minterm Identification

<table>
<thead>
<tr>
<th>Row</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
<th>Minterm</th>
<th>Maxterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F(0,0,0)</td>
<td>X' · Y · Z'</td>
<td>X + Y + Z'</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>F(0,0,1)</td>
<td>X' · Y · Z</td>
<td>X + Y + Z'</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>F(0,1,0)</td>
<td>X' · Y · Z'</td>
<td>X + Y' + Z</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>F(0,1,1)</td>
<td>X' · Y · Z</td>
<td>X + Y' + Z'</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>F(1,0,0)</td>
<td>X' · Y · Z'</td>
<td>X' + Y + Z</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>F(1,0,1)</td>
<td>X · Y · Z</td>
<td>X' + Y + Z'</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>F(1,1,0)</td>
<td>X · Y · Z'</td>
<td>X' + Y' + Z</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>F(1,1,1)</td>
<td>X · Y · Z</td>
<td>X' + Y' + Z'</td>
</tr>
</tbody>
</table>

- **0 → complemented**
- **1 → true**
# Maxterm Identification

0 → true  
1 → complemented

<table>
<thead>
<tr>
<th>Row</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F</th>
<th>Minterm</th>
<th>Maxterm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>F(0,0,0)</td>
<td>X'·Y'·Z'</td>
<td>X + Y + Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>F(0,0,1)</td>
<td>X'·Y'·Z</td>
<td>X + Y + Z'</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>F(0,1,0)</td>
<td>X'·Y'·Z'</td>
<td>X + Y' + Z</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>F(0,1,1)</td>
<td>X'·Y'·Z'</td>
<td>X + Y' + Z'</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>F(1,0,0)</td>
<td>X·Y'·Z'</td>
<td>X' + Y + Z</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>F(1,0,1)</td>
<td>X·Y'·Z</td>
<td>X' + Y + Z'</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>F(1,1,0)</td>
<td>X·Y'·Z'</td>
<td>X' + Y' + Z</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>F(1,1,1)</td>
<td>X·Y·Z</td>
<td>X' + Y' + Z'</td>
</tr>
</tbody>
</table>
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”
Example

Based on the truth table, determine the following

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F(X,Y,Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

F(X,Y,Z) expressed as:

an on-set: ____________________________

an off-set: __________________________

a sum of minterms: ________________________

a product of maxterms: ________________________
Example

Based on the truth table, determine the following

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F(X,Y,Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

F(X,Y,Z) expressed as:

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

an off-set: ________________

a sum of minterms: __________________________

a product of maxterms: __________________________

\[ \sum_{X,Y,Z}(0,3,6,7) \]
Example

Based on the truth table, determine the following

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>F(X,Y,Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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: ________________________________
Example

Based on the truth table, determine the following

F(X,Y,Z) expressed as:

an on-set: \[ \sum_{X,Y,Z}(0,3,6,7) \]

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

a sum of minterms: \[ X'Y'Z' + X'YZ + XYZ' + XYZ \]

a product of maxterms: 

\[
\begin{align*}
X \cdot Y \cdot Z & = 1 \\
0 \cdot 0 \cdot 0 & = 0 \\
0 \cdot 0 \cdot 1 & = 0 \\
0 \cdot 1 \cdot 0 & = 0 \\
0 \cdot 1 \cdot 1 & = 1 \\
1 \cdot 0 \cdot 0 & = 0 \\
1 \cdot 0 \cdot 1 & = 0 \\
1 \cdot 1 \cdot 0 & = 1 \\
1 \cdot 1 \cdot 1 & = 1
\end{align*}
\]
Example

Based on the truth table, determine the following

F(X,Y,Z) expressed as:

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

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

a sum of minterms: \( \overline{X}\cdot\overline{Y}\cdot\overline{Z} + \overline{X}\cdot\overline{Y}\cdot Z + X\cdot\overline{Y}\cdot\overline{Z} + X\cdot\overline{Y}\cdot Z \)

a product of maxterms: \( (X+Y+\overline{Z})\cdot(X+Y'+Z)\cdot (X'+Y+Z)\cdot(X'+Y+Z') \)
Clicker Quiz
1. The **ON set** for a 3-input NAND gate (with inputs X, Y, and Z) is:

A. \( \sum_{X,Y,Z}(7) \)
B. \( \sum_{X,Y,Z}(0) \)
C. \( \sum_{X,Y,Z}(0,1,2,3,4,5,6) \)
D. \( \sum_{X,Y,Z}(1,2,3,4,5,6,7) \)
E. none of the above
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
3. If the function $F(X,Y,Z)$ is represented by the ON SET $\sum_{X,Y,Z}(0,3,5,6)$, then the complement of this function $F'(X,Y,Z)$ is represented by the ON SET:

A. $\sum_{X,Y,Z}(0,3,5,6)$
B. $\sum_{X,Y,Z}(1,2,4,7)$
C. $\sum_{X,Y,Z}(1,2,4,6)$
D. $\sum_{X,Y,Z}(1,3,5,7)$
E. none of the above
4. If the function \( F(X,Y,Z) \) is represented by the ON SET \( \sum_{X,Y,Z}(0,3,5,6) \), then the dual of this function \( F^D(X,Y,Z) \) is represented by the ON SET:

A. \( \sum_{X,Y,Z}(0,3,5,6) \)
B. \( \sum_{X,Y,Z}(1,2,4,7) \)
C. \( \sum_{X,Y,Z}(1,2,4,6) \)
D. \( \sum_{X,Y,Z}(1,3,5,7) \)
E. none of the above
Example - Combinational Analysis
Example - Combinational Analysis
Example - Combinational Analysis
Example - Combinational Analysis
Example - Combinational Analysis
Example - Combinational Analysis
Example - Combinational Analysis
Example - Combinational Analysis
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 $f(X,Y,Z) = X' \cdot Y' \cdot Z' + X' \cdot Y \cdot Z' + X \cdot Y' \cdot Z + X \cdot Y \cdot Z$
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 $f(X,Y,Z) = (X+Y+Z) \cdot (X+Y'+Z') \cdot (X'+Y+Z) \cdot (X'+Y'+Z)$
Writing the function implemented by this circuit “directly” yields \( f(X, Y, Z) = (((X+Y') \cdot Z) + (X' \cdot Y \cdot Z')) = X \cdot Z + Y' \cdot Z + X' \cdot Y \cdot Z' \)
The expression \( f(X,Y,Z) = X \cdot Z + Y' \cdot Z + X' \cdot Y \cdot Z' \) corresponds to a different circuit ("two-level AND-OR") for the same logic function.
Recall that an equivalent symbol can be drawn for a gate by taking the dual of the operator and complementing all of the inputs and outputs.
Step 1: Starting at the “output end”, replace the “OR” gate with an AND gate that has its inputs and outputs complemented.
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!
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)
Clicker Quiz
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
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
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
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*.
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)

$$f(N_3,N_2,N_1,N_0) = \sum N_3,N_2,N_1,N_0(1,2,3,5,7,11,13)$$
Example – Prime Number Detector
Reading Assignment:

