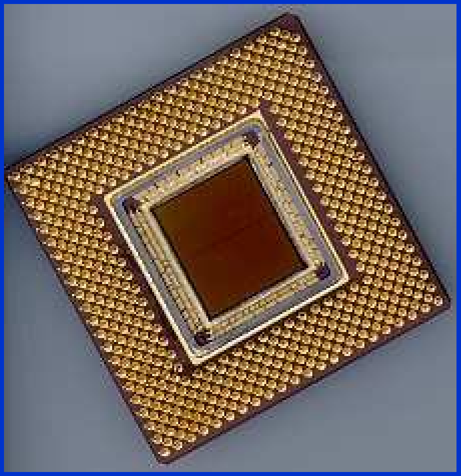


Introduction to Digital System Design

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**

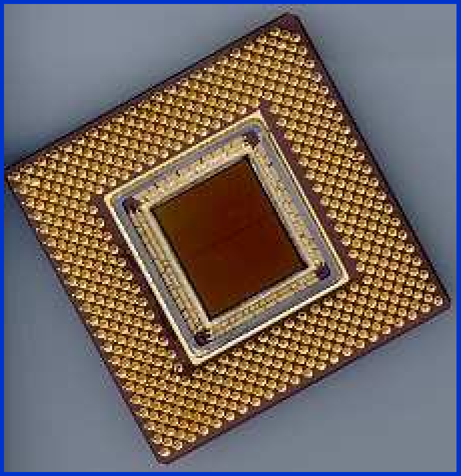
Glossary of Common Terms

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

Module 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



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

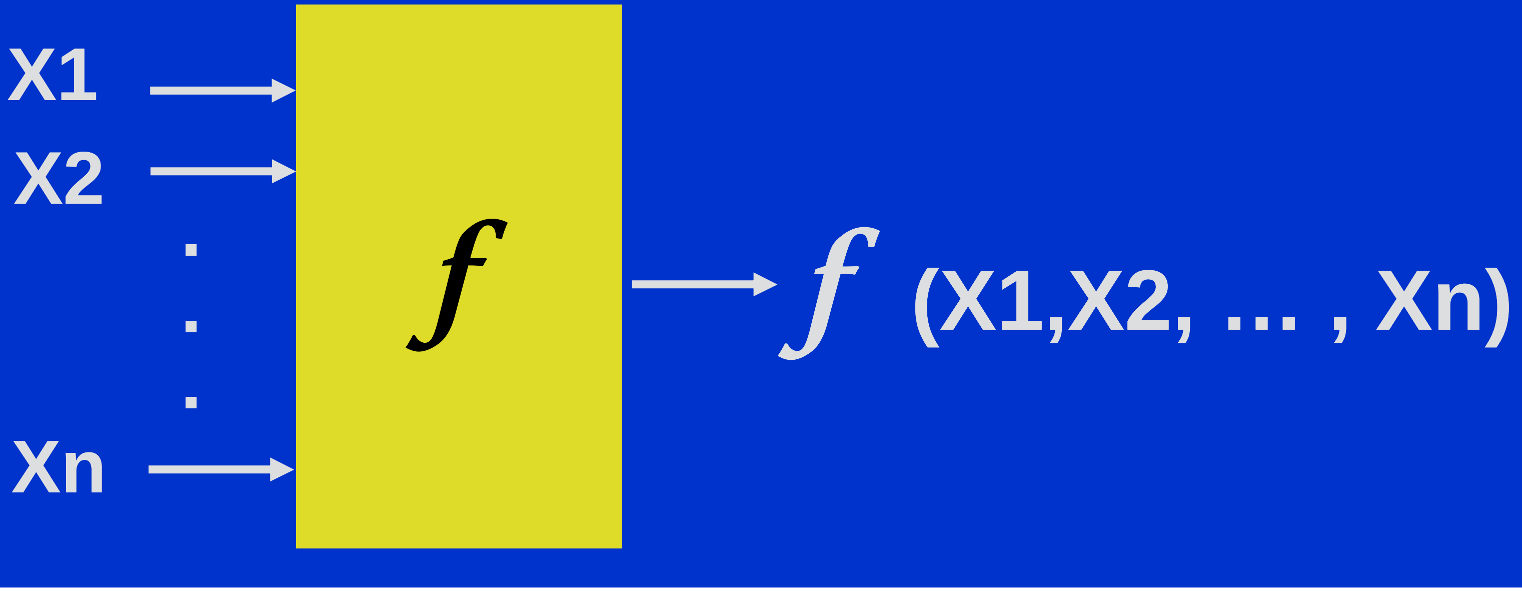
- Overview
- Definitions
- Minterm identification
- Maxterm identification
- ON Sets and OFF sets
- Combinational circuit analysis
- Equivalent symbols
- Combinational circuit synthesis

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**



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'	***Literals***
$W \bullet X \bullet Z$	***Product Term***
$X \bullet Y' + W \bullet Z$	***Sum of Products Expression***
$X + Y + Z'$	***Sum Term***
$(X + Y) \bullet (W + Z')$	***Product of Sums Expression***

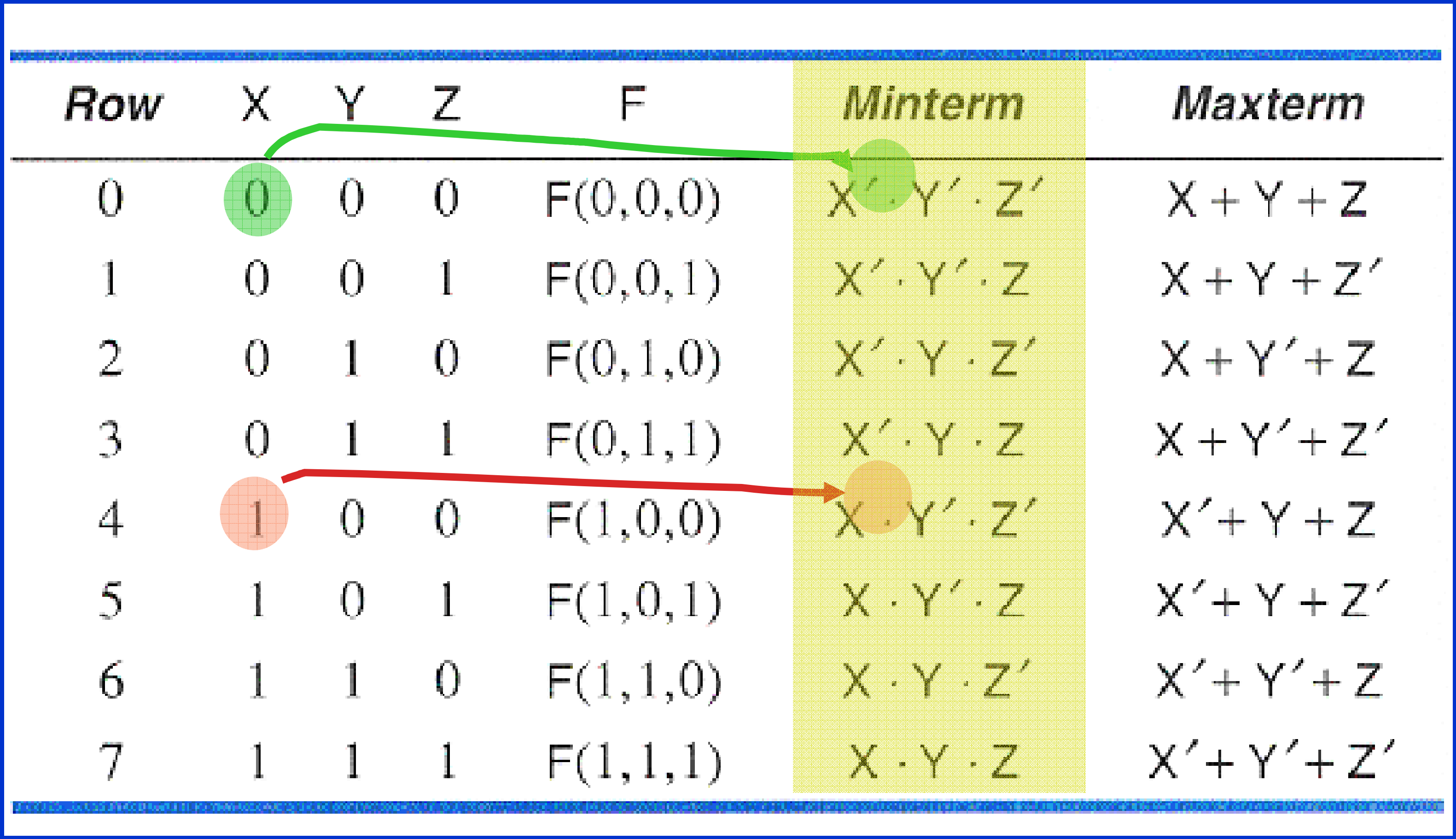
Definitions

- 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

0 → complemented

1 → true

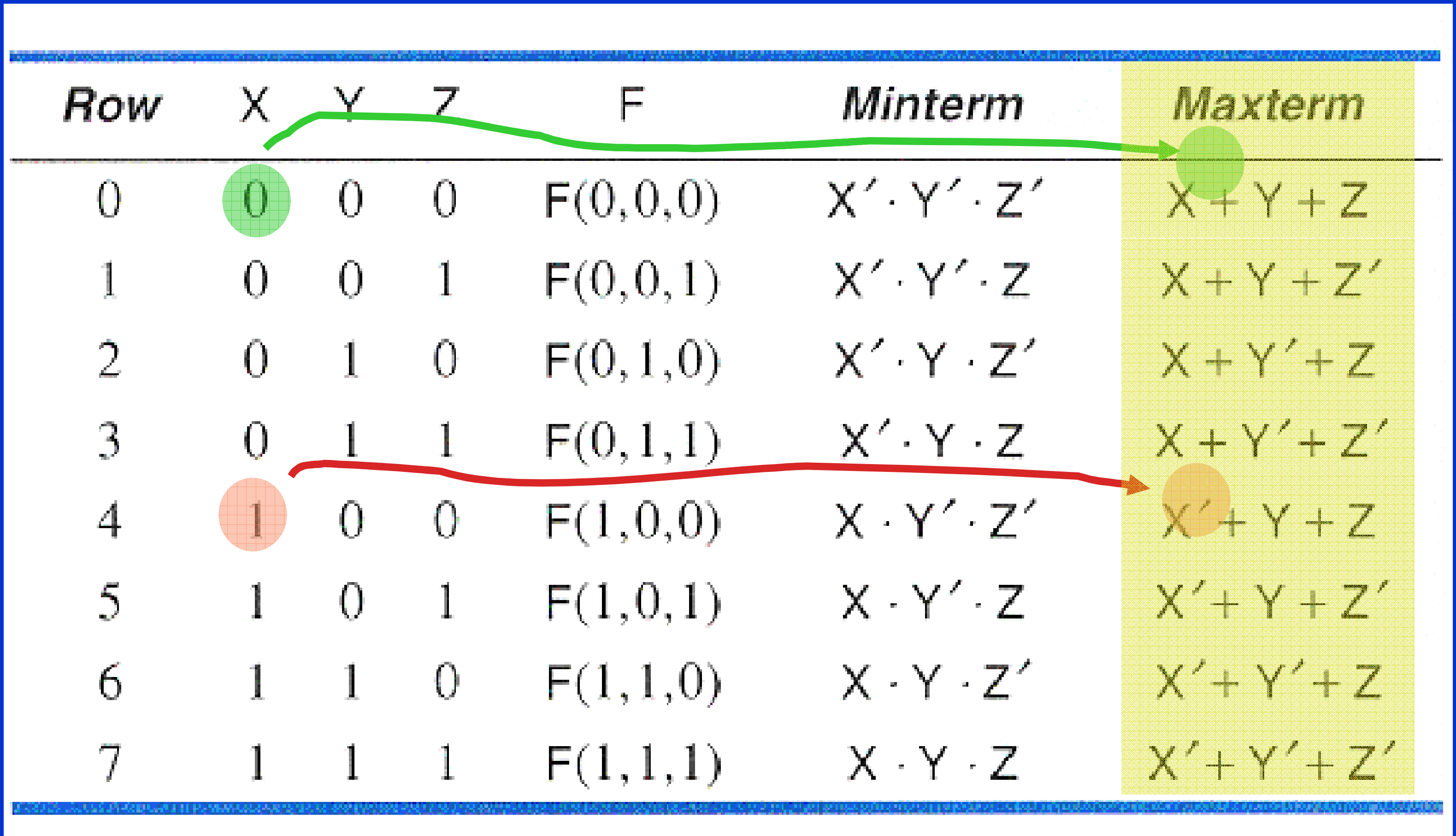


Row	X	Y	Z	F	Minterm	Maxterm
0	0	0	0	F(0,0,0)	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	F(0,0,1)	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	F(0,1,0)	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	F(0,1,1)	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	F(1,0,0)	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	F(1,0,1)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

Maxterm Identification

0 → true

1 → complemented



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

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

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

F(X,Y,Z) expressed as:

an *on-set:* ______________________________

an *off-set:* ______________________________

a *sum of minterms:* ____

a *product of maxterms:* ____

____

Example

Based on the truth table, determine the following

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

F(X,Y,Z) expressed as:

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

an *off-set:* ______________________________

a *sum of minterms:* ____

a *product of maxterms:* ______________________________________

____

Example

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

Based on the truth table, determine the following

F(X,Y,Z) expressed as:

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

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

a *sum of minterms:* ________________________________

a *product of maxterms:* ____________________________

____

Example

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

Based on the truth table, determine the following

F(X,Y,Z) expressed as:

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

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

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

a *product of maxterms:* ____________________

____

Example

Based on the truth table, determine the following

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

F(X,Y,Z) expressed as:

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

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

a *sum of minterms:* $X'\bullet Y'\bullet Z' + X'\bullet Y\bullet Z + X\bullet Y\bullet Z' + X\bullet Y\bullet Z$

a *product of maxterms:*

$(X+Y+Z')\bullet(X+Y'+Z)\bullet (X'+Y+Z)\bullet(X'+Y+Z')$

Clicker Quiz

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

A. $\Sigma_{X,Y,Z}$**(7)**

B. $\Sigma_{X,Y,Z}$**(0)**

C. $\Sigma_{X,Y,Z}$**(0,1,2,3,4,5,6)**

D. $\Sigma_{X,Y,Z}$**(1,2,3,4,5,6,7)**

E. none of the above

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

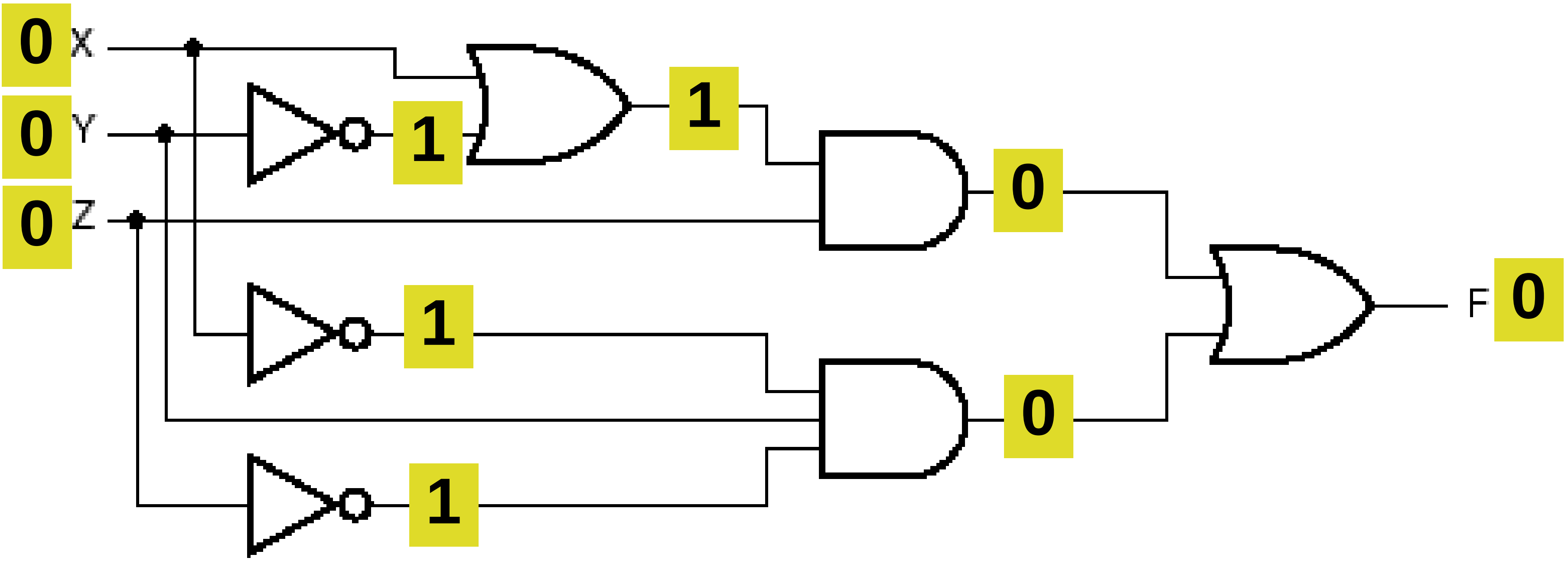
3. If the function **F(X,Y,Z)** is represented by the **ON SET** **$\sum_{X,Y,Z}(0,3,5,6)$**, then the **complement** of this function **F′(X,Y,Z)** is represented by the **ON SET**:

- **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

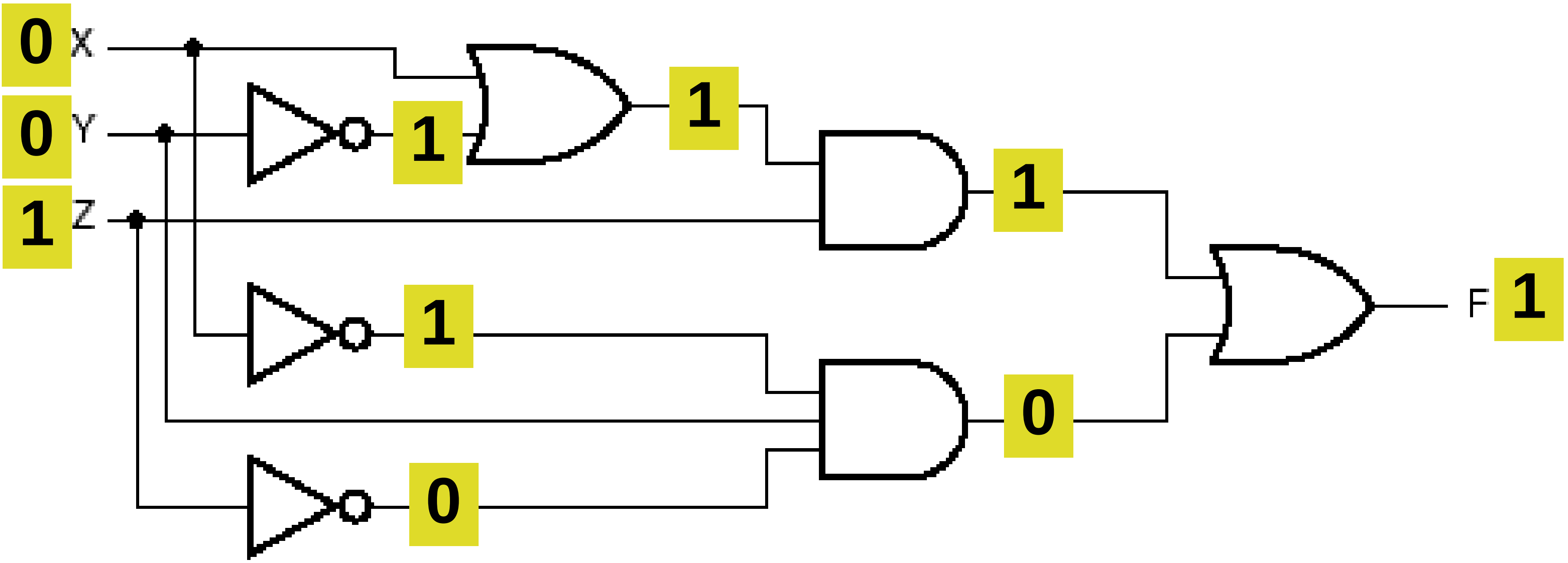
4. If the function **F(X,Y,Z)** is represented by the **ON SET** $\sum_{X,Y,Z}(0,3,5,6)$, then the **dual** of this function $F^D(X,Y,Z)$ is represented by the **ON SET:**