*DDPP 4<sup>th</sup> Ed., pp. 196-210*

Learning Objectives (review):

- 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)
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
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
Module 2-B
Mapping and Minimization

Introduction to Digital System Design
Minimization

4 bit Prime Number Detector
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'):

\[
\text{Expression} \cdot X + \text{Expression} \cdot X' = \text{Expression}
\]
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
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).
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)
Karnaugh Maps

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

Example: \( f(X,Y) = X' + Y \)
Karnaugh Maps

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

- **Example:** \( f(X,Y,Z) = X' \cdot Y' + Y \cdot Z \)
Karnaugh Maps

- Example: \( f(X,Y,Z) = X' \cdot Y' + Y \cdot Z \)
Karnaugh Maps

- Example: \( f(X,Y,Z) = X' \cdot Y' + Y \cdot Z \)
Karnaugh Maps

- Example: \( f(X, Y, Z) = X' \cdot Y' + Y \cdot Z \)
Karnaugh Maps

- Drawing for a 4-variable K-map

```
0 4 12 8
1 5 13 9
3 7 15 11
2 6 14 10
```

```
X' X X' W' W Z' Z
Y' 1 5 13 9
Y
3 7 15 11
2 6 14 10
```
Karnaugh Maps

Example: $f(W,X,Y,Z) = X' \cdot Z' + W \cdot Z + W' \cdot X$
Karnaugh Maps

- **Example**: \( f(W,X,Y,Z) = X' \cdot Z' + W \cdot Z + W' \cdot X \)
Karnaugh Maps

- **Example:** \( f(W,X,Y,Z) = X' \cdot Z' + W \cdot Z + W' \cdot X \)
Karnaugh Maps

- **Example:** \( f(W,X,Y,Z) = X' \cdot Z' + W \cdot Z + W' \cdot X \)

<table>
<thead>
<tr>
<th></th>
<th>W'</th>
<th></th>
<th></th>
<th>W</th>
<th></th>
<th></th>
<th></th>
<th>Z'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>12</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>13</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>1</td>
<td>14</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

|   |   |   |   |   | X |   |   | X' |
|---|---|---|---|---|---|---|----|
| 0 | 1 | 4 | 1 |   | 8 | 1 |    |
| 1 | 5 | 1 | 13| 1 | 9 | 1 |    |
| 2 | 6 | 1 | 14| 10| 1 | 1 |    |
|   |   |   |   |   |   |   |    |
**Karnaugh Maps**

- **Example**: \( f(W,X,Y,Z) = X' \cdot Z' + W \cdot Z + W' \cdot X \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>W'</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>W</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y'</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z'</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![Karnaugh Map Diagram](#)
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*

\[
\text{COST} = \text{No. of Gate Inputs} + \text{No. of Gate Outputs}
\]
Minimization Procedure

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

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

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

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

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

```
<table>
<thead>
<tr>
<th></th>
<th>W'</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

- **One Possibility**
Minimization Procedure

- **STEP 4:** Write a minimal, non-redundant sum-of-products expression

\[ f(W, X, Y, Z) = W' \cdot X \cdot Z \]

\[ + \ W \cdot X \cdot Z' \]

\[ + \ W \cdot Y \cdot Z' \]

\[ + \ X \cdot Y \cdot Z \]
One possible circuit implementation (AND-OR):

**COST** is 16 inputs + 5 outputs = 21
EQUIVALENT circuit implementation, obtained through graphical application of DeMorgan’s Law

Note: AND-OR ≡ NAND-NAND

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

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

[Diagram showing a truth table with cells marked for W, W', X, X', Z, Z', Y, and Y'.]
Minimization Procedure

- REVISIT STEP 4: Write a minimal, non-redundant sum-of-products expression

\[
\begin{align*}
f(W,X,Y,Z) &= W' \cdot X \cdot Z \\
&+ W \cdot X \cdot Z' \\
&+ W \cdot Y \cdot Z' \\
&+ W \cdot X \cdot Y
\end{align*}
\]
Clicker Quiz
1. The number of prime implicants is:

<table>
<thead>
<tr>
<th></th>
<th>X'</th>
<th>X</th>
<th>X'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y'</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th></th>
<th>w'</th>
<th>w</th>
<th>Z'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y'</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>w'</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>w</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Y'</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>X'</strong></td>
<td><strong>X</strong></td>
<td><strong>X'</strong></td>
<td><strong>X</strong></td>
<td><strong>X'</strong></td>
</tr>
</tbody>
</table>

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

4 bit Prime Number Detector

SoP form

Minimized
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
Minimization Procedure

\[ f(W, X, Y, Z) = W' \cdot X \cdot Z + W \cdot X \cdot Z' + W \cdot Y \cdot Z' + W \cdot X \cdot Y \]
Minimization: Clicker Quiz Example

<table>
<thead>
<tr>
<th></th>
<th>W'</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Essential**
- **Non-Essential**
Minimization: Another Example

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

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

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

```plaintext
X'  X  X'  Y'  Y  Z'  Z  W'  W
0  4  12  0  12  8  1  1
1  5  13  1  13  9  0  0
1  6  14  0  14  10  1  1
1  3  11  1  11  1  1  1
```
Minimization Procedure

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

Exercise: Find a minimal sum-of-products expression for the function mapped below.
Minimization Procedure

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

\[
\begin{array}{ccc}
X' & X & X' \\
Y' & W' & W \\
Y & 0 & 1 \\
Z' & Z & Z' \\
0 & 1 & 0 \\
3 & 1 & 0 \\
2 & 0 & 1 \\
4 & 0 & 0 \\
5 & 1 & 0 \\
6 & 1 & 1 \\
7 & 1 & 0 \\
8 & 1 & 0 \\
9 & 1 & 1 \\
10 & 1 & 0 \\
11 & 1 & 1 \\
12 & 1 & 0 \\
13 & 1 & 1 \\
14 & 1 & 0 \\
15 & 1 & 1 \\
\end{array}
\]

- essential prime implicants
Minimization Procedure

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

![Function Map]

- **non-essential prime implicant needed to cover function**
Minimization Procedure

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

\[
\begin{align*}
f(W, X, Y, Z) &= W \cdot Y' + X' \cdot Y' + W \cdot X' \cdot Z' + W' \cdot X \cdot Y' + Y' \cdot Z
\end{align*}
\]
Minimization: Product-of-Sums

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

<table>
<thead>
<tr>
<th></th>
<th>X'</th>
<th>X</th>
<th>Y</th>
<th>Y'</th>
<th>Z</th>
<th>Z'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>12</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>13</td>
<td>9</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>15</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Group zeroes to get a minimum sum-of-products expression for $f'$.
Minimization Procedure

Question: How could a minimal \textit{product-of-sums expression} for this function be found?

Group zeroes to get a minimum sum-of-products expression for $f'$

Find essential prime implicants
Minimization Procedure

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

Group zeroes to get a minimum sum-of-products expression for $f'$

Find essential prime implicants
Minimization Procedure

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

Group zeroes to get a minimum sum-of-products expression for \( f' \)

Find essential prime implicants
Minimization Procedure

Question: How could a minimal \textit{product-of-sums expression} for this function be found?

Group zeroes to get a minimum sum-of-products expression for $f'$

Find essential prime implicants

Function is completely covered
Minimization Procedure

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

\[
f' = W' \cdot Y + X' \cdot Y + W' \cdot X \cdot Z'
\]

Apply DeMorgan’s Law

\[
f = (W' + Y') \cdot (X + Y') \cdot (W + X' + Z)
\]
One possible circuit implementation (OR-AND):

**COST** is 10 inputs + 4 outputs = 14
EQUIVALENT circuit implementation, obtained through graphical application of DeMorgan’s Law

Note: OR-AND \equiv NOR-NOR

COST is 10 inputs + 4 outputs = 14 (same)
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 \( \Rightarrow \) implement \( F' \)
Solution to (a)

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

<table>
<thead>
<tr>
<th></th>
<th>X'</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y'</td>
<td></td>
<td>Y'</td>
</tr>
</tbody>
</table>
Solution to (a)

\[
\begin{array}{c|c|c|c}
Z' & X' & X \\
\hline
1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 \\
\end{array}
\]

\[F(X,Y,Z) = X' \cdot Y\]
Solution to (a)

\[ F(X,Y,Z) = X' \cdot Y + X' \cdot Z' \]
Solution to (a)

\[ F(X,Y,Z) = X' \cdot Y + X' \cdot Z' + Y \cdot Z' \]
Solution to (a)

\[
\begin{array}{c|ccc|c}
\text{X'} & \text{X} & \text{Z'} & \text{Z} & \text{Y'} \\
\hline
1 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
F(X,Y,Z) = X' \cdot Y + X' \cdot Z' + Y \cdot Z'
\]

![Logic circuit diagram](image-url)
Solution to (b)

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

<table>
<thead>
<tr>
<th></th>
<th>X'</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Z'</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y'</td>
<td>Y</td>
<td>Y'</td>
</tr>
<tr>
<td></td>
<td>Y</td>
<td>Y'</td>
</tr>
</tbody>
</table>
Solution to (b)

<table>
<thead>
<tr>
<th></th>
<th>X'</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ F'(X,Y,Z) = X \cdot Y' \]
Solution to (b)

\[ F'(X,Y,Z) = X \cdot Y' + X \cdot Z \]
Solution to (b)

<table>
<thead>
<tr>
<th></th>
<th>X'</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y'</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Y</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y'</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ F'(X,Y,Z) = X \cdot Y' + X \cdot Z + Y' \cdot Z \]
Solution to (b)

F’(X,Y,Z) = X•Y’ + X•Z + Y’•Z
“Conversion” Example

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

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

\[ \implies F'(X, Y, Z) = \underline{\hspace{10cm}} \implies \text{Map } \underline{\hspace{6cm}} \]

\[ \implies F(X, Y, Z) = \underline{\hspace{10cm}} \implies F'(X, Y, Z) \text{ in minimal POS form} \]

\[ = \underline{\hspace{12cm}} \]
“Conversion” Example

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

\[ F(X, Y, Z) = (X + Y) \cdot (X' + Y + Z) \cdot (X + Y + Z') \]

\[ \Rightarrow F'(X, Y, Z) = X' \cdot Y' + X \cdot Y' \cdot Z' + X' \cdot Y' \cdot Z \Rightarrow \text{Map zeroes} \]

<table>
<thead>
<tr>
<th></th>
<th>X’</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z’</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Y’</td>
<td>Y</td>
<td>Y’</td>
</tr>
</tbody>
</table>

\[ F(X, Y, Z) = \square \Rightarrow \]

\[ F'(X, Y, Z) \text{ in minimal POS form} = \square \]
“Conversion” Example

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

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

$$\Rightarrow F'(X,Y,Z) = X' \cdot Y' + X \cdot Y' \cdot Z' + X' \cdot Y' \cdot Z \Rightarrow \text{Map zeroes}$$

F(X,Y,Z) = Y + X\cdot Z \Rightarrow F'(X,Y,Z) \text{ in minimal POS form} = Y' \cdot (X' + Z')
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 $(1010 - 1111)_2$.
Incompletely Specified Functions

Application: Determine a logic function that will be “1” if the BCD digit input satisfies the following inequality: \(1 < N_{10} < 9\)

\[ F = \sum_{W,X,Y,Z} (2,3,4,5,6,7,8) + d(10,11,12,13,14,15) \]
**BCD Inequality Detector Example**

<table>
<thead>
<tr>
<th>$N_{10}$</th>
<th>W X Y Z</th>
<th>$F(W,X,Y,Z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>0</td>
</tr>
</tbody>
</table>
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: \( f(W, X, Y, Z) = X + Y + W \cdot Z' \)
Cost: 5 gate inputs + 2 gate outputs = 7
BCD Inequality Detector Example: POS

Minimum PS:

\[ f'(W,X,Y,Z) = \]

\[ W \cdot Z + W' \cdot X' \cdot Y' \]

\[ \rightarrow f (W,X,Y,Z) = \]

\[ (W' + Z') \cdot (W + X + Y) \]

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

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