- **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

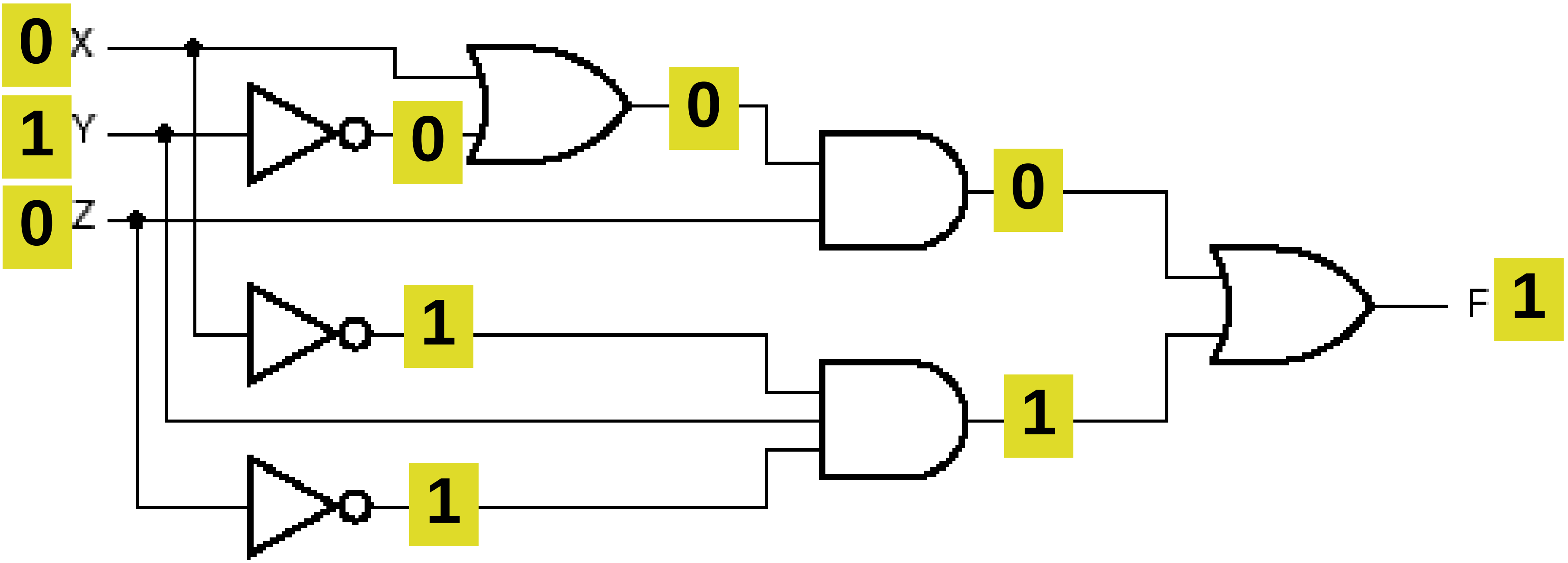
Example - Combinational Analysis



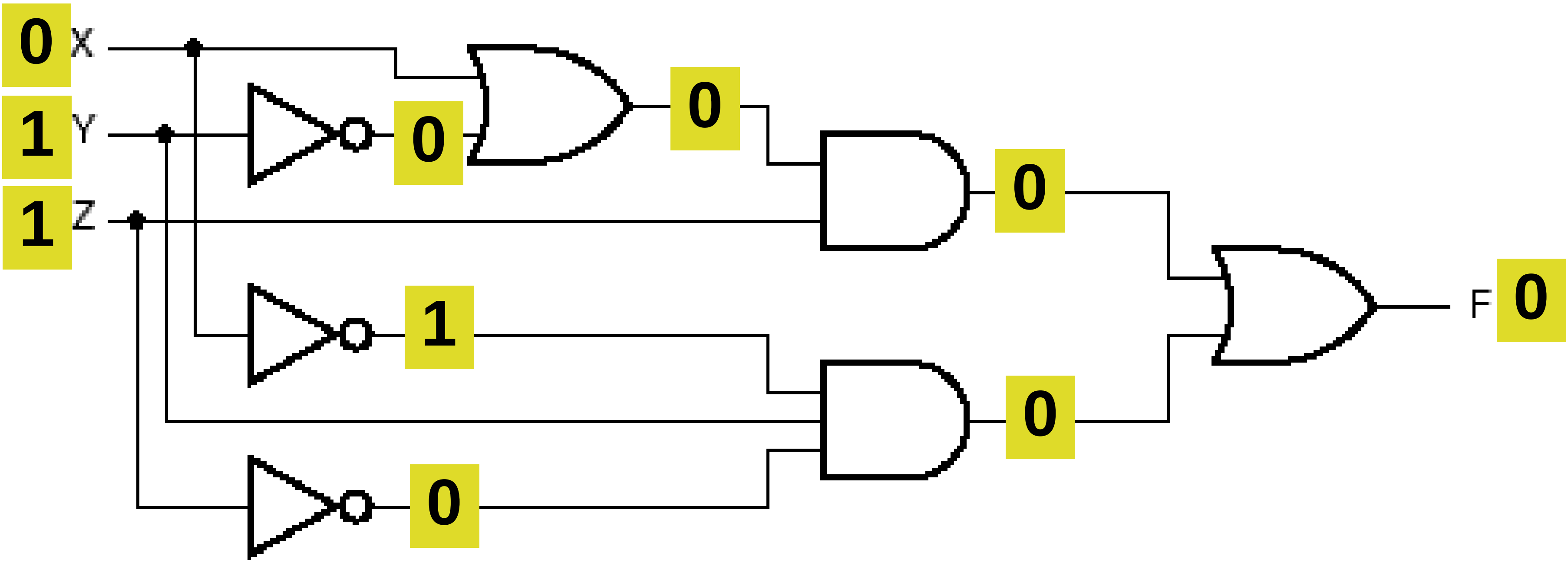
Example - Combinational Analysis



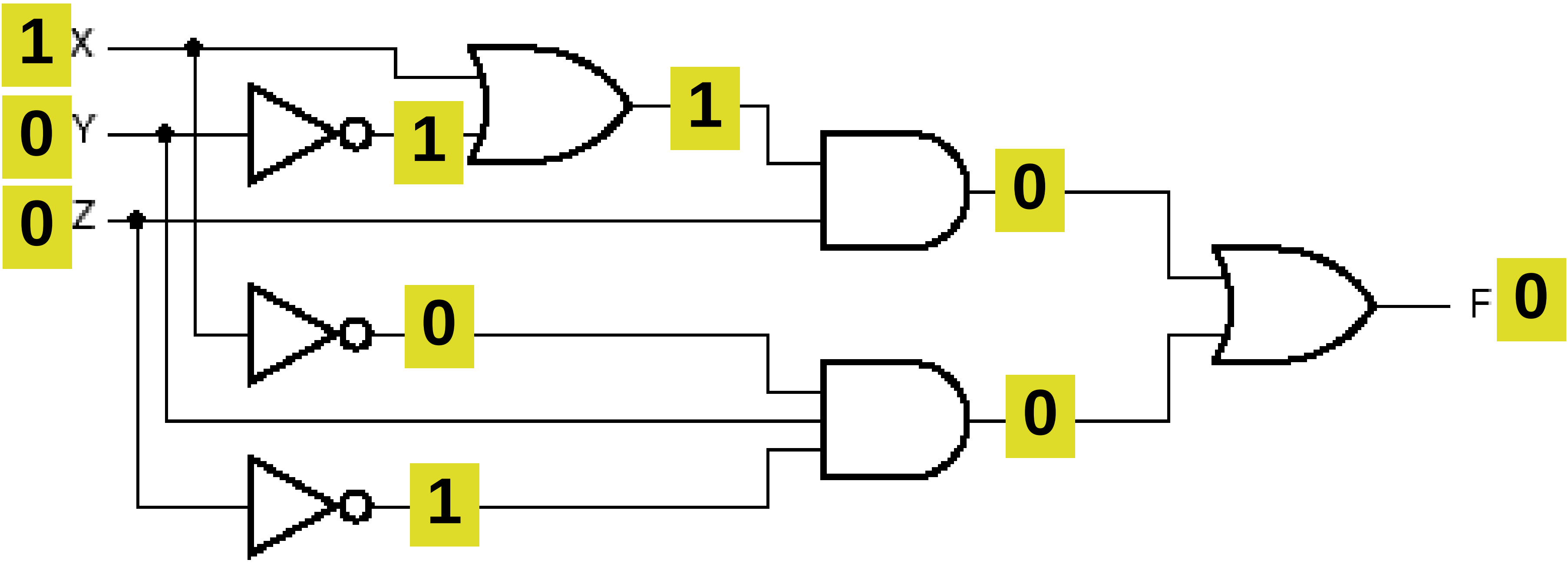
Example - Combinational Analysis



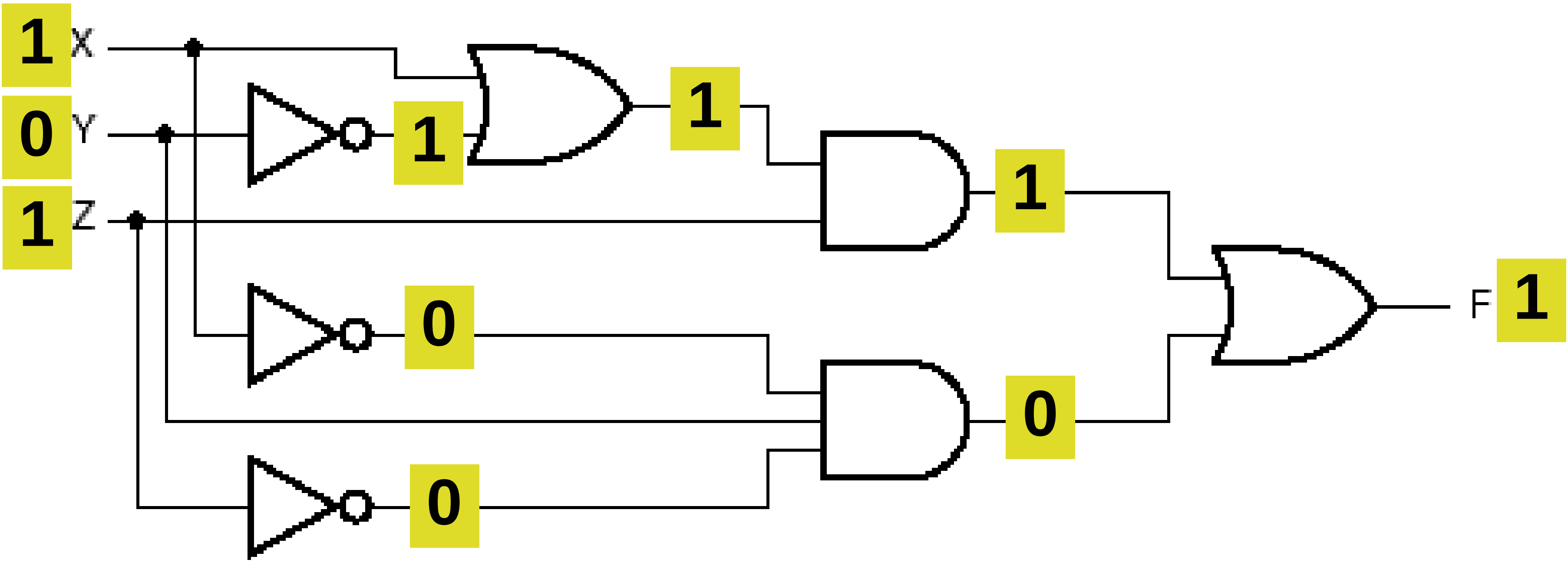
Example - Combinational Analysis



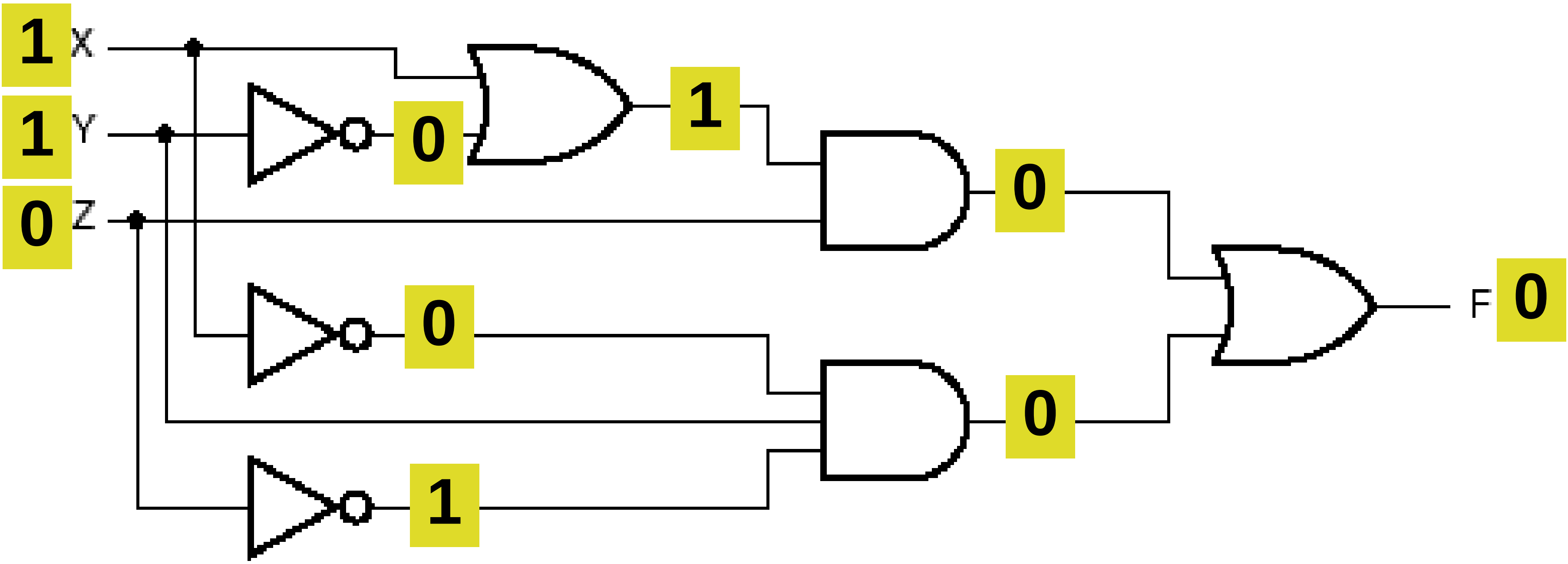
Example - Combinational Analysis



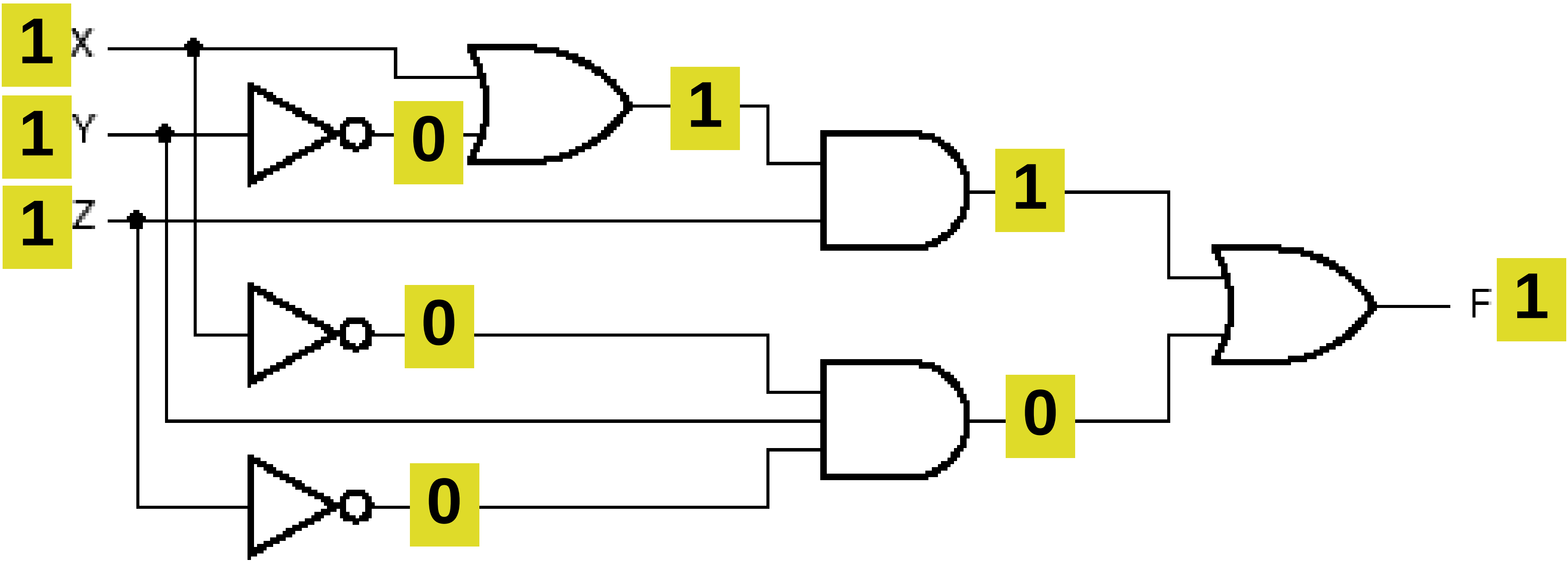
Example - Combinational Analysis



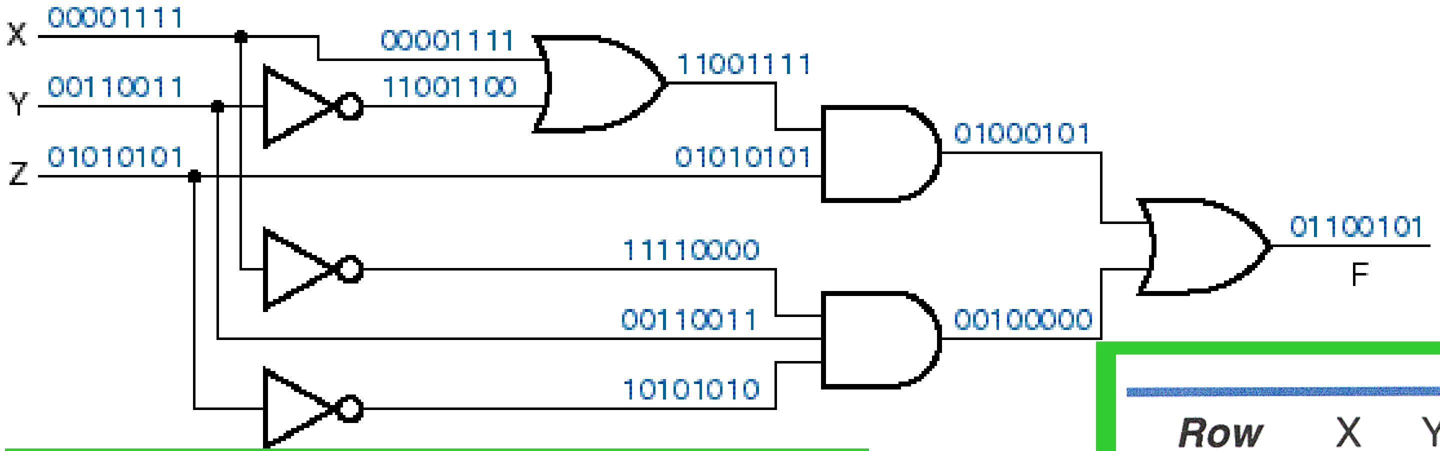
Example - Combinational Analysis



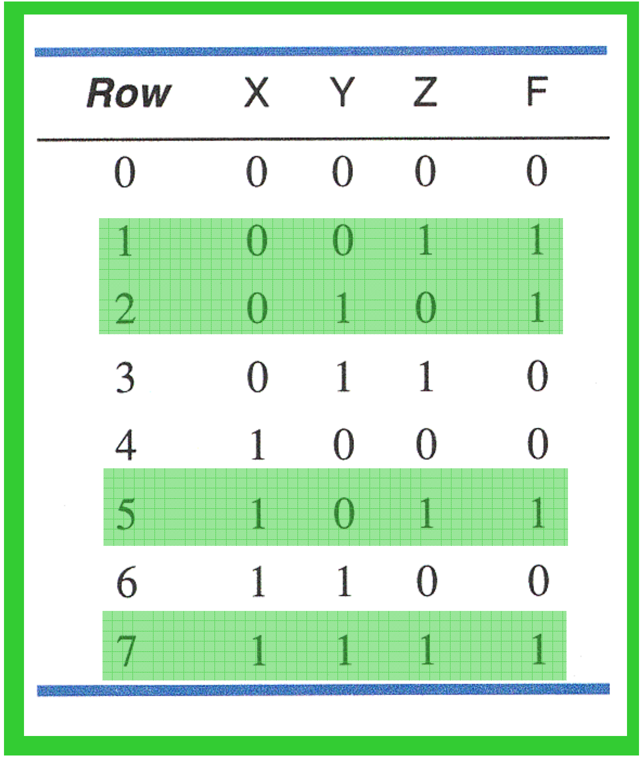
Example - Combinational Analysis



Example - Combinational Analysis



Truth Table

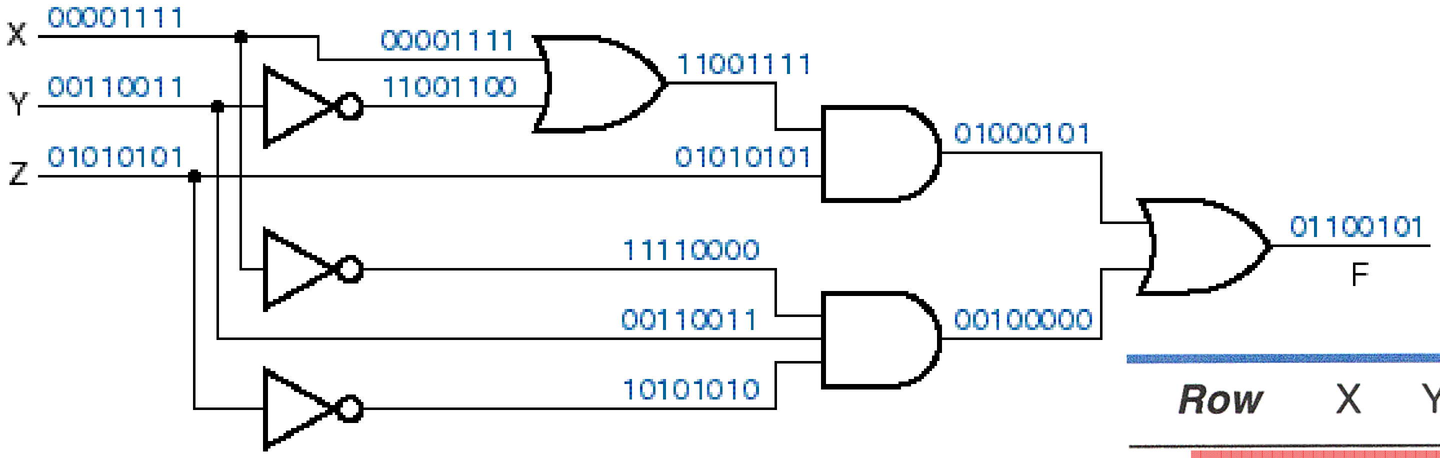


Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

The *"on set"* of this function is f(X,Y,Z) = ΣX,Y,Z(1,2,5,7)

The *canonical sum* of this function is f(X,Y,Z) = X′•Y′•Z + X′•Y•Z′ + X•Y′•Z + X•Y•Z

Example - Combinational Analysis

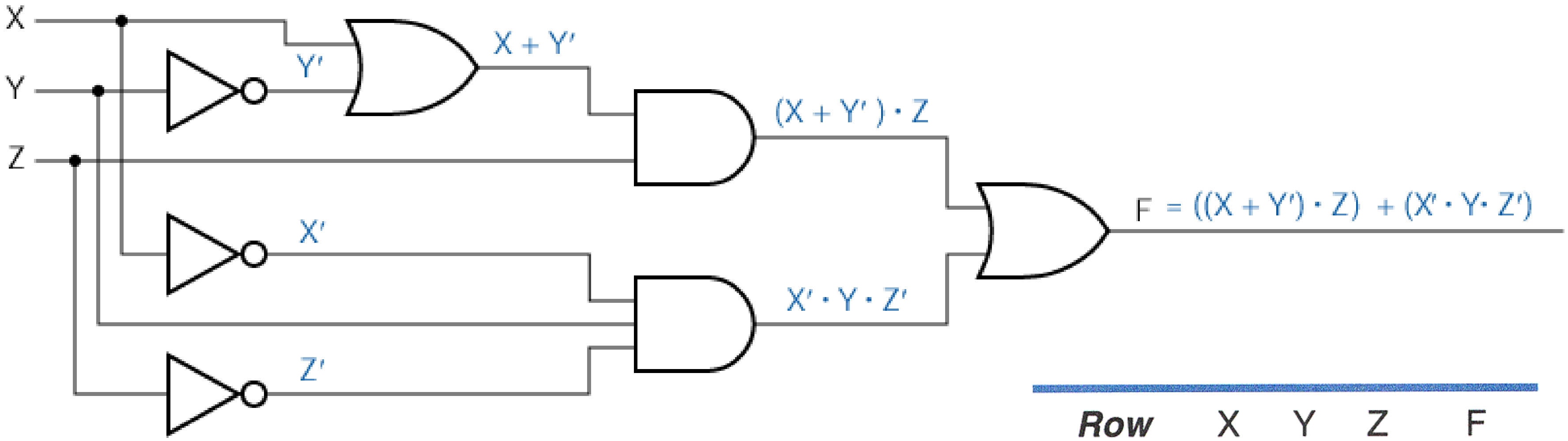


Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

The *“off set”* of this function is *f* (X,Y,Z) = ∏X,Y,Z(0,3,4,6)

The *canonical product* of this function is *f* (X,Y,Z) = (X+Y+Z) • (X+Y′+Z′) • (X′+Y+Z)• (X′+Y′+Z)

Example - Combinational Analysis



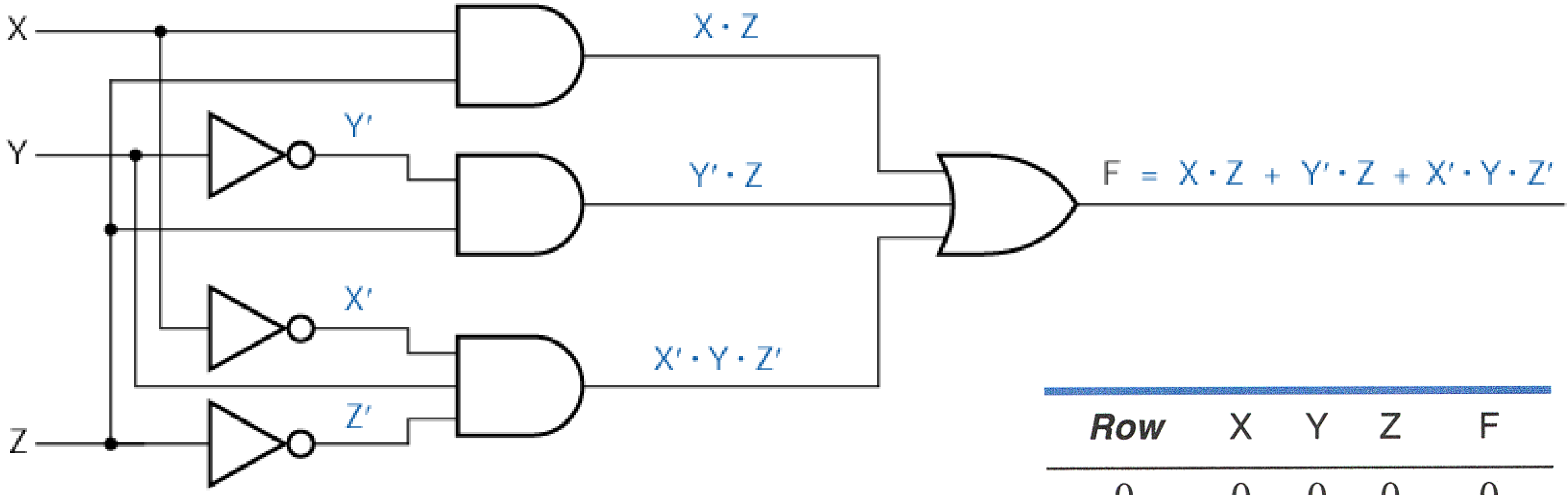
Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Writing the function implemented by this circuit “directly” yields *f* (X,Y,Z) =

((X+Y′)•Z) + (X′•Y•Z′) =

X•Z + Y′•Z + X′•Y•Z′

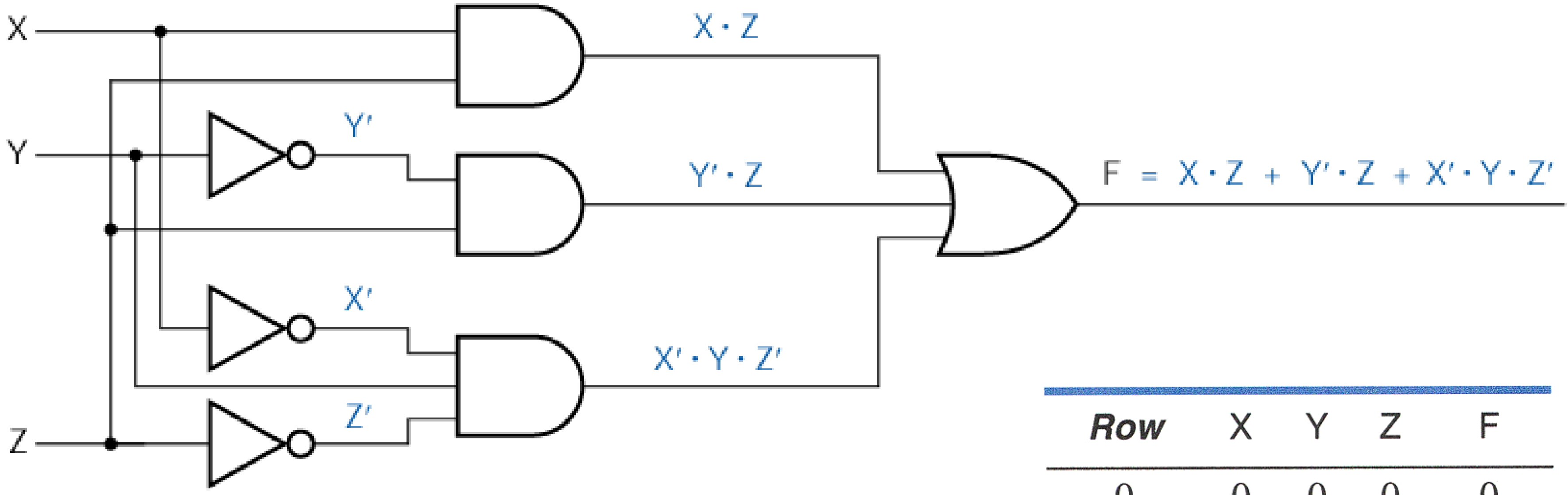
Example - Combinational Analysis



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

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

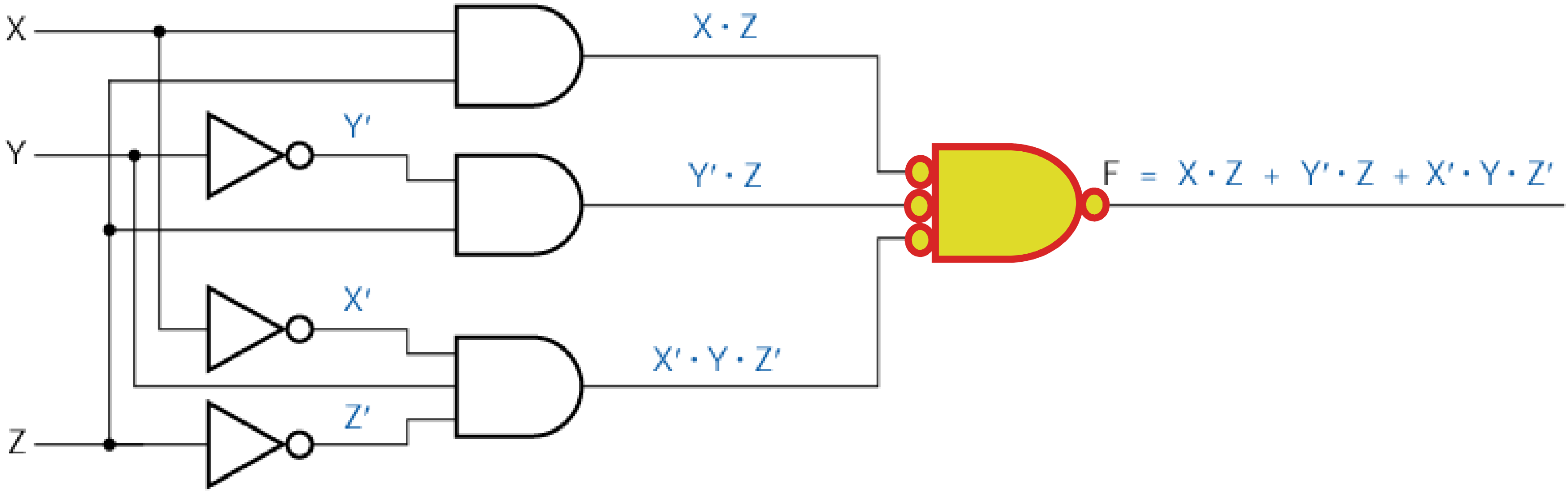
Example – Equivalent Symbols



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

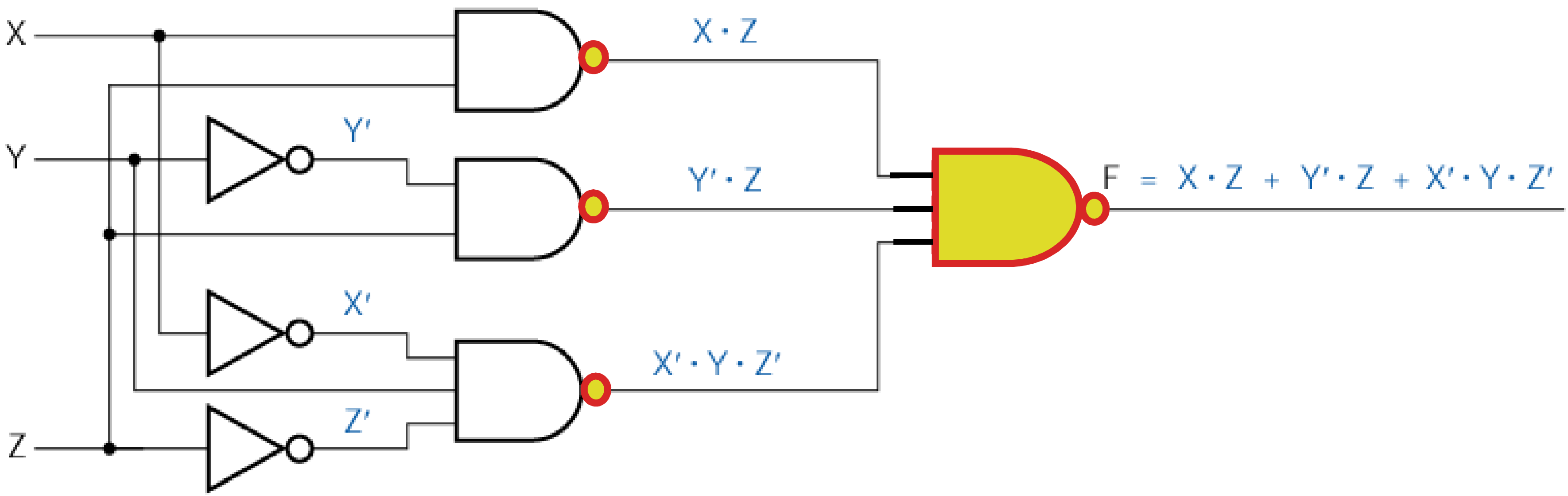
Recall that an *equivalent symbol* can be drawn for a gate by taking the *dual* of the operator and *complementing* all of the *inputs and outputs*

Example – Equivalent Symbols



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

Example – Equivalent Symbols



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

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

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

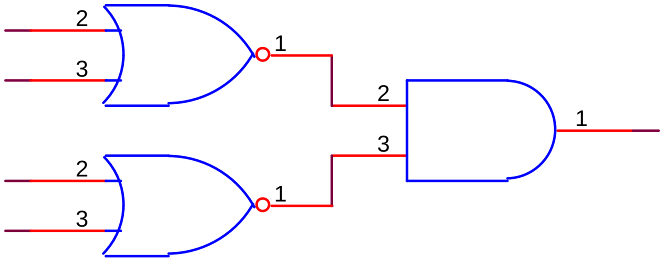
B. an OR gate with inverted inputs

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

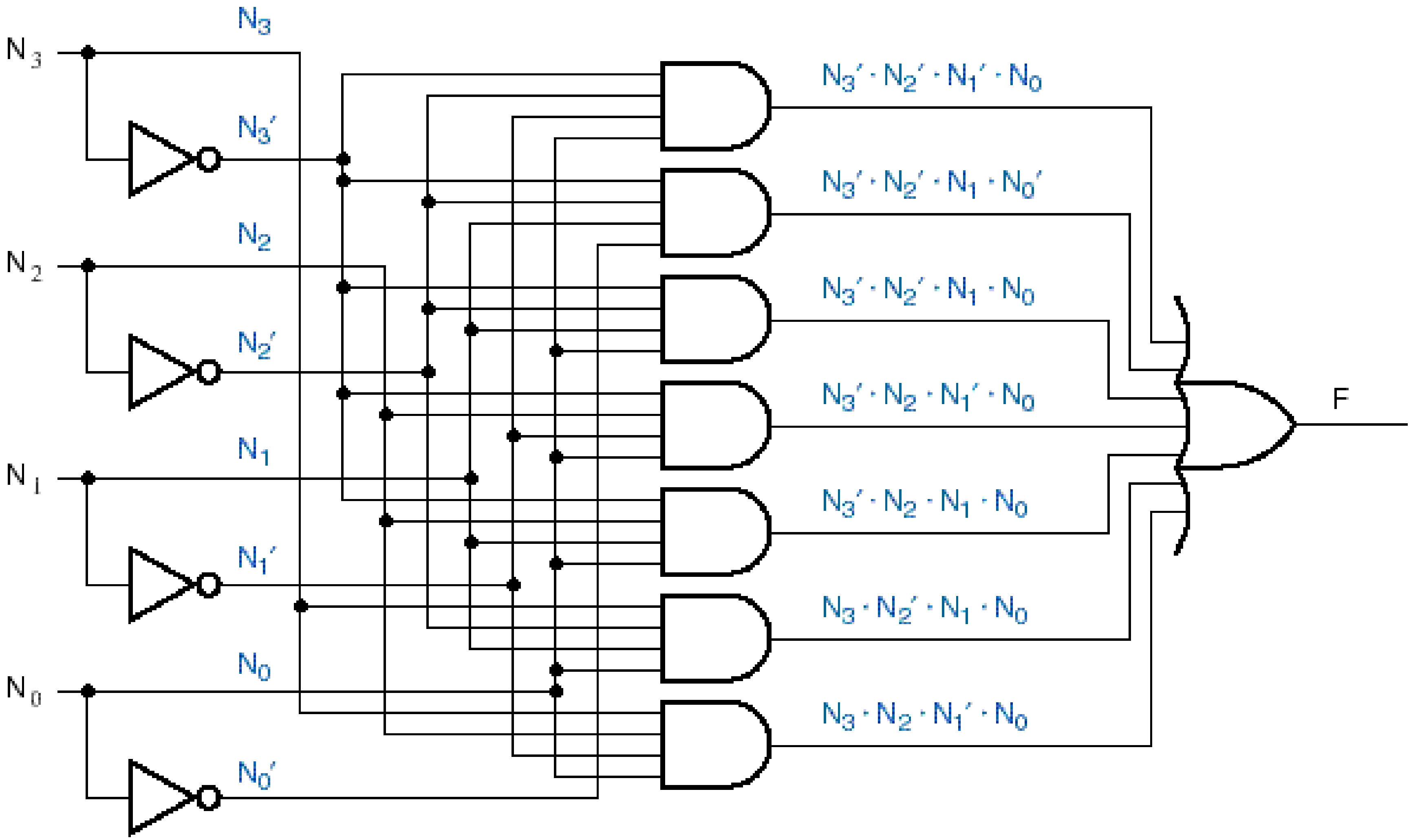
- A circuit *realizes* (“makes real”) an expression if its output function equals that expression
- Such a circuit is called a *realization* of the function
- Typically there are *many possible realizations* of the *same function*
- Circuit transformations can be made *algebraically* or *graphically*