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.

\[ f(W, X, Y, Z) = W + X \cdot Y' \]
\[ f(W, X, Y, Z) = W' \cdot (X' + Y) \]
Cost: 4 gate inputs + 2 gate outputs = 6 cost units
Clicker Quiz
1. The cost of a minimal sum of products realization of this function (assuming both true and complemented variables are available) would be:

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

A. 9    B. 10    C. 11    D. 12    E. none of the above
3. Assuming the availability of \textbf{only true} input variables, the \textbf{fewest number of 2-input NAND gates} that are needed to realize this function is: 

\begin{itemize}
  \item [A.] 6
  \item [B.] 7
  \item [C.] 8
  \item [D.] 9
  \item [E.] none of the above
\end{itemize}
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
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
Reading Assignment:
DDPP 4th Ed., pp. 224-229

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:

<table>
<thead>
<tr>
<th></th>
<th>Z'</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>X'</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ f(X, Y, Z) = X \cdot Z' + Y \cdot Z \]

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

\[ f(X, Y, Z) = X \cdot Z' + Y \cdot Z + X \cdot Y \]

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
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. Hi-Z
E. none of the above
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. Hi-Z
E. none of the above
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. Hi-Z
E. none of the above
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
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
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
Module 2-D
XOR/XNOR Functions

Introduction to
Digital System Design
Reading Assignment:
*DDPP* 4th Ed., pp. 447-448

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

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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>$X ⊕ Y$</th>
<th>$(X ⊕ Y)'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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
XOR K-Map

K-map of 2-variable XOR function

\[ X \oplus Y = X' \cdot Y + X \cdot Y' \]

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

- cascade XOR circuit

The output of an n-variable XOR function is 1 if an odd number of inputs are 1

The output of an n-variable XNOR function is 1 if an even number of inputs are 1

Realization of an n-variable XOR or XNOR function will require $2^{n-1}$ P-terms
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.
Example – “Diagonal” K-map
Example – “Diagonal” K-map

- Minimize function to the extent possible

F(W,X,Y,Z) = 
W'X'Y'Z' + W'X'YZ' + W'X'YZ + WX'YZ'
Example – “Diagonal” K-map

- Factor out XOR/XNOR expressions

\[ F(W, X, Y, Z) = \]
\[ W' \cdot X' \cdot Y' \cdot Z' + W' \cdot X \cdot Y' \cdot Z \\
+ W \cdot X \cdot Y \cdot Z + W \cdot X' \cdot Y \cdot Z' \]
\[ = X' \cdot Z' \cdot (W' \cdot Y' + W \cdot Y) \\
+ X \cdot Z \cdot (W' \cdot Y' + W \cdot Y) \]
Example – “Diagonal” K-map

- Factor out XOR/XNOR expressions

\[
F(W,X,Y,Z) = W'X'Y'Z' + W'XY'Z' + WXY'Z + W'X'YZ' \\
= X'Z' \cdot (W'Y' + WY) + XZ \cdot (W'Y' + WY) \\
= (X'Z' + XZ) \cdot (W'Y' + WY)
\]
Example – “Diagonal” K-map

- Write function in terms of XOR/XNOR operators

\[
F(W,X,Y,Z) = W' \cdot X' \cdot Y' \cdot Z' + W' \cdot X \cdot Y' \cdot Z + W \cdot X \cdot Y \cdot Z + W \cdot X' \cdot Y \cdot Z' \\
= X' \cdot Z' \cdot (W' \cdot Y' + W \cdot Y) + X \cdot Z \cdot (W' \cdot Y' + W \cdot Y) \\
= (X' \cdot Z' + X \cdot Z) \cdot (W' \cdot Y' + W \cdot Y) \\
= (X \oplus Z)' \cdot (W \oplus Y)'
\]
Example – “Diagonal” K-map

- Realize using XOR/XNOR gates

\[
\begin{align*}
X & \quad Z \\
W & \quad Y \\
X & \quad Z \\
W & \quad Y
\end{align*}
\]

\[
F(W,X,Y,Z) = \text{COST} = 6 \text{ inputs} + 3 \text{ outputs} = 9
\]
Example – “Diagonal” K-map

- Compare with minimal SoP realization

\[
\text{COST} = 20 \text{ inputs} + 5 \text{ outputs} = 25
\]
Example – “X”-map
Example – “X”-map

- Minimize function to the extent possible

F(W,X,Y,Z) = X’\cdot Z’ + X\cdot Z
Example – “X”-map

Write function in terms of XOR/XNOR operators

\[ F(W,X,Y,Z) = X' \cdot Z' + X \cdot Z \]
\[ = (X \oplus Z)' \]
Example – “X”-map

- Compare costs

F(W,X,Y,Z) =
X′•Z′ + X•Z  Cost=9
= (X ⊕ Z)′  Cost=3
Clicker Quiz
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
2. The **ON set** of the function realized by this circuit is:

A. \( \sum_{X,Y}(0,2) \)
B. \( \sum_{X,Y}(0,3) \)
C. \( \sum_{X,Y}(1,2) \)
D. \( \sum_{X,Y}(1,3) \)
E. none of the above
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
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
5. The following is **NOT** an equivalent symbol for an XOR gate:

A. ![Diagram A]

B. ![Diagram B]

C. ![Diagram C]

D. ![Diagram D]

E. none of the above
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
Introduction to Digital System Design

Module 2-E
Programmable Logic Devices
Reading Assignment:

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
Outline

- Overview
- Programmable Logic Arrays (PLAs)
- Programmable Array Logic (PALs)
- Generic Array Logic (GALs)
- Complex PLDs
- Field Programmable Gate Arrays (FPGAs)
- Summary
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 \( n \times m \) PLA with \( p \) product terms.
Programmable Logic Array

- 4 x 3 PLA with 6 product terms

Potential connections indicated by “X”
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).
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.
Programmable Logic Array

- Compact view of 4 x 3 PLA with 6 P-terms
Programmable Logic Array

- 4 x 3 PLA programmed to implement three logic equations

\[ I_1 \cdot I_2 + I_1' \cdot I_2' \cdot I_3' \cdot I_4' \]
\[ I_1 \cdot I_3' + I_1' \cdot I_3 \cdot I_4 + I_2 \]
\[ I_1 \cdot I_2 + I_1 \cdot I_3' + I_1' \cdot I_2' \cdot I_4' \]
Programmable Array Logic

- A special case of PLA is the *programmable array logic* (PAL) device.
- Unlike a PLA, a PAL device has a *fixed OR array* (i.e., AND gates cannot 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 *output-only*, output *cascaded* to another function input, or feedback to create a sequential circuit.

![An example of a PAL](image)
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.
GAL Combinational Macrocell

“fuse” matrix

P-term router
(1:2 demux)

I/O pin
dedicated inputs
dedicated output enable (OE) pin

tri-state enable selector (4:1 mux)

output polarity control

product terms
(P-terms)

INPUT PIN 0

INPUT PIN 9

...
GAL22V10 Block Diagram

number of AND array inputs

number of macrocells and associated I/O pins

200
GAL22V10 AND Array ("Fuse Matrix")
GAL22V10 Output Logic Macrocell ("OLMC")

Number of P-terms per OLMC ranges from 8 to 16