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*)**

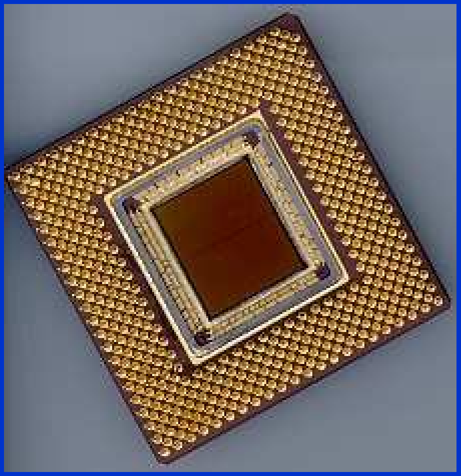
$$f\ (N_3,N_2,N_1,N_0) = \Sigma_{N_3,N_2,N_1,N_0}(1,2,3,5,7,11,13)$$

Example – Prime Number Detector


N_3
N_2
N_1
N_0
N_3
N_3'
N_2
N_2'
N_1
N_1'
N_0
N_0'
$N_3' \cdot N_2' \cdot N_1' \cdot N_0$
$N_3' \cdot N_2' \cdot N_1 \cdot N_0'$
$N_3' \cdot N_2' \cdot N_1 \cdot N_0$
$N_3' \cdot N_2 \cdot N_1' \cdot N_0$
$N_3' \cdot N_2 \cdot N_1 \cdot N_0$
$N_3 \cdot N_2' \cdot N_1 \cdot N_0$
$N_3 \cdot N_2 \cdot N_1' \cdot N_0$
F

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



Introduction to Digital System Design

Module 2-B

Mapping and Minimization

Reading Assignment:

DDPP 4th Ed., pp. 210-222

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′):

$$\text{Expression} \bullet X + \text{Expression} \bullet X' = \text{Expression}$$

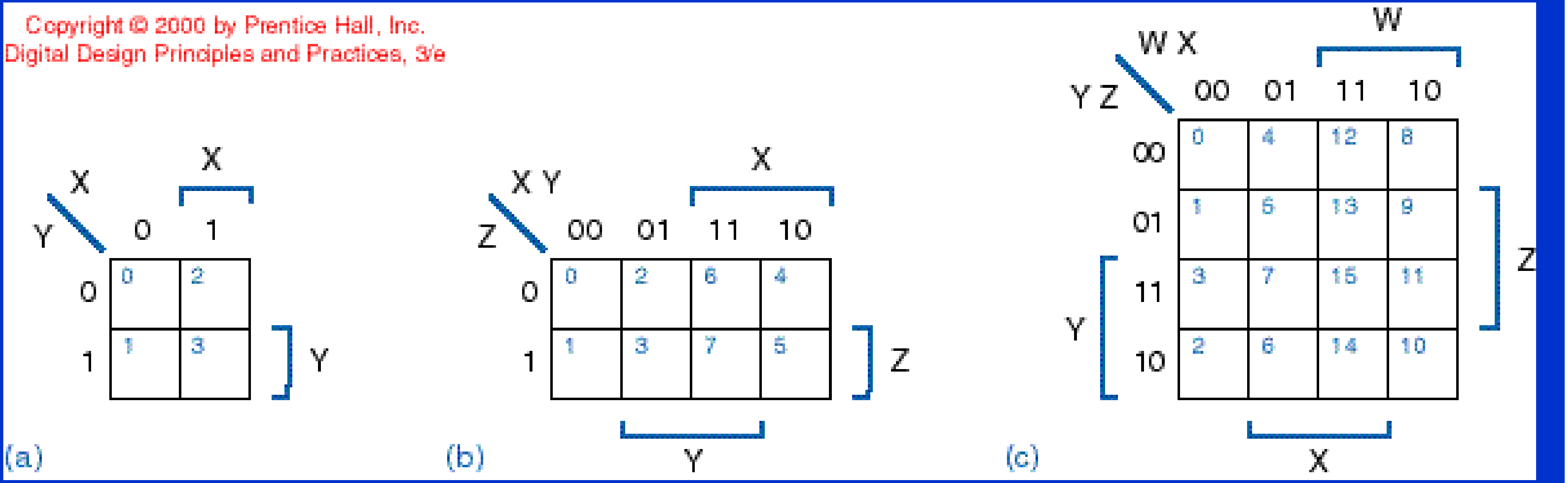
Overview

- 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**)



(a)

Y \ X	0	1
0	0	2
1	1	3

(b)

Z \ X Y	00	01	11	10
0	0	2	6	4
1	1	3	7	5

(c)

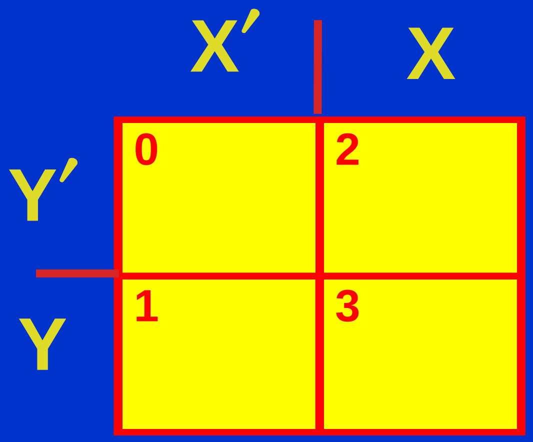
Y Z \ W X	00	01	11	10
00	0	4	12	8
01	1	5	13	9
11	3	7	15	11
10	2	6	14	10

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



	X′	X
Y′	0	2
Y	1	3

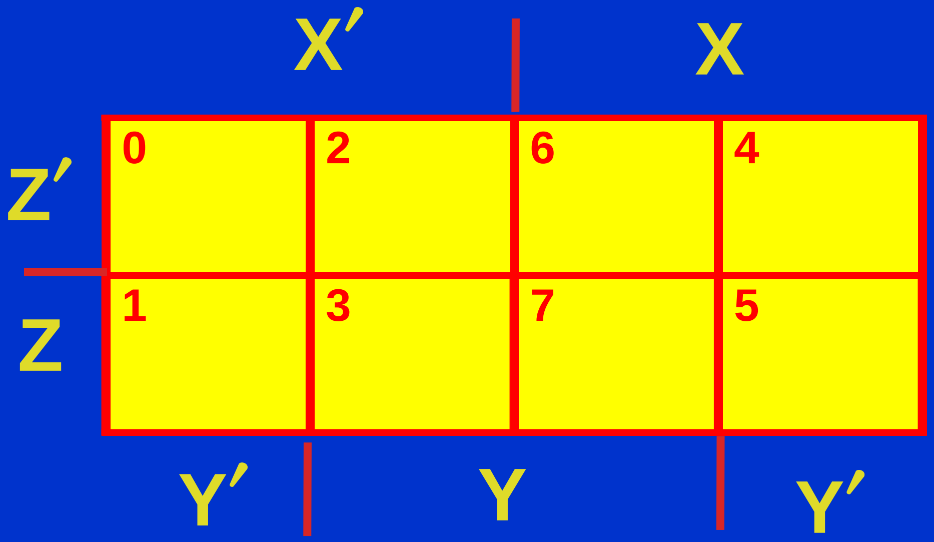
Karnaugh Maps

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

	X′	X
Y′	0 1	2 0
Y	1 1	3 1

Karnaugh Maps

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



	X′	X′	X	X
	Y′	Y	Y	Y′
Z′	0	2	6	4
Z	1	3	7	5

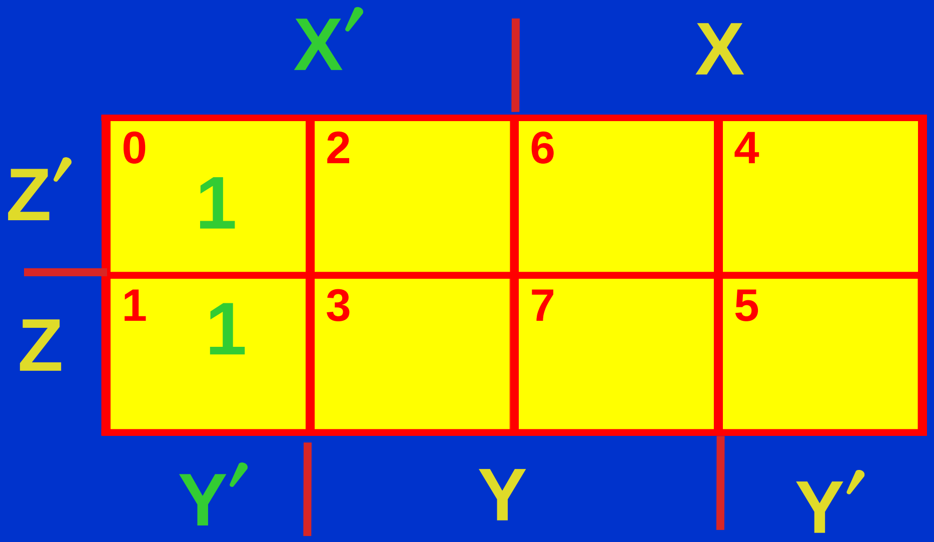
Karnaugh Maps

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

	X′		X	
Z′	0	2	6	4
Z	1	3	7	5
	Y′	Y		Y′

Karnaugh Maps

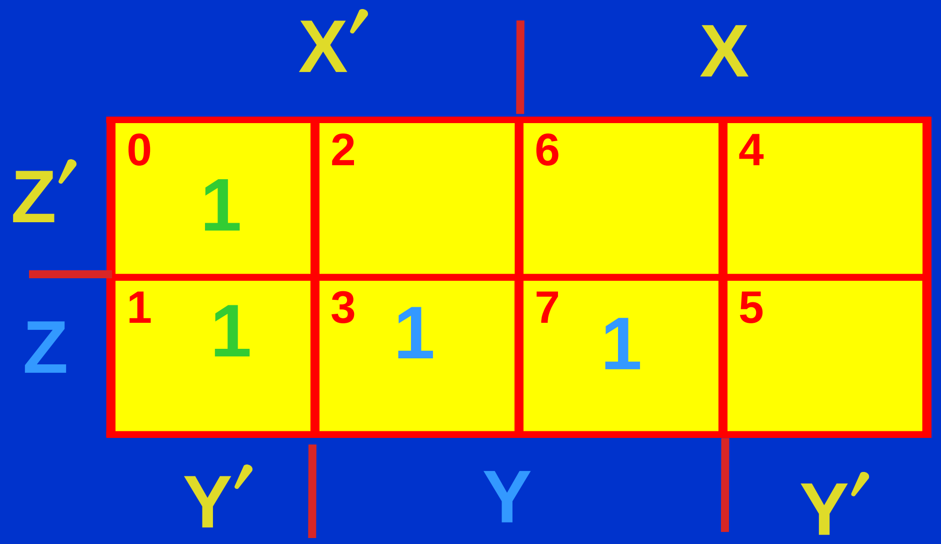
- **Example:** $f(X,Y,Z) = X' \bullet Y' + Y \bullet Z$



	X′		X	
	Y′	Y	Y	Y′
Z′	0 1	2	6	4
Z	1 1	3	7	5

Karnaugh Maps

- **Example:** f(X,Y,Z) = X′•Y′ + Y•Z



	X′		X	
Z′	0 1	2	6	4
Z	1 1	3 1	7 1	5
	Y′	Y		Y′

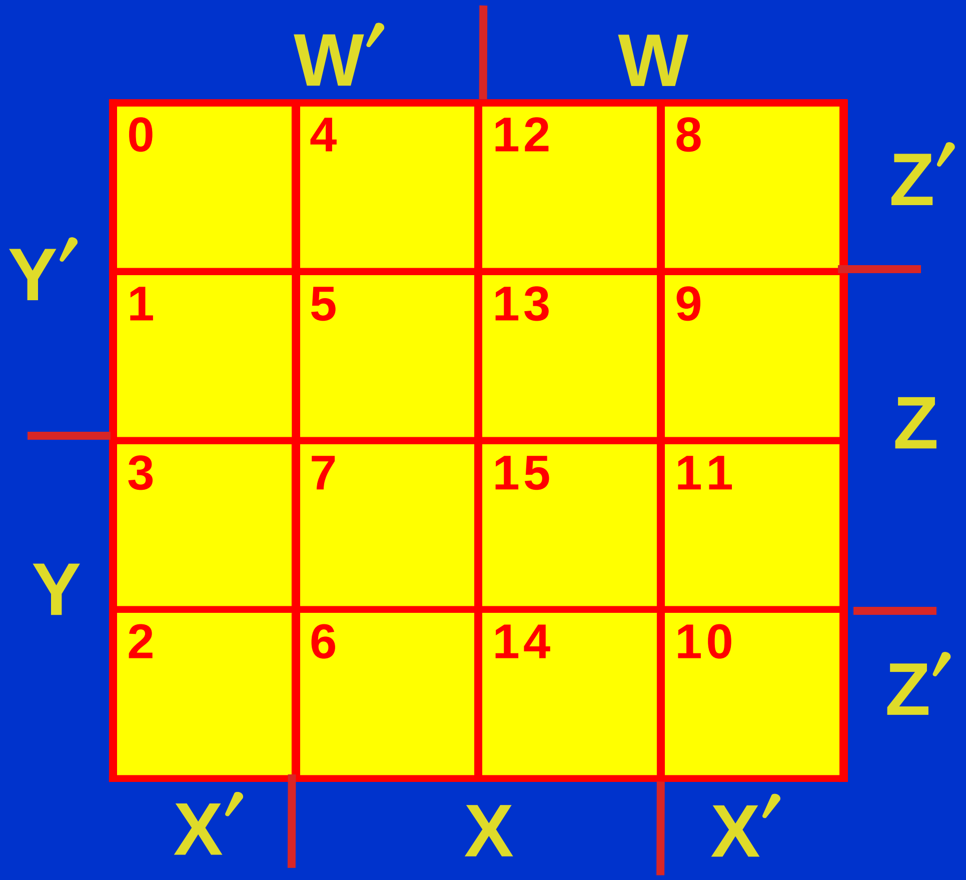
Karnaugh Maps

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

	X′		X	
Z′	0 1	2 0	6 0	4 0
Z	1 1	3 1	7 1	5 0
	Y′	Y		Y′

Karnaugh Maps

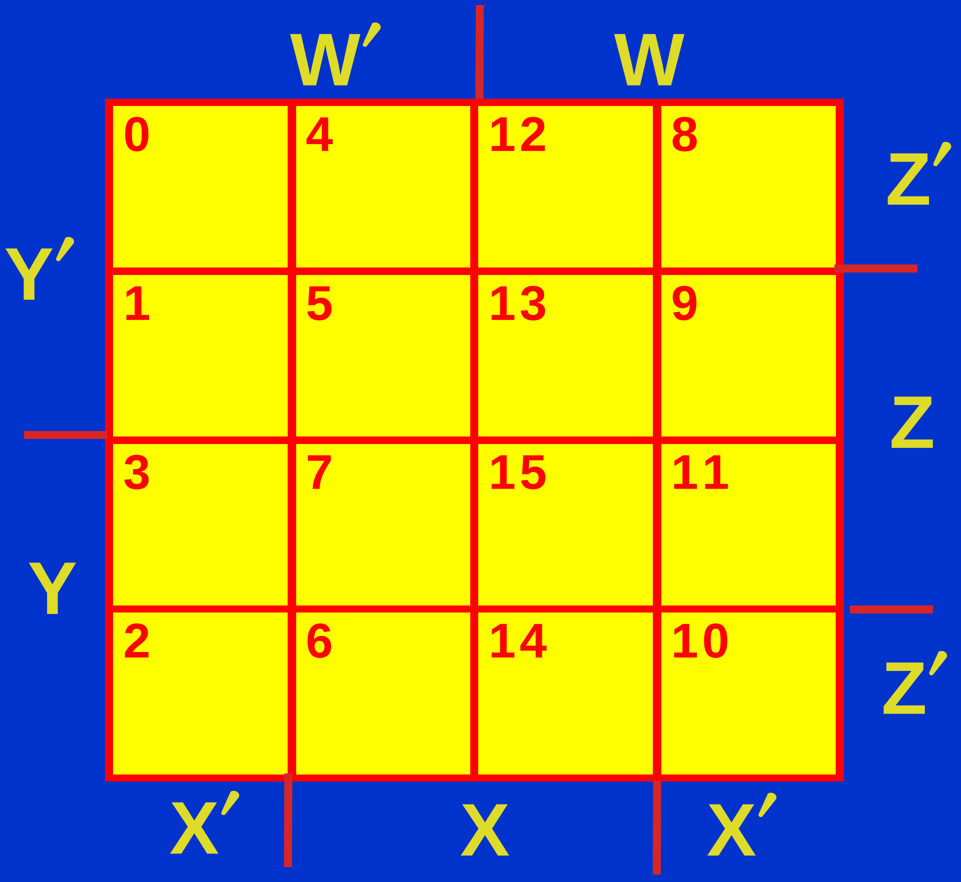
- An alternate drawing for a 4-variable K-map



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

Karnaugh Maps

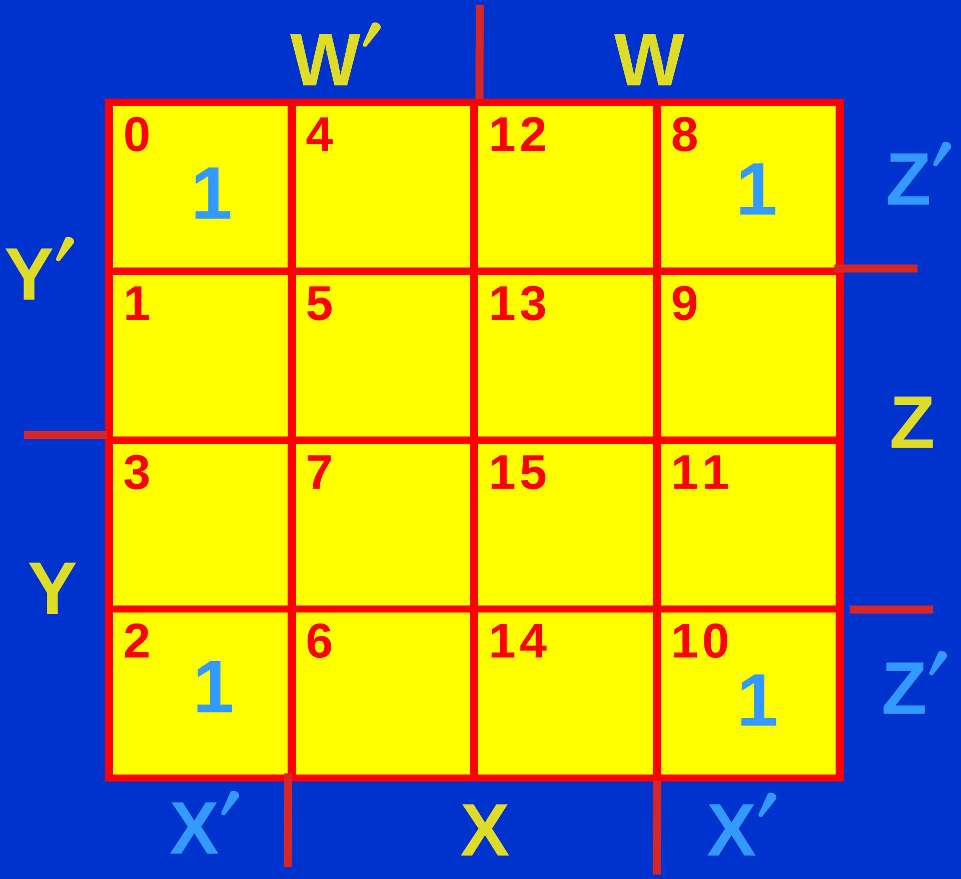
- **Example:** $f(W,X,Y,Z) = X' \bullet Z' + W \bullet Z + W' \bullet X$