GAL22V10 Output Logic Macrocell ("OLMC")

Single P-term per OLMC dedicated to tri-state buffer enable
All OLMC edge-triggered D flip-flops utilize common clock (CLK), asynchronous reset (AR), and asynchronous preset (SP) signals.
GAL22V10 Output Logic Macrocell ("OLMC")

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

Note: Tri-state buffer is turned off to use I/O pin as an input
GAL22V10 Pinout

- Clock or data input
- Data inputs
- Macrocell I/O pins (inputs or outputs)
- Data input
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
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
ispMACH 4000ZE Generic Logic Block

To GRP

1+OE

1+OE

1+OE

1+OE

1+OE

1+OE

To ORP

To Product Term
Output Enable Sharing.
Also, To Input Enable of
Power Guard on I/Os
in the block.

36 Inputs
from GRP

CLK0
CLK1
CLK2
CLK3

Clock
Generator

16 MC Feedback Signals

16 Macrocells

Logic Allocator

AND Array
36 Inputs,
83 Product Terms
ispMACH 4000ZE 36-Input AND Array

Note:
⊗ Indicates programmable fuse.
ispMACH 4000ZE Generic Logic Block

To GRP

Clock Generator

16 Macrocells

Logic Allocator

AND Array, 36 Inputs, 83 Product Terms

36 Inputs from GRP

To ORP

To Product Term Output Enable Sharing. Also, To Input Enable of Power Guard on I/Os in the block.
ispMACH 4000ZE Logic Allocator (Slice)
ispMACH 4000ZE Generic Logic Block

To GRP

To ORP

To Product Term Output Enable Sharing. Also, To Input Enable of Power Guard on I/Os in the block.
ispMACH 4000ZE Macrocell
ispMACH 4000ZE I/O Cell
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)
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
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
Introduction to Digital System Design

Module 2-F
Hardware Description Languages
Reading Assignment:

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.
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 ECE270.
- 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’bxxxxx where n is the bit-width of the signal and x is 0 or 1
Verilog wire type

- **wire** is a basic data type in Verilog
- Similar to an actual wire, these variables cannot store logic value and are used to connect signals between inputs, outputs and logic elements such as gates
- **wire** is used to model Combinational Logic
- **wire** can take on four basic values
  - 0 – logical zero
  - 1 – logical one
  - X – Unknown value
  - Z – High-Impedance state
Verilog Bitwise Operators

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

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

ispLEVER reports use different notation for some operators

<table>
<thead>
<tr>
<th>Logical operation</th>
<th>In Verilog</th>
<th>In ispLEVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>~</td>
<td>!</td>
</tr>
<tr>
<td>XOR</td>
<td>^</td>
<td>$</td>
</tr>
</tbody>
</table>
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.

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

<table>
<thead>
<tr>
<th>3-bit wires A,B,X and Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A is assigned the constant 3-bit value of 110</td>
</tr>
<tr>
<td>B is assigned the constant 3-bit value of 101</td>
</tr>
<tr>
<td>wire X is assigned the value of A bitwise AND-ed with B i.e. 100</td>
</tr>
<tr>
<td>wire Y is assigned the value of A bitwise OR-ed with B i.e. 111</td>
</tr>
</tbody>
</table>
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
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

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

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
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] on pin 7, Y[0] on pin 8
The concatenation operator { } concatenates multiple wires to make a bit vector
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
/* 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
// are as intended

endmodule

Note: Explicit pin declarations can be omitted and automatically assigned
/* 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
/* 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
Verilog “reg” data types

• Similar to wire, but reg can be used to store information like registers.

• Unlike wire, “reg” can be used to model both Combinational and Sequential logic.

• For behavioral code using an always block, the output must be type reg.

• For dataflow code with assign statements, the outputs must be of type wire.

• Examples:

  reg myvar;    // one bit variable called myvar
  reg [7:0] myvec; // 8 bit variable called myvec
always block in Verilog

**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
always block in Verilog

Example always block:

always @ (A,B,C) begin
  ... 
end

always @ (posedge CLK) begin
  ... 
end

always @ (*) begin
  ... 
end

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

All statements will be evaluated on the positive (rising) edge of CLK signal.

All statements will be evaluated whenever any input signal in the always block changes.
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
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;
        3'b001: Y = 1'b1;
        3'b010: Y = 1'b1;
        // etc. for all input combinations
    endcase
endmodule

Y must be declared as reg type to be an output of an always block
See Wakerly for more information

@ (Sel, A, B) is a sensitivity list. For combinational logic, list all inputs
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;
        3'b001: Y = 1'b1;
        3'b010: Y = 1'b1;
        // etc. for all input combinations
        default: Y = 1'b0; // or use a default case
    endcase
endmodule

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
/* 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'b00010;
            4'b0110: abcdf = 5'b00100;
            4'b0111: abcdf = 5'b00010;
            4'b1000: abcdf = 5'b10000;
            4'b1001: abcdf = 5'b01000;
            4'b1010: abcdf = 5'b00010;
            4'b1011: abcdf = 5'b00001;
            4'b1100: abcdf = 5'b00010;
            4'b1101: abcdf = 5'b10000;
            4'b1110: abcdf = 5'b00100;
            4'b1111: abcdf = 5'b10000;
        end
        assign {A,B,C,D,F} = abcdf;
    endmodule

Example Verilog Module #2

Compares each case against a concatenated 4-bit vector with E at bit position 3, R at position 2, S at position 1 and T at position 0

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

assign A = abcdf[4], B = abcdf[3], etc.
Example Verilog module #3A

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

    // bitwise &, |, ^ can be used as
    // reduction operators on a vector
    assign X = ^I;

    // you could also write this as
endmodule

Each XOR gate increases P-terms by a factor of 2

Equation requires 8 P-terms → can be realized on any 22V10 macrocell (any I/O pin)
Example Verilog module #3B

```verilog
/* 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
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="18,29,23"*/;
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 PLD (which doubles the propagation delay)
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.

E.g. `module_name instance_name (signal_list);` will instantiate a module of type `module_name` called `instance_name`. `signal_list` corresponds to the inputs and outputs, also called the port list.

E.g. `and AND2 (XY, X, Y);` will instantiate an AND gate with inputs X and Y with output XY
E.g. `xor OR2 (X_Y,X,Y);` instantiates 2 input XOR gate
### Built in Primitives in Verilog

- **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 [section 5.7](#) in the text.
Structural code in Verilog

- Example to show 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
Clicker Quiz
1. Which of the following is not a valid Verilog identifier?

A. X2
B. 2X
C. XY
D. _XY
E. none of the above
2. Which of the following specifies a range of bits within a bit vector $X$ in Verilog?

A. $X_3..X_1$
B. $X(3:1)$
C. $[3:1]$
D. $X[3:1]$
E. none of the above
3. For input or output port declarations, which of the following statements is not true?
   A. "synthesis loc" declarations associate the device’s physical pins with symbolic port names
   B. pin numbers are optional
   C. if pin numbers are not specified, the pin numbers are assigned by the “fitter” program based on the PLD characteristics
   D. the pin may be declared active high or active low
   E. none of the above
4. The **order** in which different **assign expressions** are placed in the body of a Verilog module **does not matter**.
   