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

Karnaugh Maps

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



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

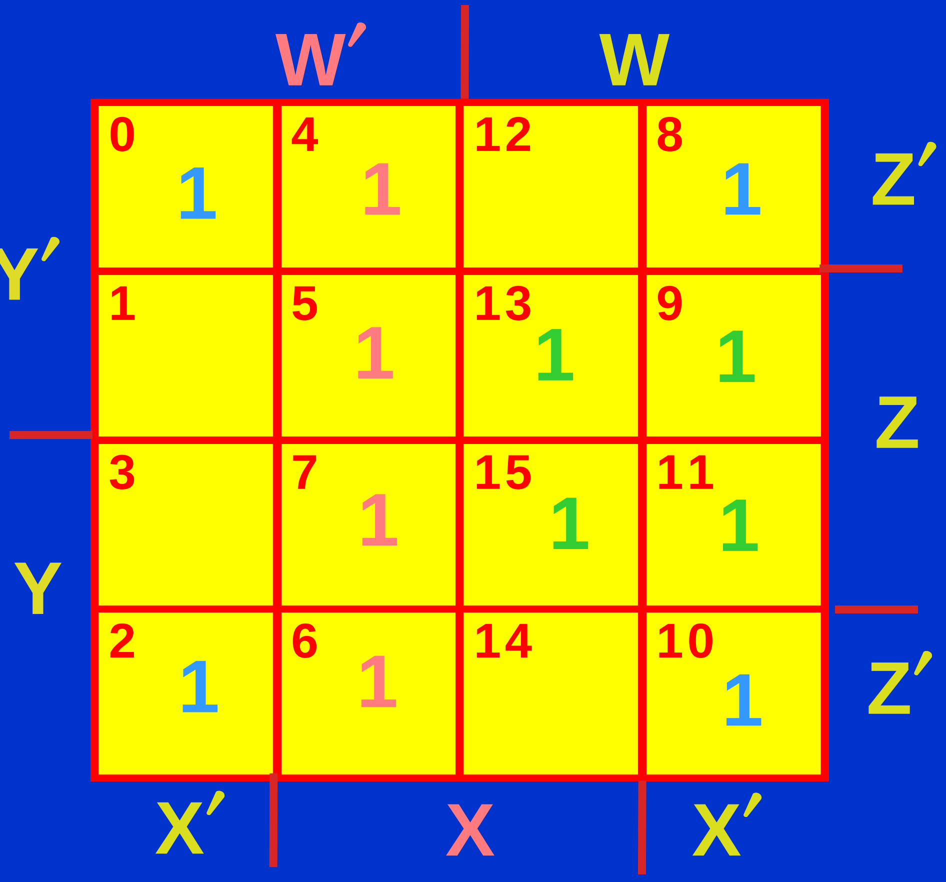
Karnaugh Maps

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

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

Karnaugh Maps

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



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

Karnaugh Maps

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

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

Minimization

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

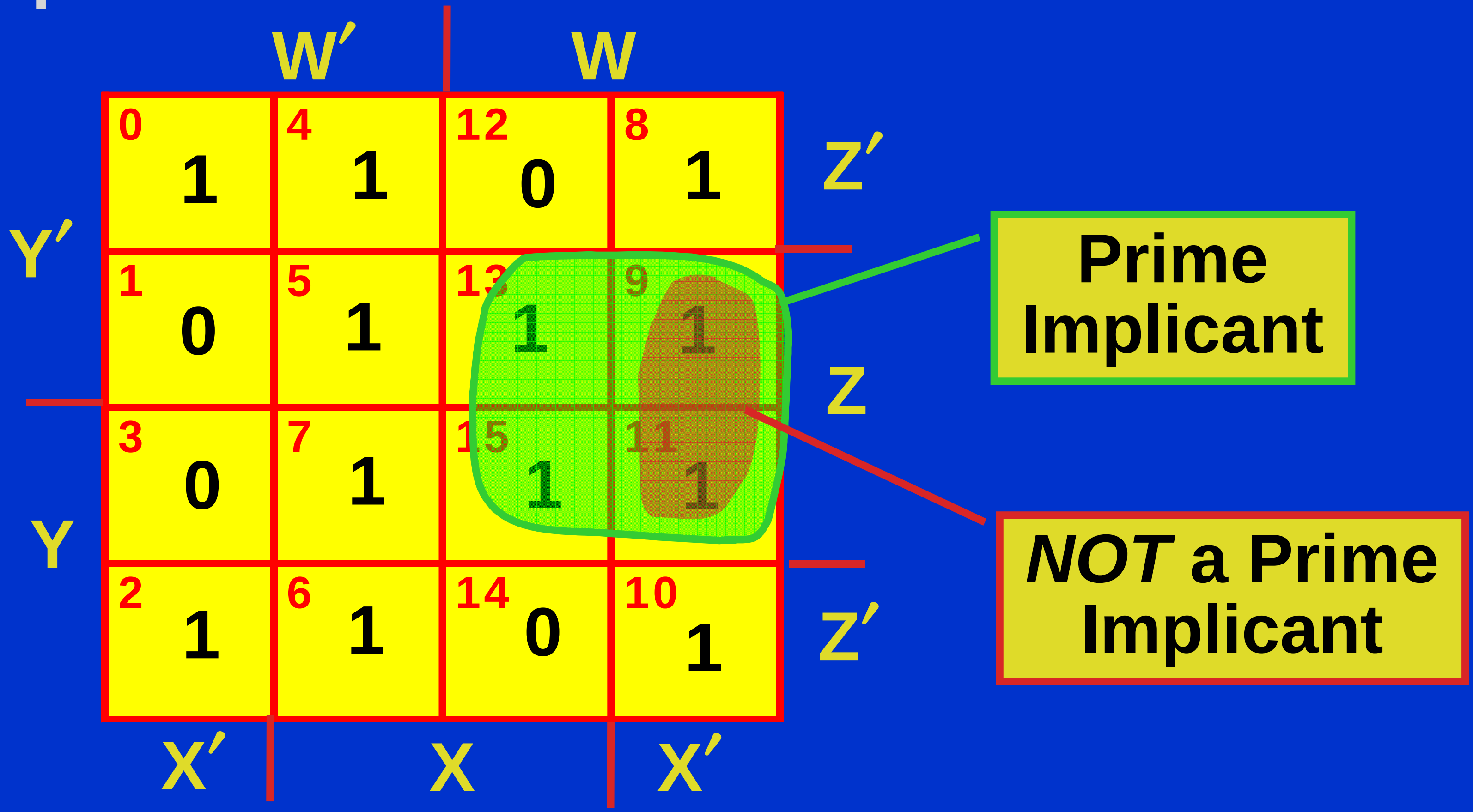
Translation: The *minimal sum* has the fewest possible product terms (first-level gates / second-level gate inputs) and the fewest possible literals (first-level gate inputs)

Minimization

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

Minimization

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



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

Prime Implicant

NOT a Prime Implicant

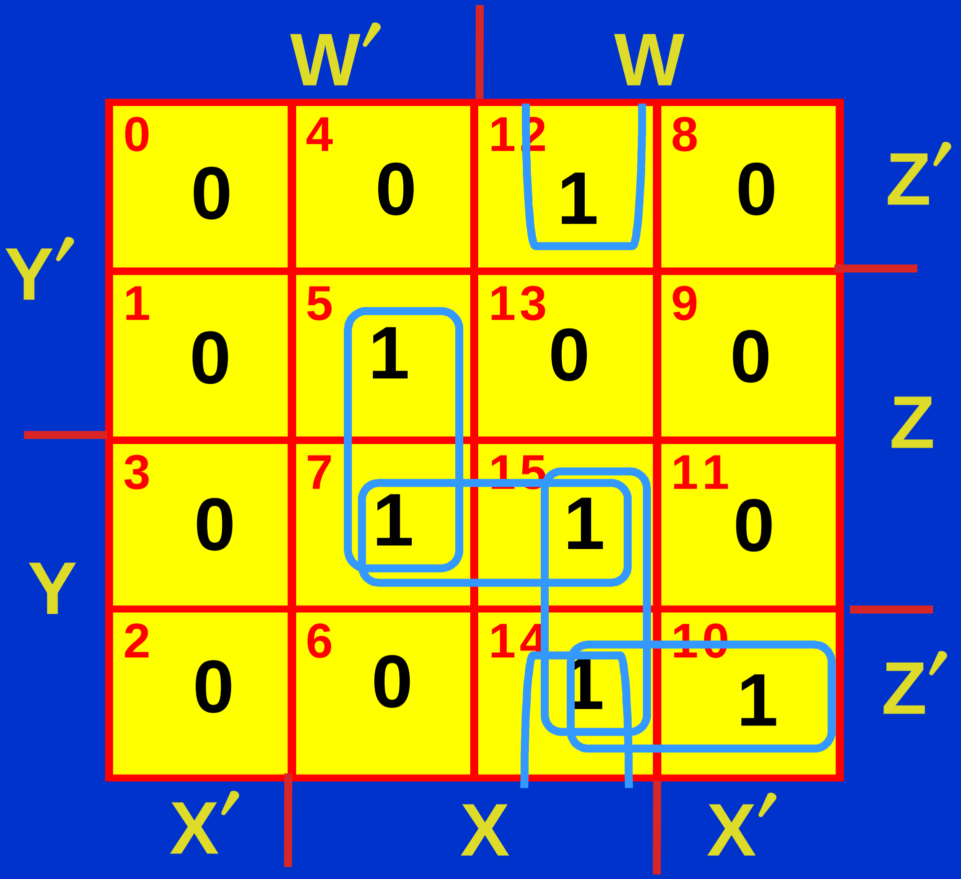
Minimization

- Prime Implicant Theorem: *A minimal sum is a sum of prime implicants* (i.e., to find a minimal sum, we need not consider any product terms that are not prime implicants)
- Definition: An *essential prime implicant* has at least one square in the grouping *not shared* by another prime implicant, i.e., it has at least one “unique” square, called a *distinguished 1-cell*
- Definition: A *non-essential prime implicant* is a grouping with no unique squares
- Definition: The *cost criterion* we will use is that *gate inputs and outputs are of equal cost*

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

Minimization Procedure

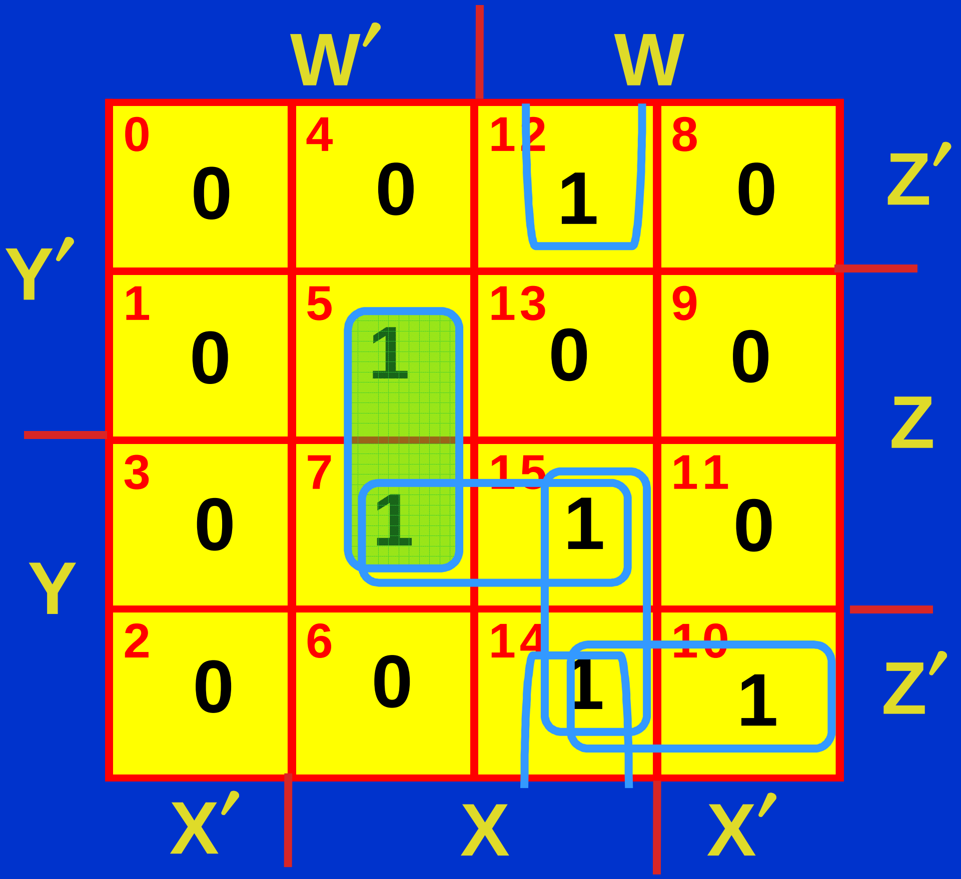
- **STEP 1:** **Circle all the prime implicants**



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

Minimization Procedure

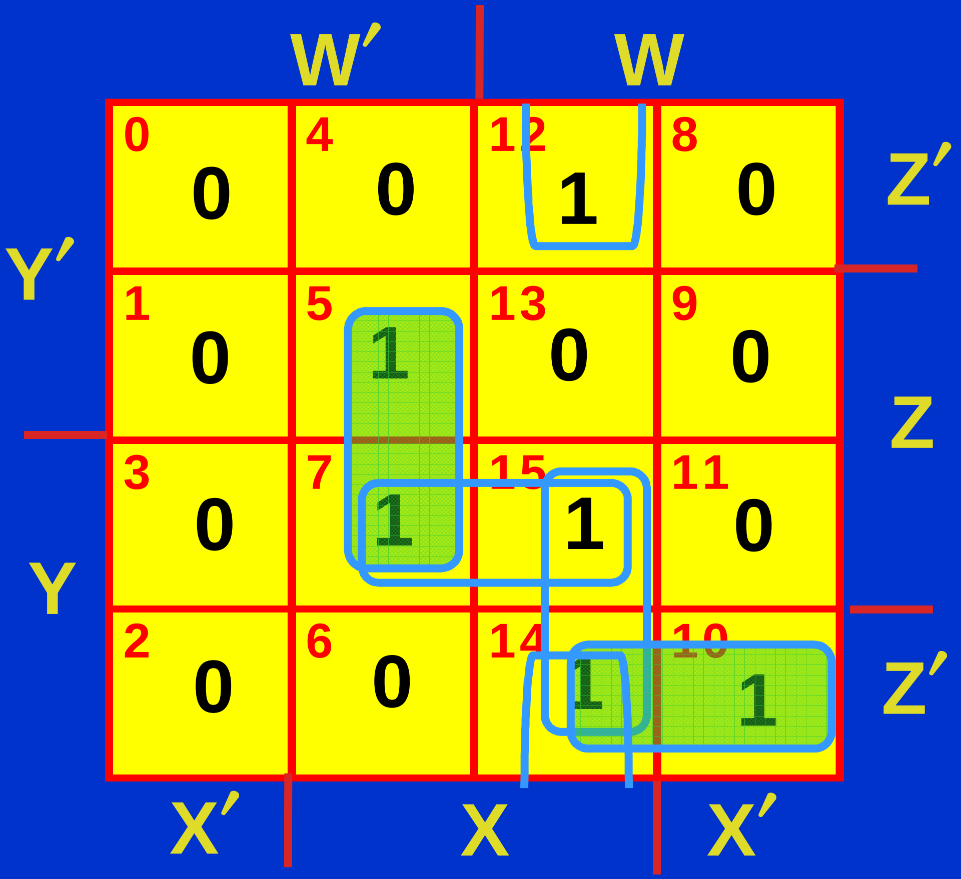
- **STEP 2: Note the essential prime implicants**



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

Minimization Procedure

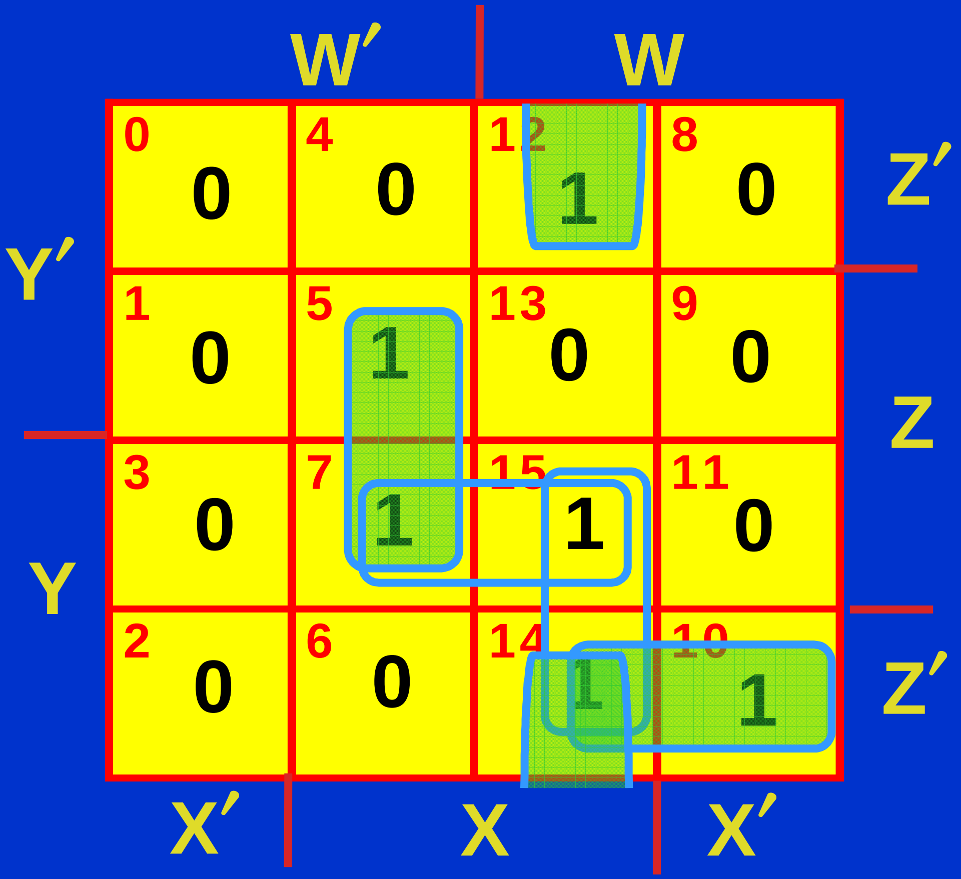
- **STEP 2: Note the essential prime implicants**



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

Minimization Procedure

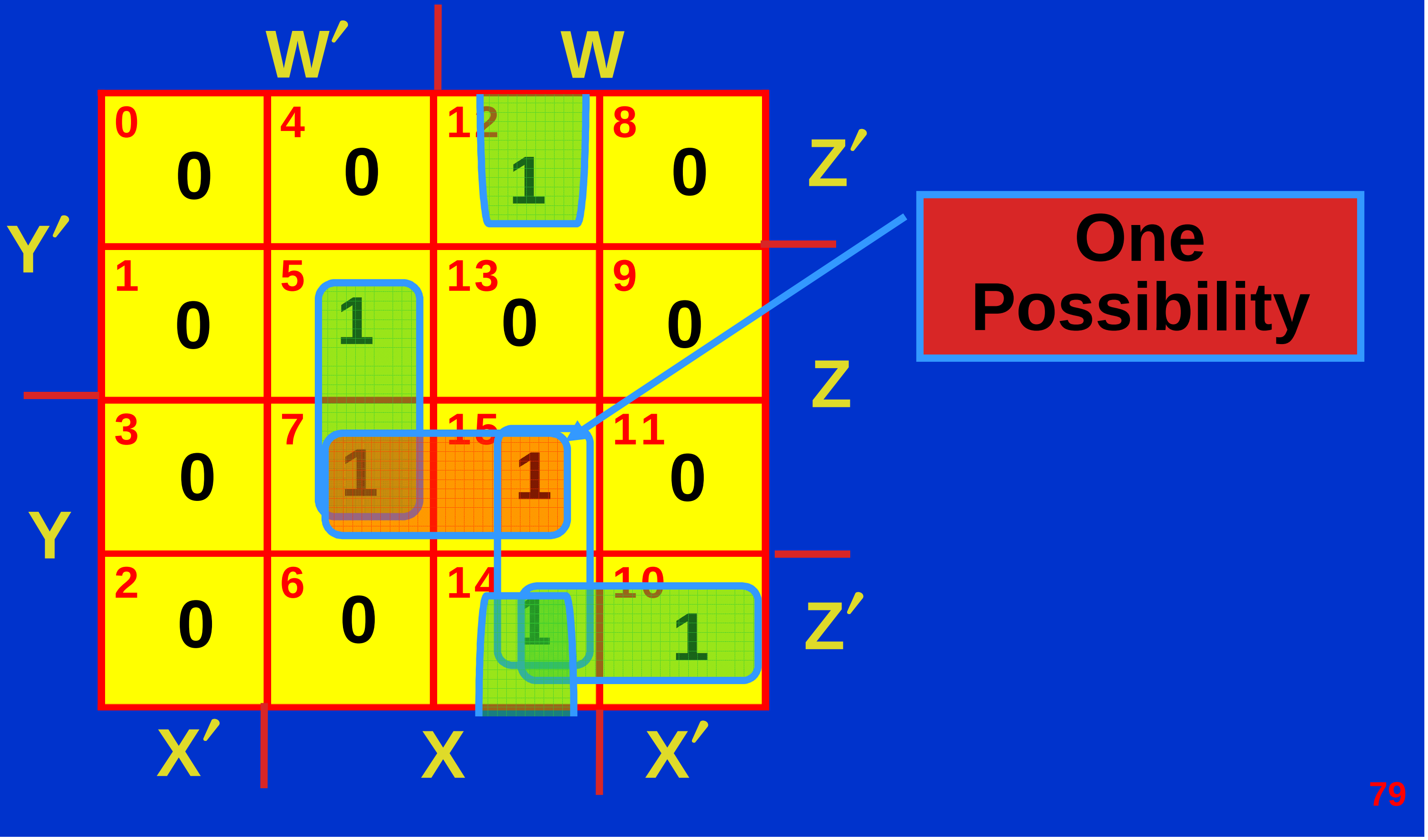
- **STEP 2: Note the essential prime implicants**



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

Minimization Procedure

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

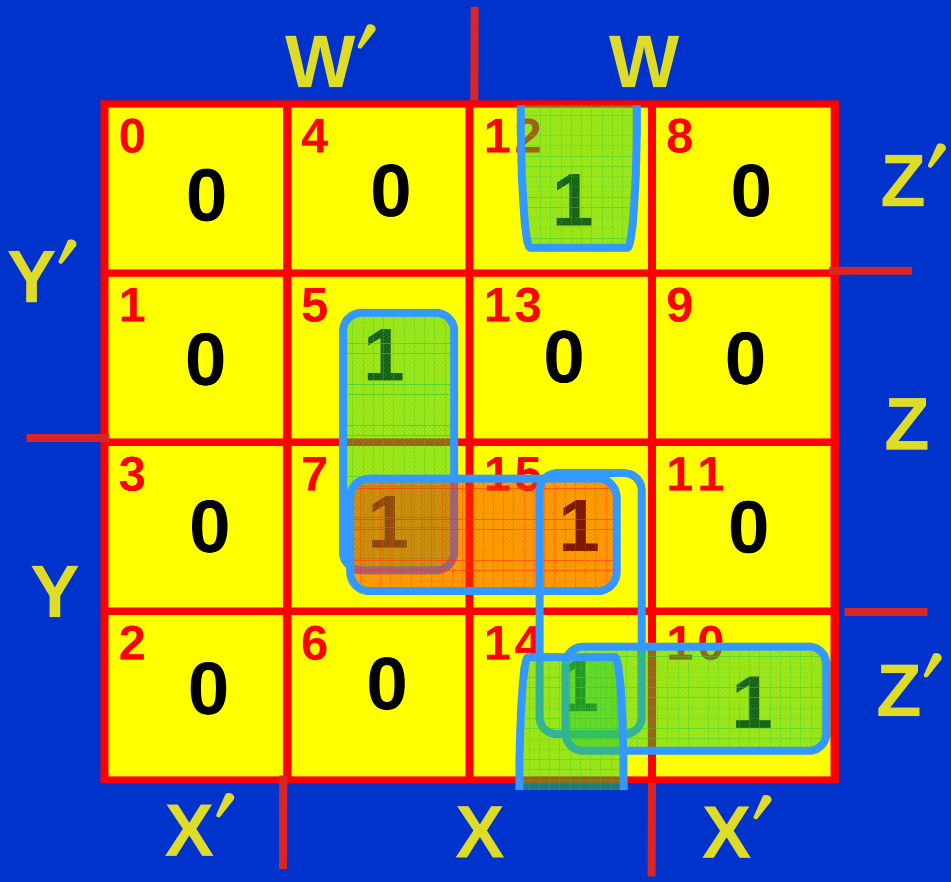


	W′	W′	W	W	
Y′	0 0	4 0	12 1	8 0	Z′
Y′	1 0	5 1	13 0	9 0	Z
Y	3 0	7 1	15 1	11 0	Z
Y	2 0	6 0	14 1	10 1	Z′
	X′	X	X	X′	

One Possibility

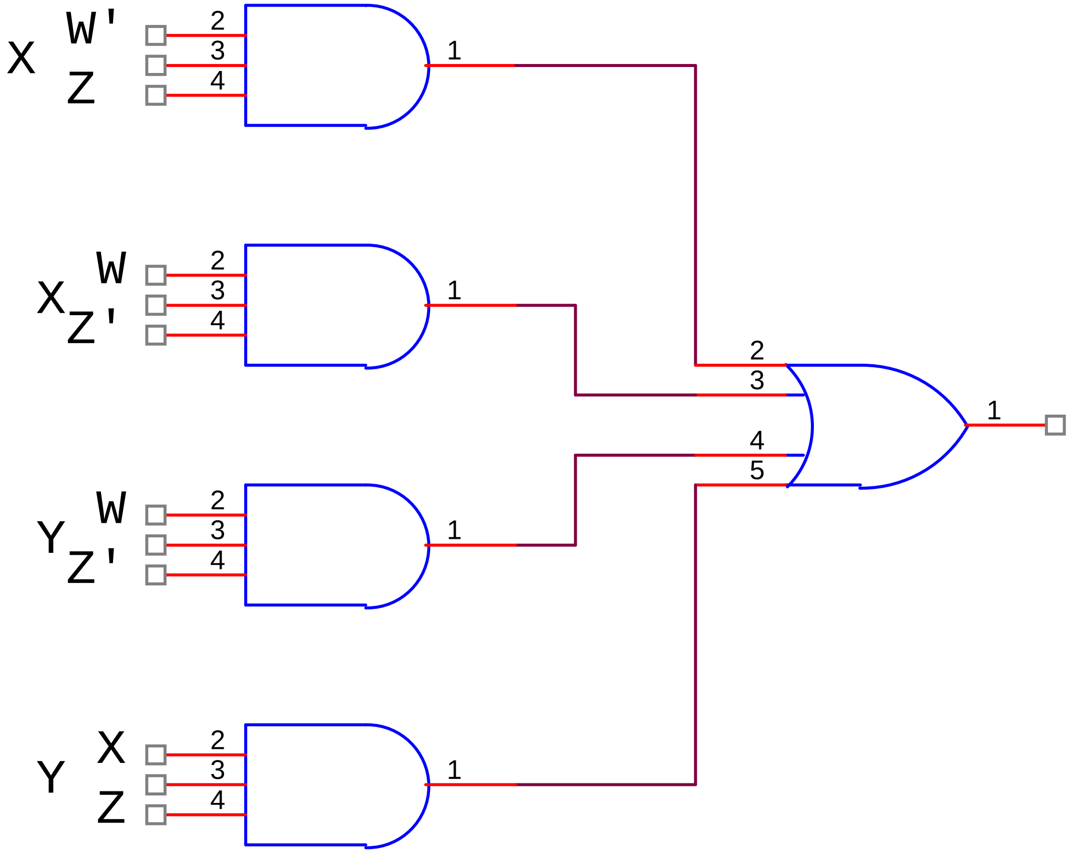
Minimization Procedure

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



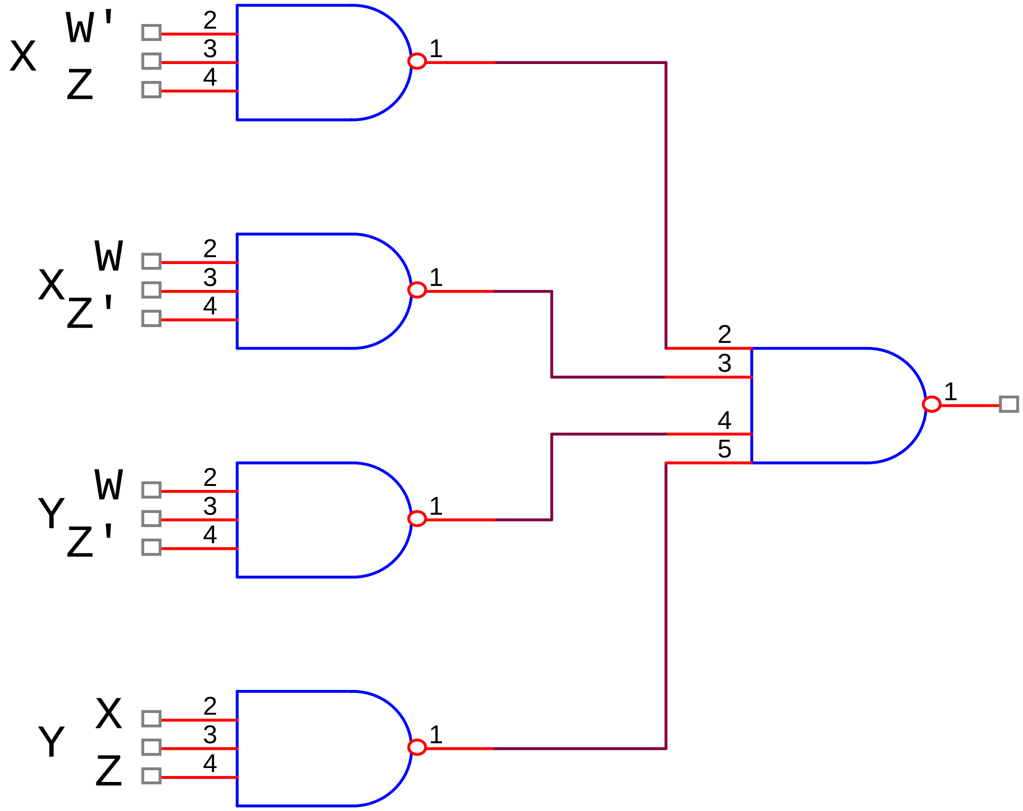
	W′	W′	W	W	
Y′	0 0	4 0	12 1	8 0	Z′
Y′	1 0	5 1	13 0	9 0	Z
Y	3 0	7 1	15 1	11 0	Z
Y	2 0	6 0	14 1	10 1	Z′
	X′	X	X	X′	

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



One possible circuit implementation (AND-OR):

COST is 16 inputs + 5 outputs = 21



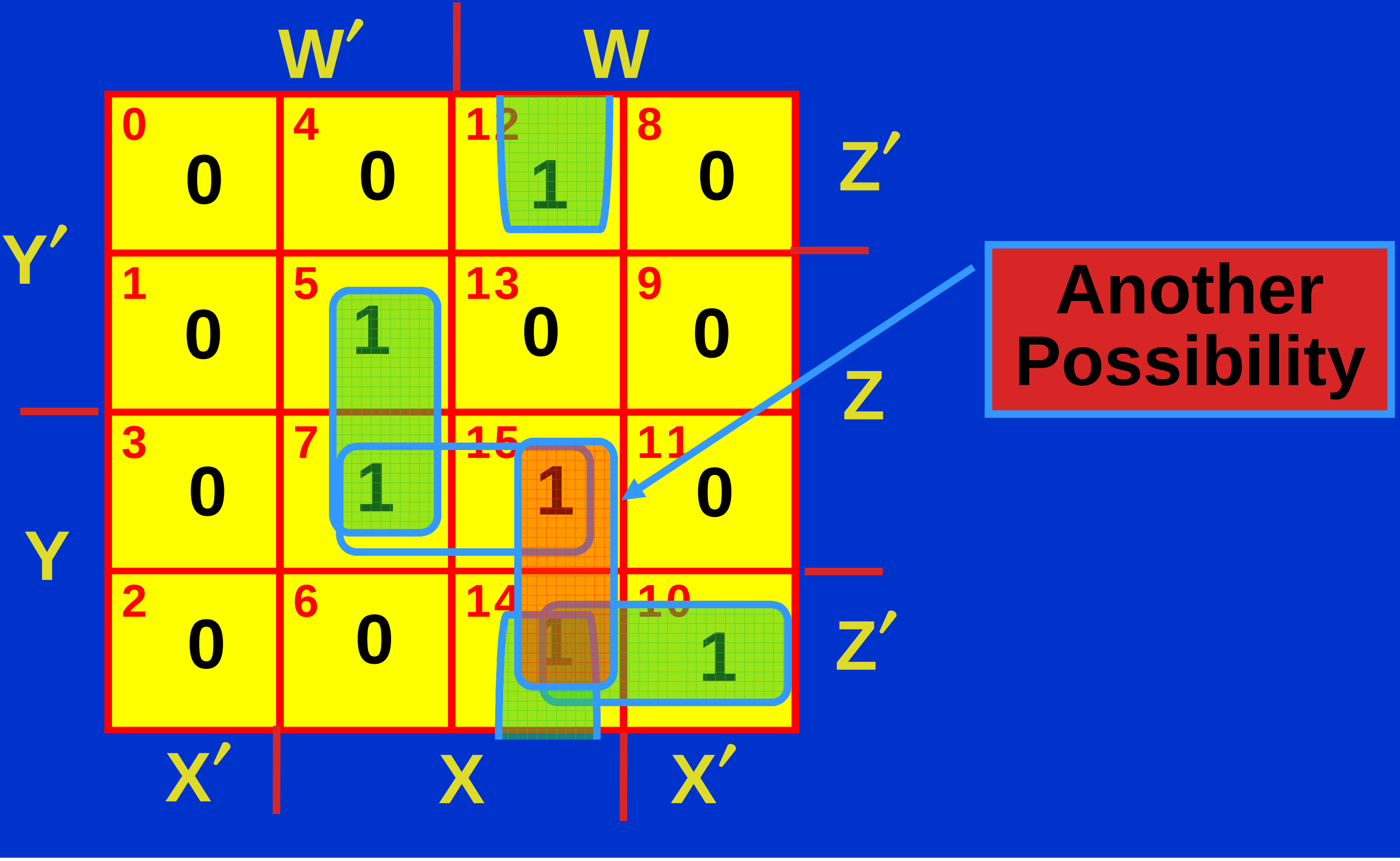
EQUIVALENT circuit implementation, obtained through graphical application of DeMorgan's Law