   A. true
   B. false
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)
K-Map of “Grade Distribution”

```plaintext
<table>
<thead>
<tr>
<th>S'</th>
<th>E'</th>
<th>E</th>
<th>T'</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>A</td>
<td>A</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Note: The K-Map represents a simplified version of the grade distribution with different letter grades assigned to different quadrants.
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

A = S’\cdot R + T \cdot R

COST = 6 inputs + 3 outputs = 9
**Working K-Map for “A” – PoS**

<table>
<thead>
<tr>
<th></th>
<th>E'</th>
<th></th>
<th>E</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S'</td>
<td>0</td>
<td>4</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>9</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7</td>
<td>15</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
<td>14</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**A' = R' + S·T'**

**A = R • (S' + T)**

COST = 4 inputs + 2 outputs = 6

Cheaper than SoP
Working K-Map for “B” – SoP

\[ B = E \cdot S' \cdot R' + R' \cdot S' \cdot T' \]

\[ \text{COST} = 8 \text{ inputs} + 3 \text{ outputs} = 11 \]
Working K-Map for “B” – PoS

B' = S + R + E'\cdot T
B = S'\cdot R'\cdot (E+T')

COST = 5 inputs + 2 outputs = 7

Cheaper than SoP
Working K-Map for “C” – SoP

\[ C = S \cdot T' \]

COST = 3 inputs + 2 outputs = 5
Working K-Map for “C” – PoS

\[ C' = S' + T \]
\[ C = S \cdot T' \]

COST = 2 inputs + 1 output = 3

Cheaper than SoP
Working K-Map for “D” – SoP

\[ D = E' \cdot T \cdot R' \]

COST = 4 inputs + 2 outputs = 6
Working K-Map for “D” – PoS

D = E' \cdot T \cdot R'

COST = 3 inputs + 1 output = 4

Cheaper than SoP
### Working K-Map for “F” - SoP

<table>
<thead>
<tr>
<th></th>
<th>E'</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R'</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Logic Function

\[ F = E \cdot S \cdot R' \cdot T \]

### Cost

- 5 inputs
- 2 outputs
- Total cost = 7
Working K-Map for “F” - PoS

F' = E' + S' + R + T'

F = E \cdot S \cdot R' \cdot T

COST = 4 inputs + 1 output = 5

Cheaper than SoP
SSI “Final Answer”…

2/3 - 7404

1/2 - 7402

2/3 - 7410

1/4 - 7400

1/4 - 7402

1/3 - 7410

1/6 - 7404

1/2 - 7400

1/4 - 7402

3/4 - 7400

5/6 - 7404

1 - 7402

1 - 7410

4 integrated circuits total 272
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'b10000;
    4'b1001: ABCDF = 5'b10000;
    4'b1010: ABCDF = 5'b00100;
    4'b1011: ABCDF = 5'b00010;
    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
Module 2-G
Combinational Building Blocks:
Decoders / Demultiplexers
Reading Assignment:
*DDPP* 4th Ed., pp. 384-390, 403-409

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
Outline

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

![Decoder Diagram]

*Figure 1: Diagram of a decoder circuit showing input code word, enable inputs, map, and output code word.*
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*).
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
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.
Example: 2-to-4 (2:4) Decoder
Example: 2-to-4 (2:4) Decoder
Example: 2-to-4 (2:4) Decoder
Example: 2-to-4 (2:4) Decoder
Example: 2-to-4 (2:4) Decoder
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).
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

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

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

Here, output of NAND gate is ACTIVE HIGH

ON set = \(\sum_{x,y,z}(1,2,4,7)\)

\[ F(X,Y,Z) = X \oplus Y \oplus Z \]
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 = $\Pi_{X,Y,Z}(0,2,7)$

$F(X,Y,Z) = X'\cdot Z + X\cdot Y' + X\cdot Z'$
Decoders in Verilog

/* 3:8 decoder Active-Low o/p implemented using 22V10 */

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


endmodule
Decoders/Demultiplexers in Verilog

/* 3:8 decoder / 1:8 demultiplexer Active-High o/p implemented using 22V10 */

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


endmodule
Clicker Quiz
1. The **OFF set** realized by this decoder-based circuit is:

A. $\Pi_{X,Y,Z}(0,2,5,7)$

B. $\Pi_{X,Y,Z}(1,3,4,6)$

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

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

E. none of the above
2. The **ON set** realized by this decoder-based circuit is:

A. \( \Sigma_{X,Y,Z}(0,2,5,7) \)

B. \( \Sigma_{X,Y,Z}(1,3,4,6) \)

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

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

E. none of the above
Special Purpose Decoders

- A seven-segment decoder has 4-bit BCD or hexadecimal data as its input code and “seven-segment code” as its output code.

\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array} \]

\[ \begin{array}{cccc}
\alpha & \beta & \gamma & \delta \\
\epsilon & \zeta & \eta & \theta \\
\varphi & \chi & \psi & \omega \\
\end{array} \]
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
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
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
Example: Hexadecimal 7-Segment Decoder

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
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'b1111001;
    4'b1110:  SEG7 = 7'b1001111;
    4'b1111:  SEG7 = 7'b1000111;
  endcase
end

assign {A,B,C,D,E,F,G} = SEG7;
endmodule
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
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
assign {A,B,C,D,E,F,G} = SEG7;
endmodule
Module 2-H
Combinational Building Blocks:
Encoders and Tri-State Outputs
Reading Assignment:

*DDPP* 4th Ed., pp. 408-412, 430-432

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
Outline

- Overview
- Priority Encoders
- Tri-State Outputs
- Keypad Encoders
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* (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 (GS) to indicate if any input has been asserted
Verilog casez construct

- use ? as wild card
- beware of non-unique expressions. 1\textsuperscript{st} 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 give Y = 1'b1
module pri_enc(I, E, GS);

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

    reg [3:0] EGS;

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

        endcase
    end

    assign {E,GS} = EGS;

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

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

<table>
<thead>
<tr>
<th>P-Terms</th>
<th>Fan-in</th>
<th>Fan-out</th>
<th>Type</th>
<th>Name (attributes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4/1</td>
<td>4</td>
<td>1</td>
<td>Pin-</td>
<td>E_2_</td>
</tr>
<tr>
<td>8/1</td>
<td>8</td>
<td>1</td>
<td>Pin-</td>
<td>GS</td>
</tr>
<tr>
<td>4/3</td>
<td>6</td>
<td>1</td>
<td>Pin-</td>
<td>E_1_</td>
</tr>
<tr>
<td>4/4</td>
<td>7</td>
<td>1</td>
<td>Pin</td>
<td>E_0_</td>
</tr>
</tbody>
</table>

20/9  Best P-Term Total: 9
      Total Pins: 12
      Total Nodes: 0
      Average P-Term/Output: 2

Positive-Polarity (SoP) Equations:

E_2_ = (I_7_ # I_6_ # I_5_ # I_4_);

GS = (I_7_ # I_6_ # I_5_ # I_4_ # I_3_ # I_2_ # I_1_ # I_0_);

E_1_ = (I_7_ # I_6_ # !I_5_ & !I_4_ & I_3_ # !I_5_ & !I_4_ & I_2_);

E_0_ = (I_7_ # !I_6_ & I_5_ # !I_6_ & !I_4_ & I_3_ # !I_6_ & !I_4_ & !I_2_ & I_1_);

Reverse-Polarity Equations:

!E_2_ = (!I_7_ & !I_6_ & !I_5_ & !I_4_);

!GS = (!I_7_ & !I_6_ & !I_5_ & !I_4_ & !I_3_ & !I_2_ & !I_1_ & !I_0_);

!E_1_ = (!I_7_ & !I_6_ & !I_5_ # !I_7_ & !I_6_ & !I_4_ # !I_7_ & !I_6_ & !I_3_ & !I_2_);

!E_0_ = (!I_7_ & !I_6_ # !I_7_ & !I_5_ & !I_4_ # !I_7_ & !I_5_ & !I_3_ & !I_2_ & !I_7_ & !I_5_ &
Tri-State Outputs

- Tri-state outputs can be assigned 3 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
- e.g. ENABLE asserted high will allow the input (logic 1 or 0) to be seen on the OUTPUT. ENABLE deasserted will pull OUTPUT to Hi-Z
Tri-State Outputs

- In Verilog, an output value of 'bZ (High-Impedance or Hi-Z) assigned to an output port disables the output
- “tri” is the datatype used for tri-state variables
- 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
  - If EN == 1, D_z = D. If EN == 0, D_z=1'bZ (disabled)
- **Example**: Create an 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 (GS) that is asserted when any input is asserted
Example: 4-to-2 Priority Encoder with Tri-State Outputs
/* 4-to-2 Priority Encoder With Tri-State Enable */

module prienc42(I, E_z, GS, 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 GS;       // Strobe output (high if any input is asserted)
  reg [2:0] EGS;        // EGS = {E,GS}

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

  assign GS = EGS[0];
  assign E_z = EN ? EGS[2:1] : 2'bZ;
endmodule
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)
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; // E3 E2 E1 E0 - encoded BCD output
output wire KS;       // Key strobe (high when any key pressed)

reg [4:0] EKS;

assign KS = EKS[0];
assign E_z = EN ? EKS[4:1] : 4'bZZZZ;

always @ (K) begin
  casez (K)
    10'b0000000000: EKS = 5'b00000;
    10'b0000000001: EKS = 5'b00001;
    10'b000000001?: EKS = 5'b00011;
    10'b00000001???: EKS = 5'b00101;
    10'b0000001????: EKS = 5'b00111;
    10'b000001?????: EKS = 5'b01001;
    10'b00001??????: EKS = 5'b01011;
    10'b0001???????: EKS = 5'b01101;
    10'b001????????: EKS = 5'b01111;
    10'b01?????????: EKS = 5'b10001;
    10'b1??????????: EKS = 5'b10011;
  endcase
end

endmodule
Clicker Quiz
module diff_pri(A, B, C, D, E, GS);

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

    reg [2:0] EGS;

    assign E = EGS[2:1];
    assign GS = EGS[0];

    always @ (A, B, C, D) begin
        casez ({A,B,C,D})
            4'b0000:  EGS = 3'b000;
            4'b0001:  EGS = 3'b111;
            4'b001?:  EGS = 3'b101;
            4'b01??:  EGS = 3'b011;
            4'b1???:  EGS = 3'b001;
        endcase
    end
endmodule
1. The highest priority input is:

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

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

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

reg [2:0] EGS;

assign E = EGS[2:1];
assign GS = EGS[0];

always @ (A, B, C, D) begin
  casez ({A,B,C,D})
    4'b0000:  EGS = 3'b000;
    4'b0001:  EGS = 3'b111;
    4'b001?:  EGS = 3'b101;
    4'b01??:  EGS = 3'b011;
    4'b1???:  EGS = 3'b001;
  endcase
end
endmodule
```
2. The **lowest priority input** is:

A. A  
B. B  
C. C  
D. D  
E. none of the above
3. If input A is asserted, the outputs will be:

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

/* Different Priority Encoder */
module diff_pri(A, B, C, D, E, GS);
    input wire A, B, C, D;
    output wire [1:0] E;
    output wire GS;

    reg [2:0] EGS;

    assign E = EGS[2:1];
    assign GS = EGS[0];

    always @ (A, B, C, D) begin
        casez ({A,B,C,D})
            4'b0000: EGS = 3'b000;
            4'b0001: EGS = 3'b111;
            4'b001?: EGS = 3'b101;
            4'b01??: EGS = 3'b011;
            4'b1???: EGS = 3'b001;
        endcase
    end
endmodule
4. When inputs B and C are asserted simultaneously (and A is negated) the outputs will be:

A. E1=0, E0=0, GS=1
B. E1=0, E0=1, GS=1
C. E1=1, E0=0, GS=1
D. E1=1, E0=1, GS=1
E. none of the above
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
Module 2-I
Combinational Building Blocks: Multiplexers
Reading Assignment:
*DDPP* 4th Ed., pp. 432-440, 445-446

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
Definition: A *multiplexer* is a digital switch that uses *s* select lines to determine which of *n = 2s* 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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$a_0$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$a_1$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$a_2$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$a_3$</td>
</tr>
</tbody>
</table>

Functional values assigned to each combination
### Multiplexer Truth Table Analogy

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Diagram: AND function**

- **Inputs:** X, Y
- **Outputs:** D0, D1, D2, D3, F
- **Truth Table:**
  - F(X,Y) = \( i_1 \cdot i_0 \)
- **Function:** F(X,Y) = AND function

- **Outputs connected to inputs:**
  - D0
  - D1
  - D2
  - D3

**F(X,Y) output**
Multiplexer Truth Table Analogy

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

OR function
Multiplexer Truth Table Analogy

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

XOR function

\[ F(X,Y) = D_0 i_1 i_0 + D_1 i_1 i_0 + D_2 i_1 i_0 + D_3 i_1 i_0 \]
## Multiplexer Truth Table Analogy

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The table above represents a multiplexer with inputs X and Y, and outputs F(X,Y). The XNOR function is shown in the diagram.

![Multiplexer Diagram](image)

The diagram illustrates how the inputs X and Y are mapped to the outputs D0, D1, D2, and D3 through the XNOR function.
Multiplexer Truth Table Analogy

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a₀</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>a₁</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>a₂</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>a₃</td>
</tr>
</tbody>
</table>

Question: How many different functions of $S$ variables are possible?
Multiplexer Truth Table Analogy

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

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>F(X,Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a₀</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>a₁</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>a₂</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>a₃</td>
</tr>
</tbody>
</table>

Answer: \(2^2\)
Example: 8-to-1 (8:1) Multiplexer

Data inputs

D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7

8:1

Select lines

F

Output

D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7

i_2 i_1 i_0
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 $F(X,Y,Z) = X \cdot Z + X' \cdot (Y \oplus Z)$

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

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

$= X \cdot Z + X' \cdot Y' \cdot Z + X' \cdot Y \cdot Z'$
Example: Multiplexer Function Realization

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

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

Eight to One MUX

$F(X,Y,Z) = \Sigma_{X,Y,Z}(1,2,5,7)$
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)
Example: 8-to-1 x 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


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

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

module mux418a(EN, S, A, B, C, D, Y_z);

input wire EN; // Tri-state output enable line
input wire [1:0] S; // Select inputs
input wire [7:0] A, B, C, D; // 8-bit input buses
output tri [7:0] Y_z; // 8-bit output bus

wire [7:0] Y;

assign Y = !S[1] & !S[0] & A |
            !S[1] & S[0] & B |
            S[1] & !S[0] & C |
            S[1] & S[0] & D;

assign Y_z = EN ? Y : 8'bZZZZZZZZZ;

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

text

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'bZZZZZZZZZ;

  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
Clicker Quiz
/* 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
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
2. When EN=0, S1=1, and S0=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 inputs D
   E. none of the above
3. When EN=1, S1=1, and S0=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
Summary

- Once we have a formal description of a 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

- Key combinational building blocks include decoders, encoders, tri-state outputs (buses), and multiplexers
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
Module 2-J
Top Level Modules

Introduction to
Digital System Design
Reading Assignment:
*DDPP* 4th Ed., pp. 306-308

Learning Objectives:
- Understand the need for using top level modules
- Understand how top level modules are created in Verilog
Outline

- Overview
- Instantiating modules
- Example top level modules
Overview

- **Definition**: The highest level module in 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.
- As they are the highest in hierarchy, it is possible that they have an empty port list.

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

module top_name();
  wire w, x, 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 order assigns A = w, B = x C = y, D = z by how they are ordered in the instantiation

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