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**

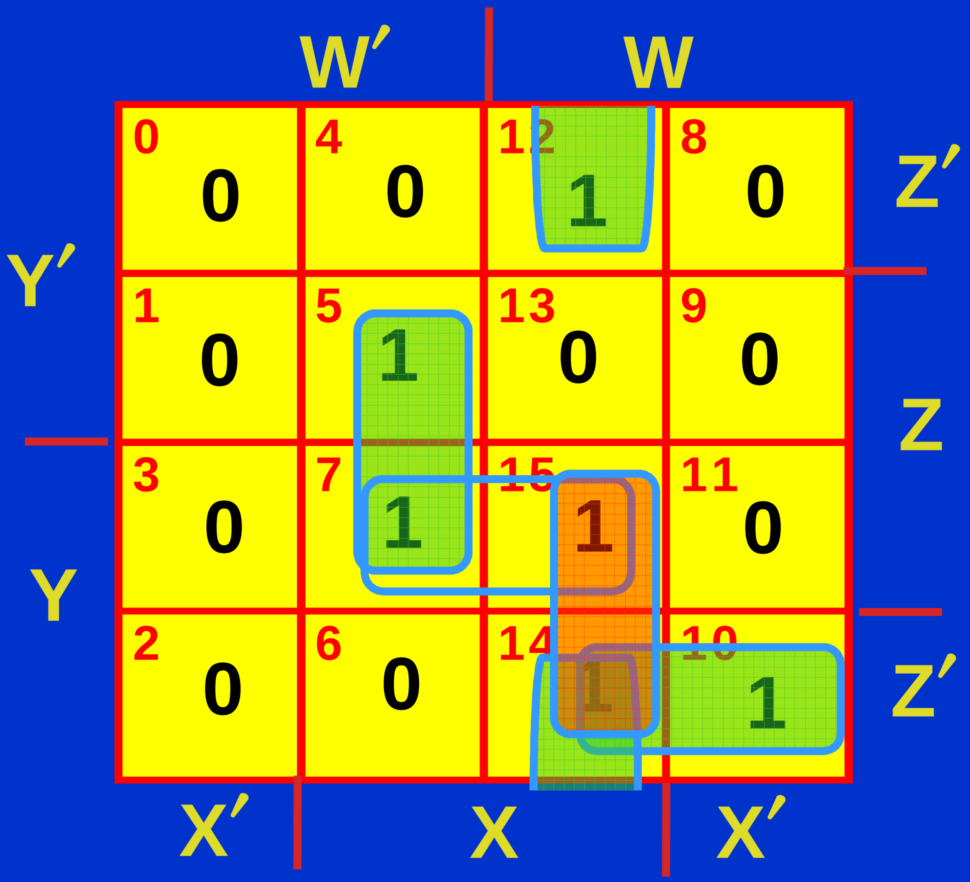


	W′	W′	W	W	
Y′	0 0	4 0	12 1	8 0	Z′
Y′	1 0	5 1	13 0	9 0	Z
Y	3 0	7 1	15 1	11 0	Z
Y	2 0	6 0	14 1	10 1	Z′
	X′	X	X	X′	

Another Possibility

Minimization Procedure

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



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

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

Clicker Quiz

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

1. The number of prime implicants is:

A. **1**

B. **2**

C. **3**

D. **4**

E. **5**

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

2. The number of essential prime implicants is:

A. **1**

B. **2**

C. **3**

D. **4**

E. **5**

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

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

A. **1**

B. **2**

C. **3**

D. **4**

E. **5**

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

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

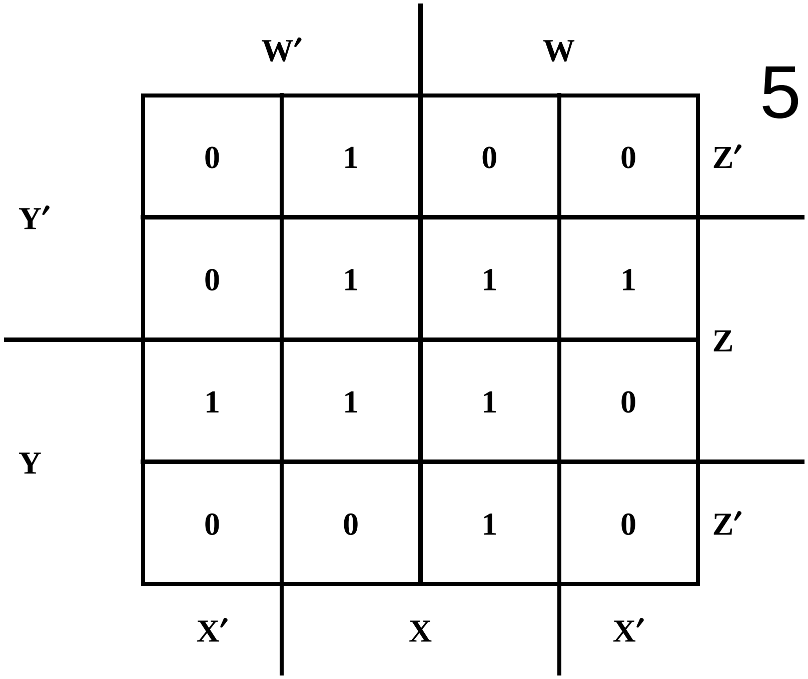
A. **1**

B. **2**

C. **3**

D. **4**

E. **5**



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

5. The ON SET for this function:

A. **$\sum_{W,X,Y,Z}(2,4,5,6,9,10,11,12)$**

B. **$\sum_{W,X,Y,Z}(3,4,5,7,9,13,14,15)$**

C. **$\sum_{W,X,Y,Z}(3,4,5,7,9,10,11,13)$**

D. **$\sum_{W,X,Y,Z}(2,4,5,6,9,13,14,15)$**

E. **none of the above**

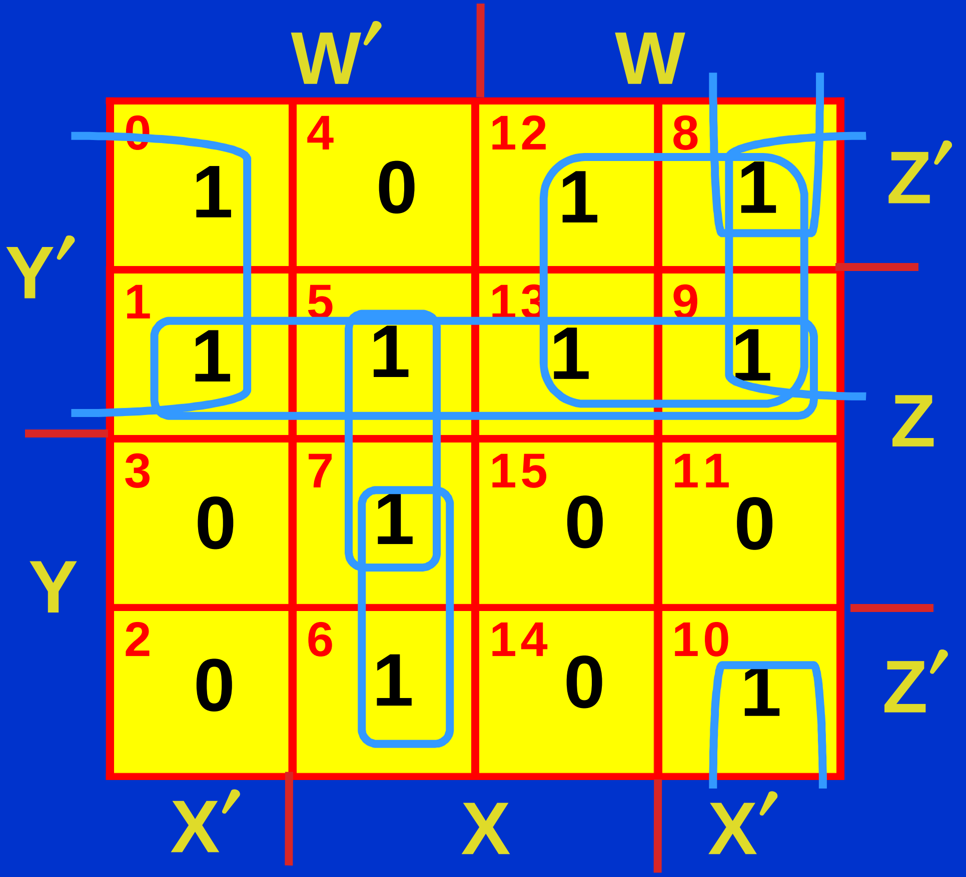
Minimization Procedure

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

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

Minimization Procedure

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

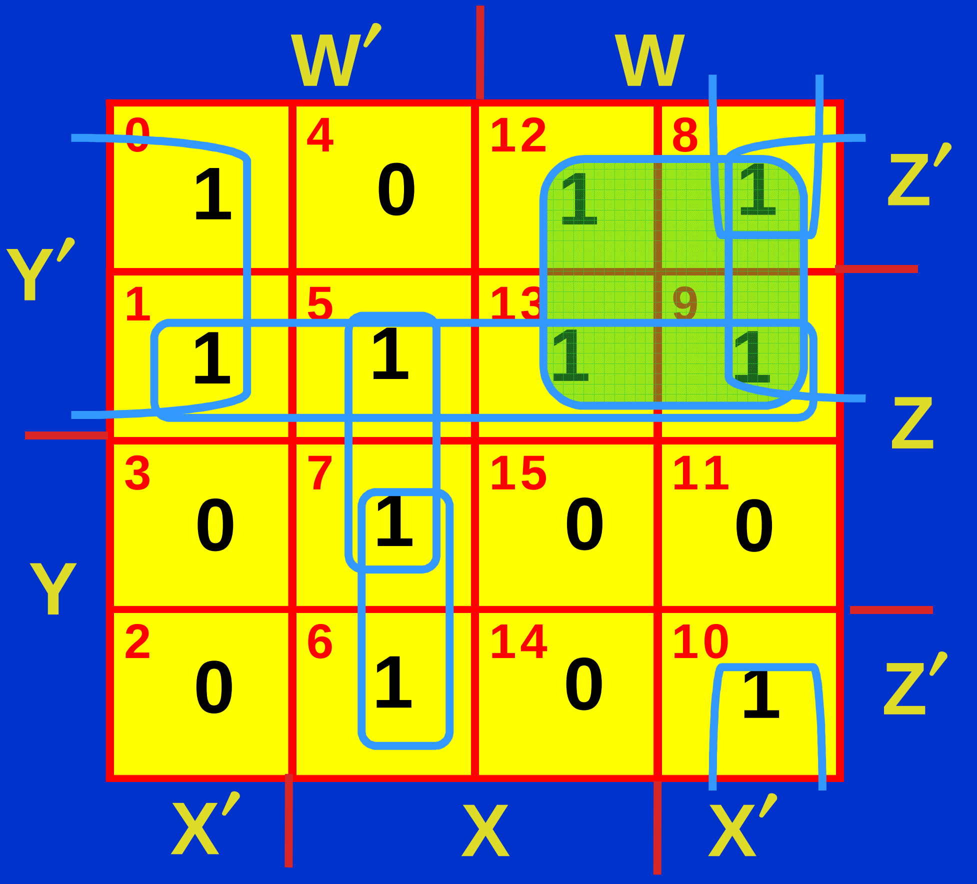


0 1	4 0	12 1	8 1
1 1	5 1	13 1	9 1
3 0	7 1	15 0	11 0
2 0	6 1	14 0	10 1

━━ prime implicants

Minimization Procedure

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

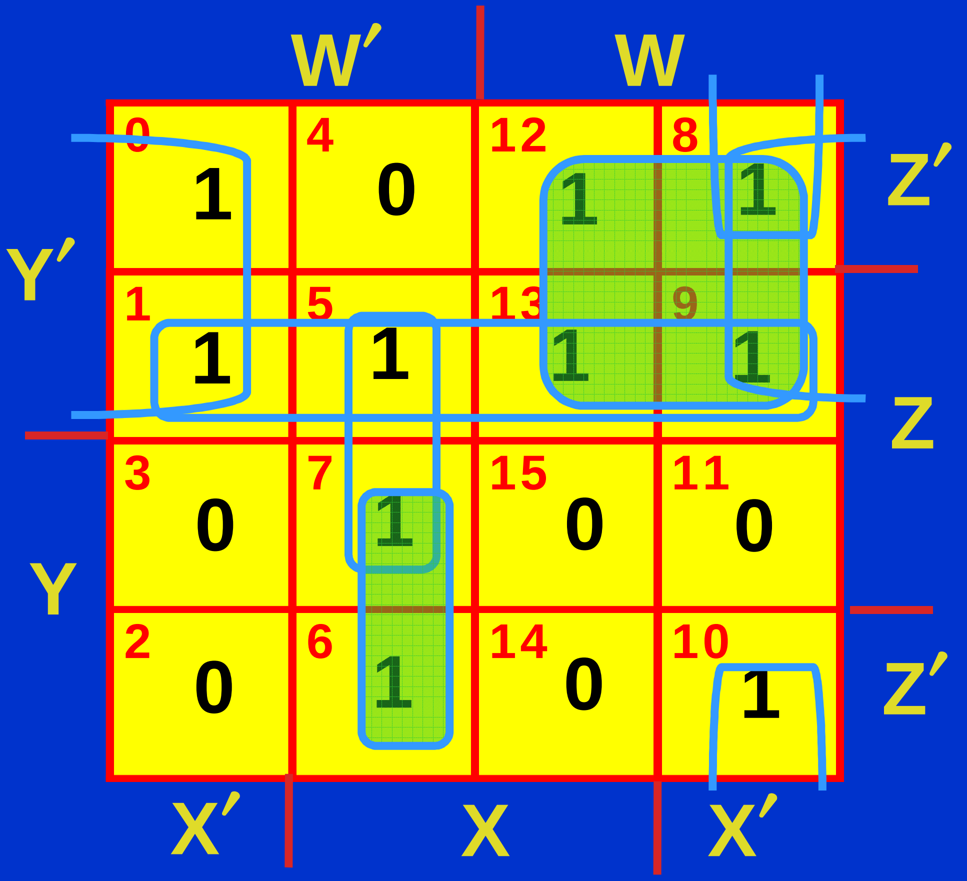


0 1	4 0	12 1	8 1
1 1	5 1	13 1	9 1
3 0	7 1	15 0	11 0
2 0	6 1	14 0	10 1

essential prime implicants

Minimization Procedure

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

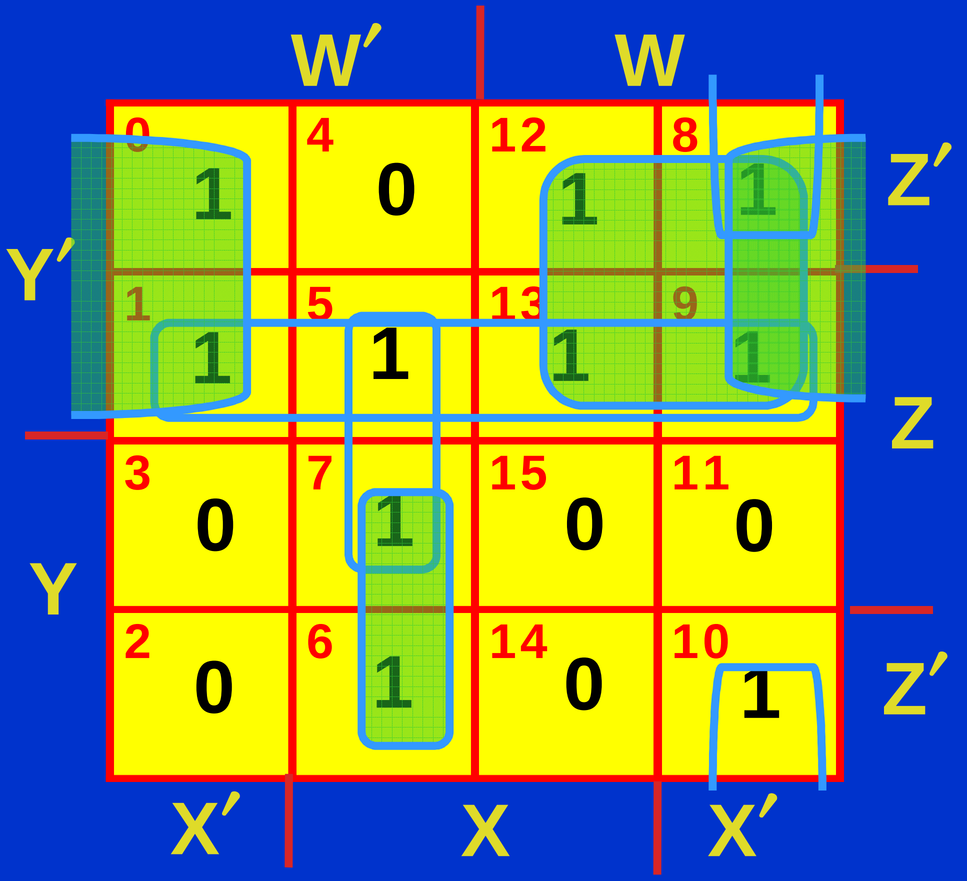


0 1	4 0	12 1	8 1
1 1	5 1	13 1	9 1
3 0	7 1	15 0	11 0
2 0	6 1	14 0	10 1

essential prime implicants

Minimization Procedure

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

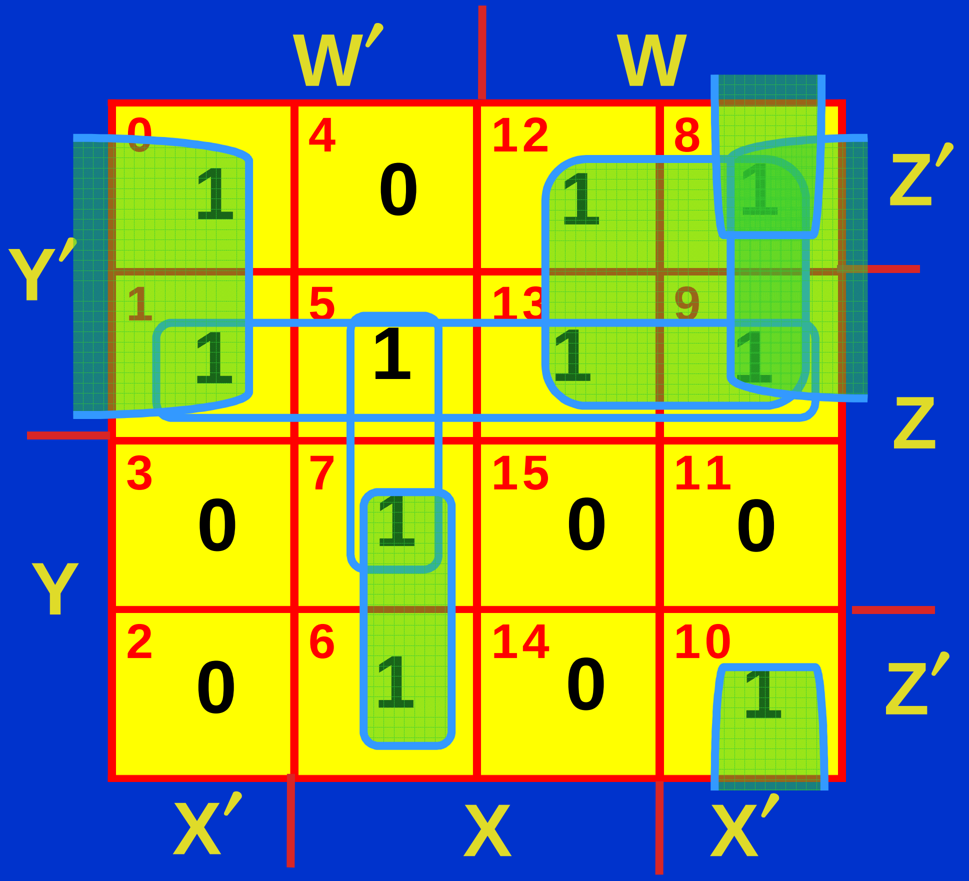


0 1	4 0	12 1	8 1
1 1	5 1	13 1	9 1
3 0	7 1	15 0	11 0
2 0	6 1	14 0	10 1

essential prime implicants

Minimization Procedure

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

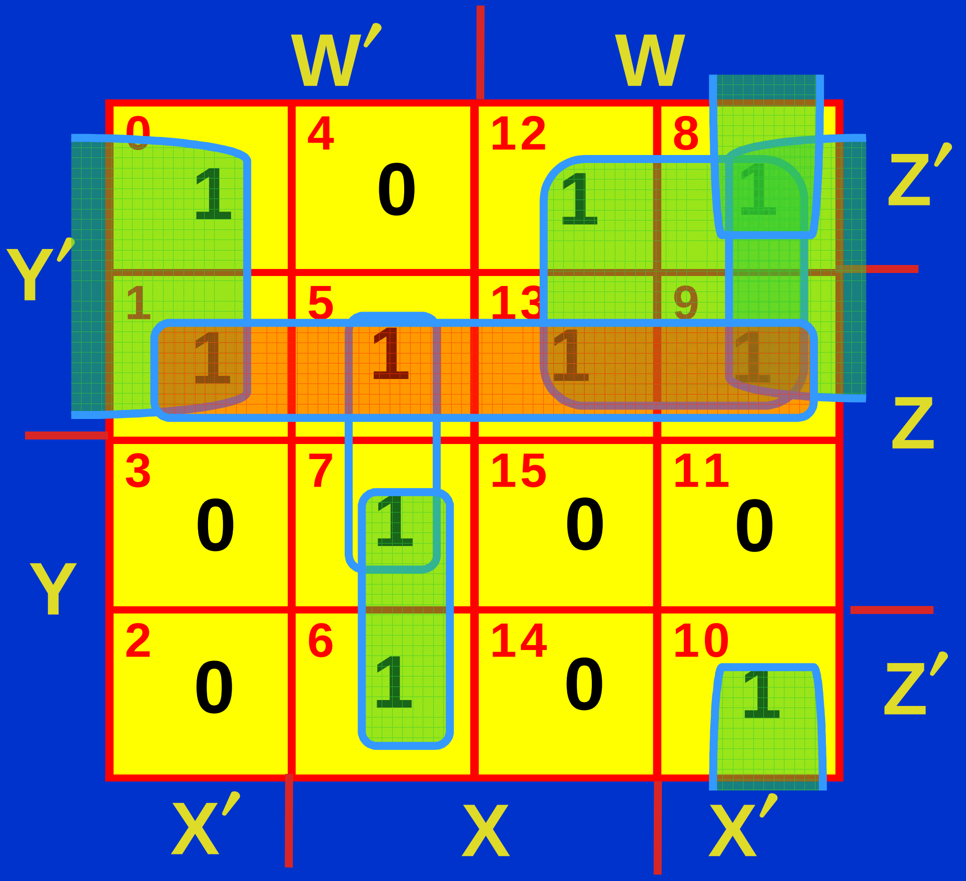


	W′	W′	W	W	
	X′	X	X	X′	
Y′	0: 1	4: 0	12: 1	8: 1	Z′
Y′	1: 1	5: 1	13: 1	9: 1	Z
Y	3: 0	7: 1	15: 0	11: 0	Z
Y	2: 0	6: 1	14: 0	10: 1	Z′

essential prime implicants

Minimization Procedure

- <u>Exercise</u>: Find a minimal sum-of-products expression for the function mapped below

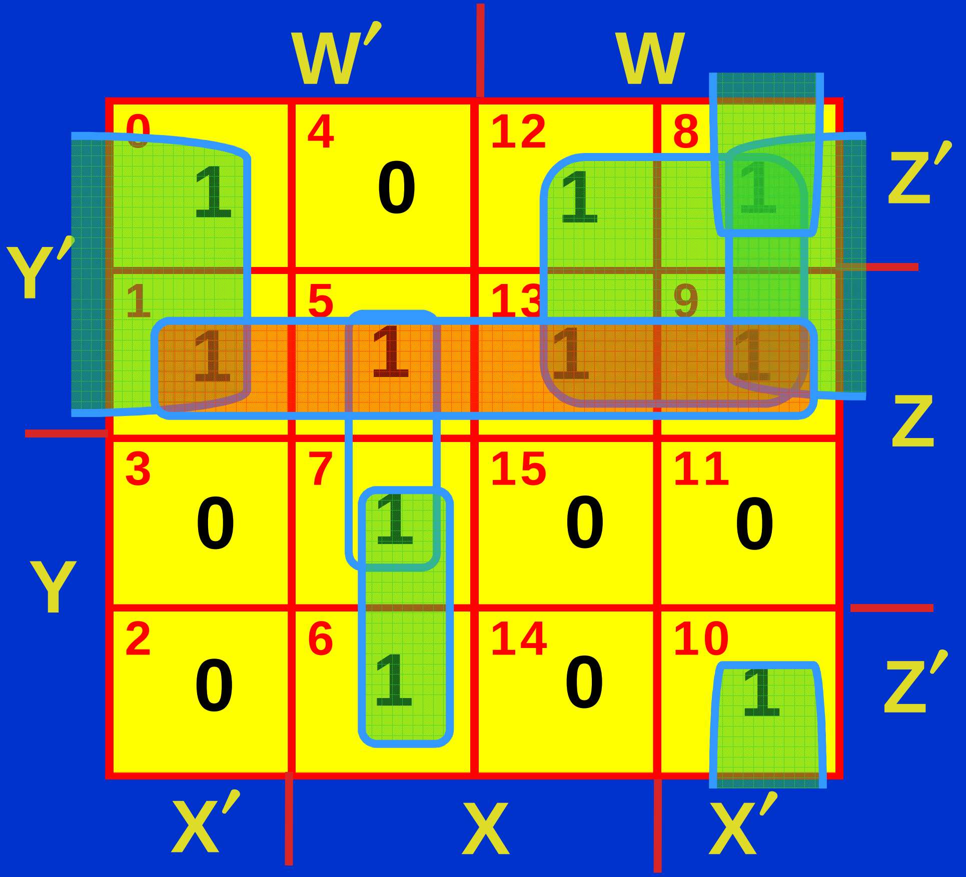


0 1	4 0	12 1	8 1
1 1	5 1	13 1	9 1
3 0	7 1	15 0	11 0
2 0	6 1	14 0	10 1

non-essential prime implicant needed to cover function

Minimization Procedure

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



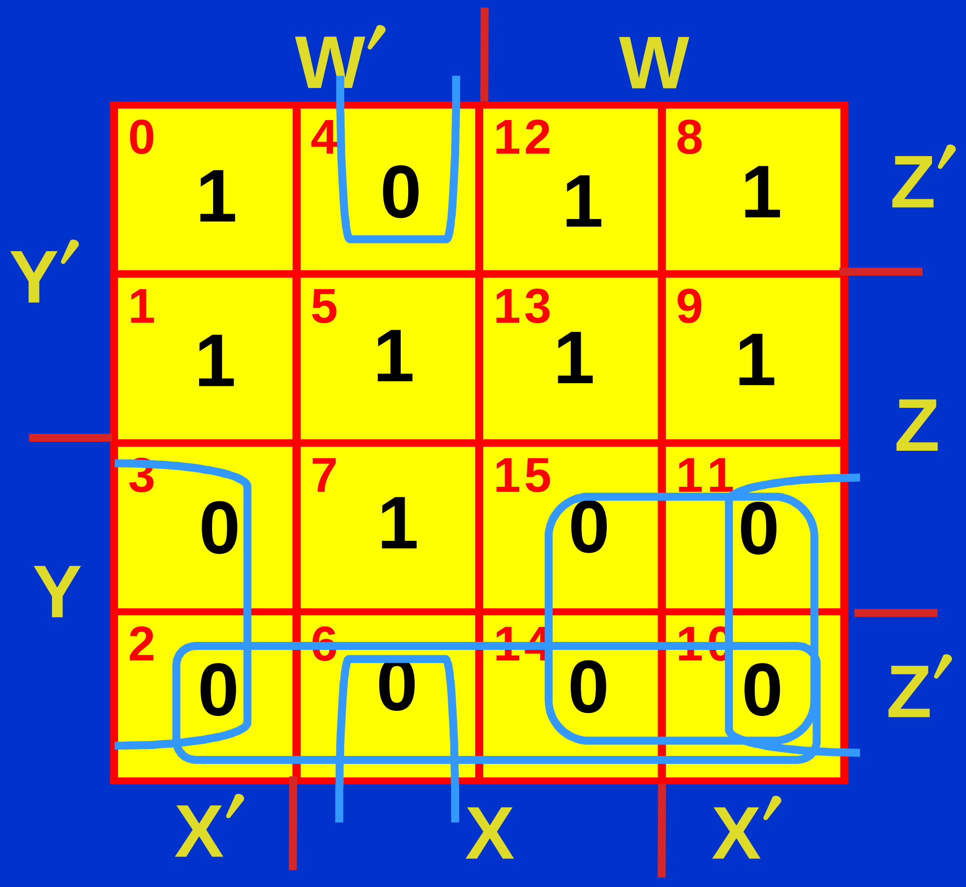
	W′		W		
Y′	0 1	4 0	12 1	8 1	Z′
	1 1	5 1	13 1	9 1	Z
Y	3 0	7 1	15 0	11 0	
	2 0	6 1	14 0	10 1	Z′
	X′	X		X′	

f **(W,X,Y,Z) =**

W•Y′ + X′•Y′ + W•X′•Z′ + W′•X•Y + Y′•Z

Minimization Procedure

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

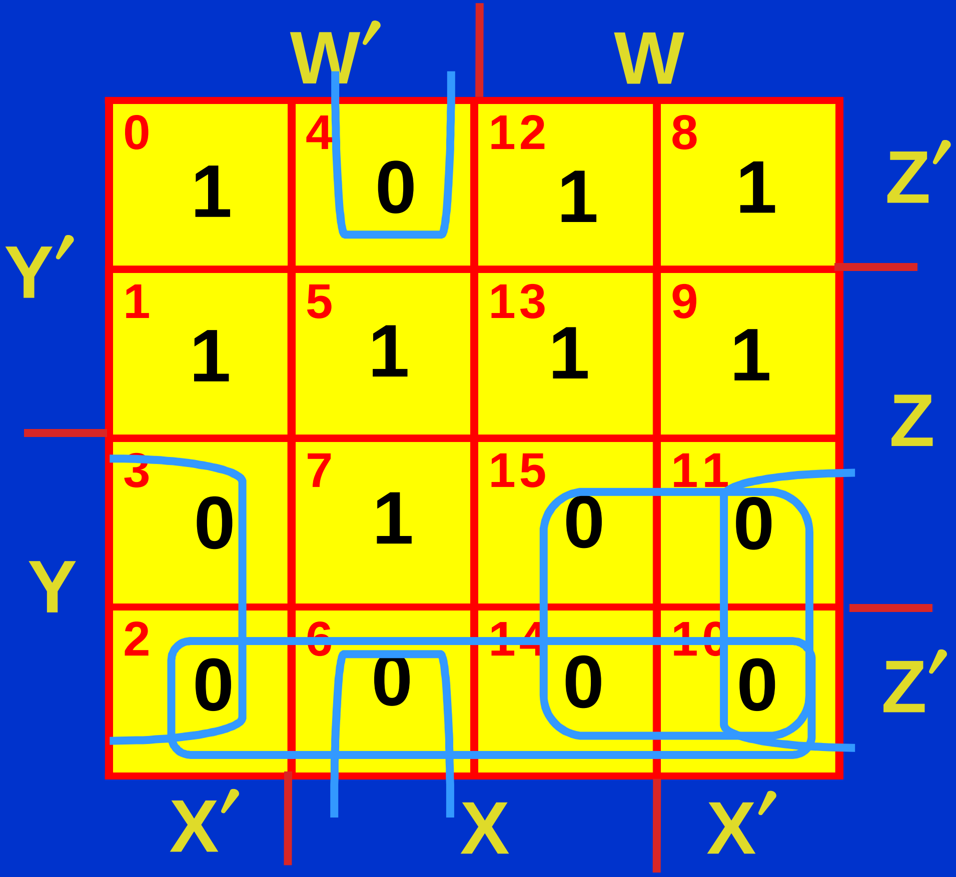


	W′	W′	W	W	
Y′	0 1	4 0	12 1	8 1	Z′
Y′	1 1	5 1	13 1	9 1	Z
Y	3 0	7 1	15 0	11 0	Z
Y	2 0	6 0	14 0	10 0	Z′
	X′	X	X	X′	

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

Minimization Procedure

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



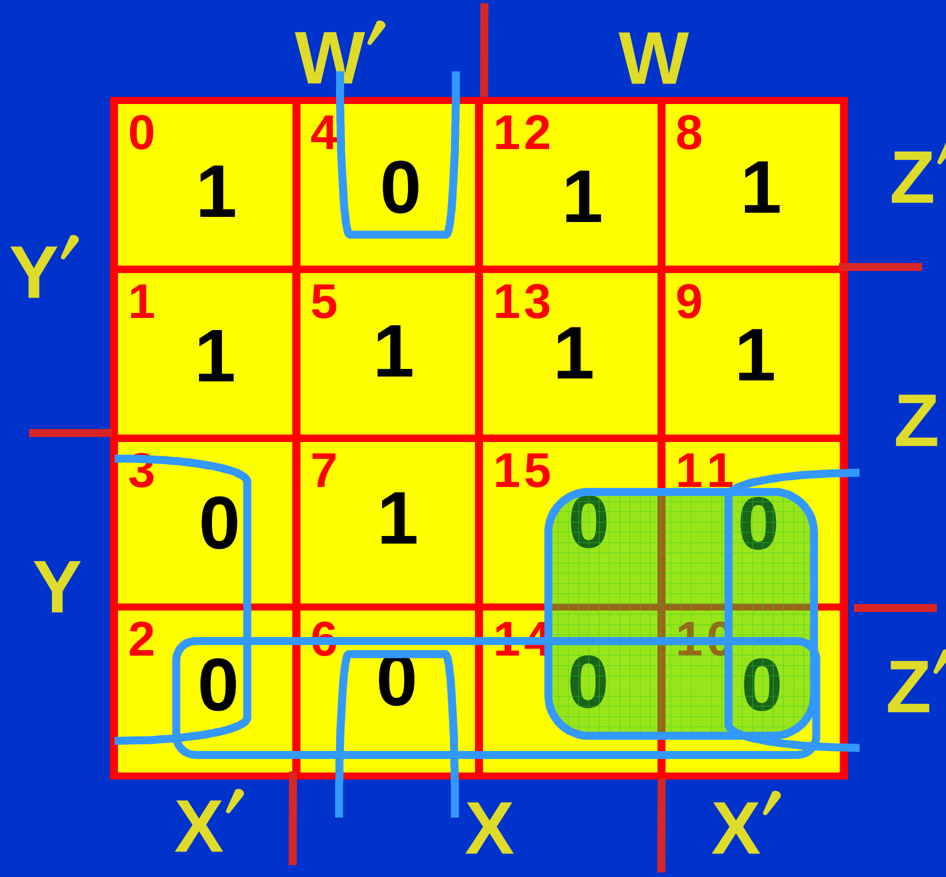
		W′	W′	W	W	
Y′		0 1	4 0	12 1	8 1	Z′
Y′		1 1	5 1	13 1	9 1	Z
Y		3 0	7 1	15 0	11 0	Z
Y		2 0	6 0	14 0	10 0	Z′
		X′	X	X	X′	

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?**



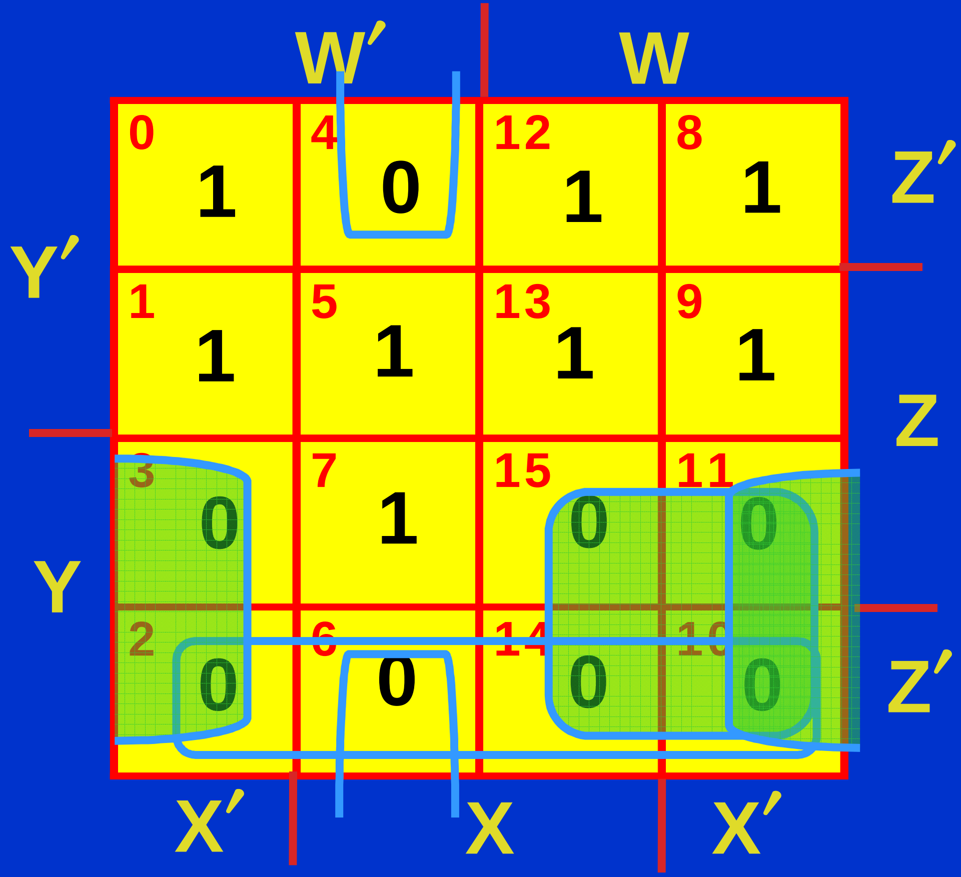
	W′		W		
Y′	0 1	4 0	12 1	8 1	Z′
Y′	1 1	5 1	13 1	9 1	Z
Y	3 0	7 1	15 0	11 0	Z
Y	2 0	6 0	14 0	10 0	Z′
	X′	X	X	X′	

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

Find essential prime implicants

Minimization Procedure

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



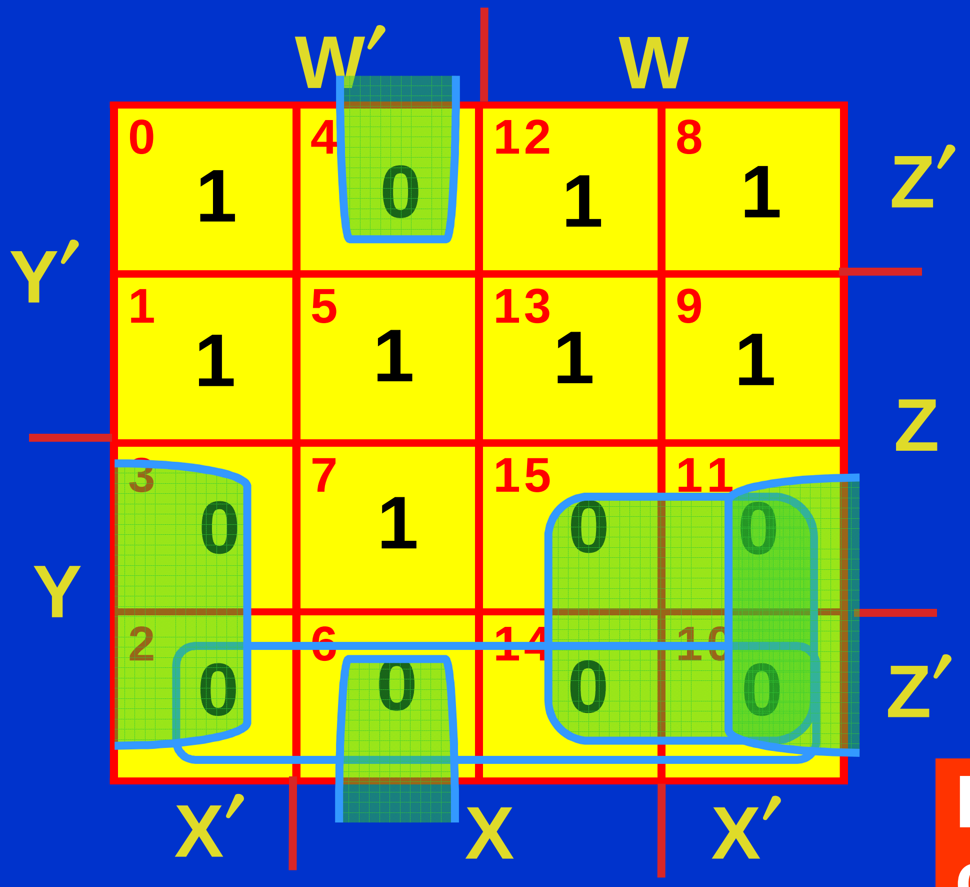
		W′	W′	W	W	
Y′		0 1	4 0	12 1	8 1	Z′
Y′		1 1	5 1	13 1	9 1	Z
Y		3 0	7 1	15 0	11 0	Z
Y		2 0	6 0	14 0	10 0	Z′
		X′	X	X	X′	

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

Find essential prime implicants

Minimization Procedure

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



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

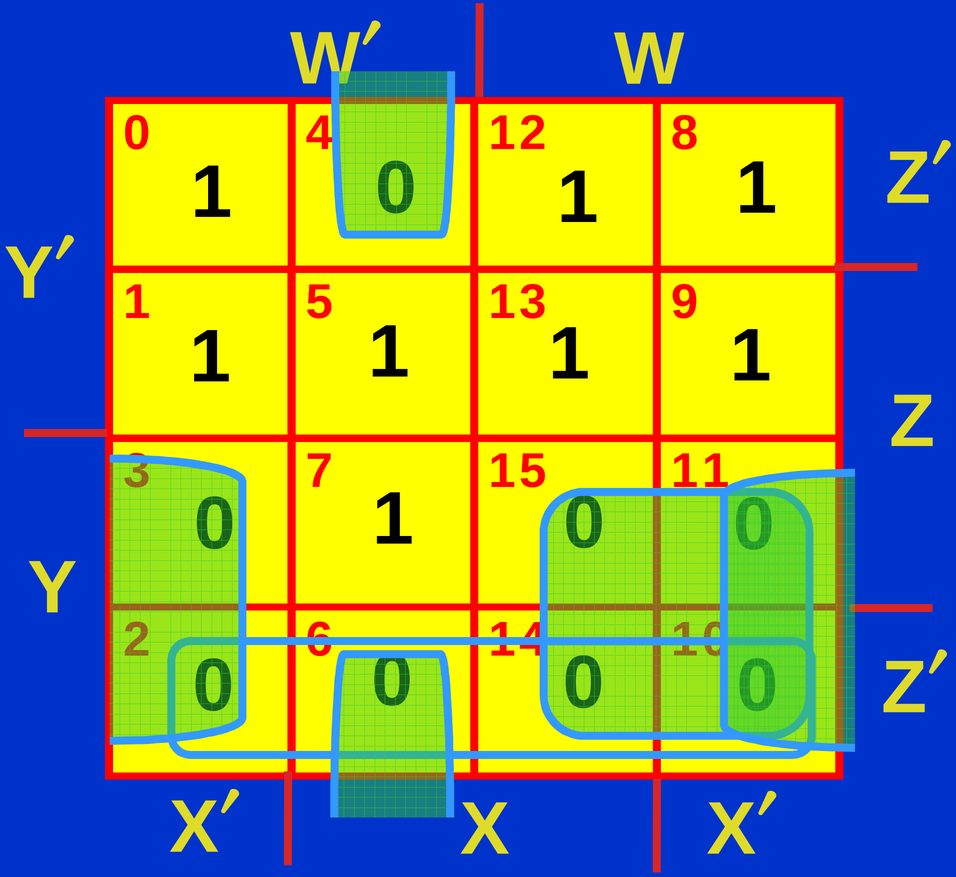
Group zeroes to get a minimum sum-of-products expression for *f ′*

Find essential prime implicants

Function is completely covered

Minimization Procedure

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

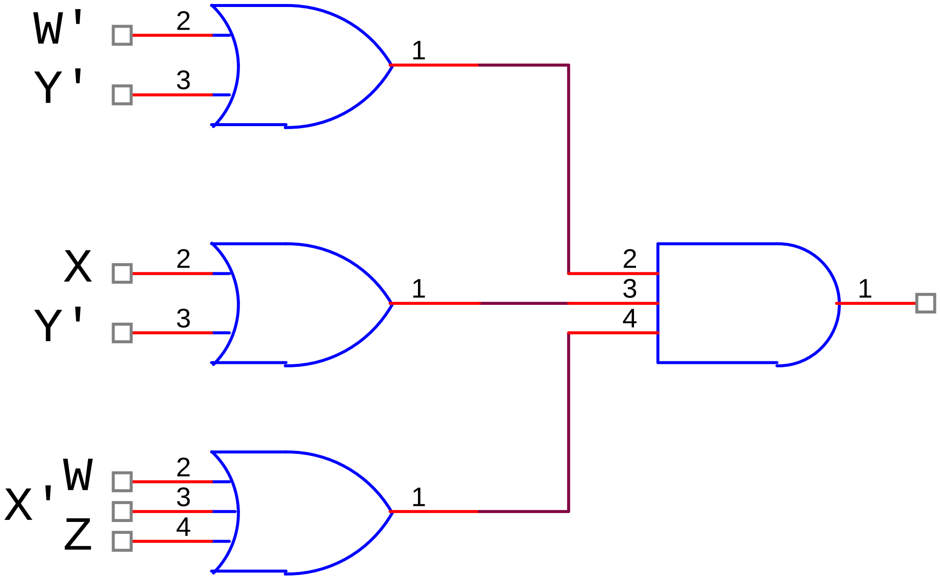


0 1	4 0	12 1	8 1
1 1	5 1	13 1	9 1
3 0	7 1	15 0	11 0
2 0	6 0	14 0	10 0

$f\,' = W\bullet Y + X'\bullet Y$
$+ W'\bullet X\bullet Z'$

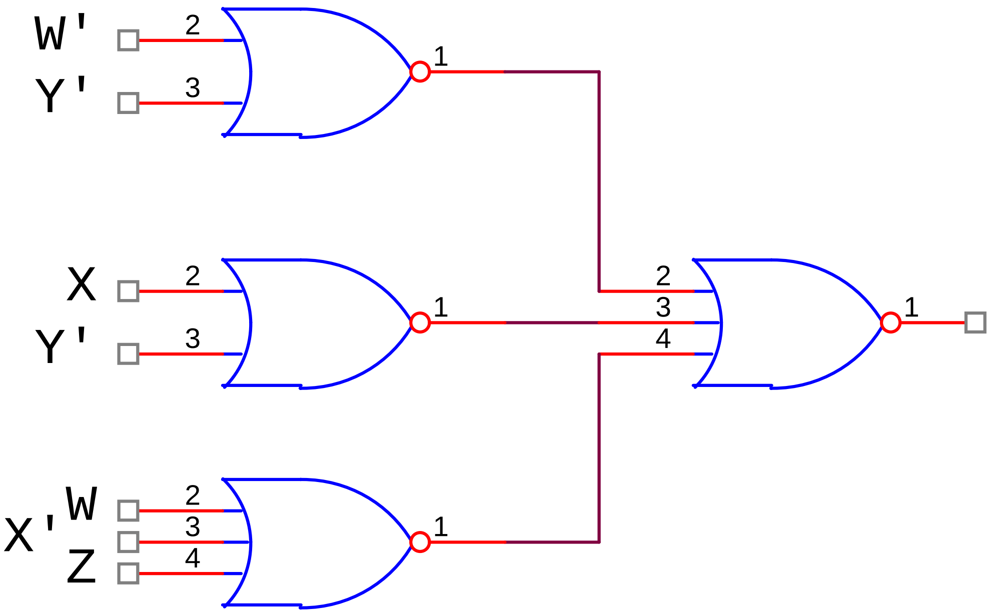
Apply DeMorgan's Law

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



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 ≡ NOR-NOR

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

More Minimization Examples

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

(a) using a single 7400 (quad 2-input NAND) plus a single 7410 (triple 3-input NAND)

(b) using a single 7403 (quad 2-input open-drain NAND)

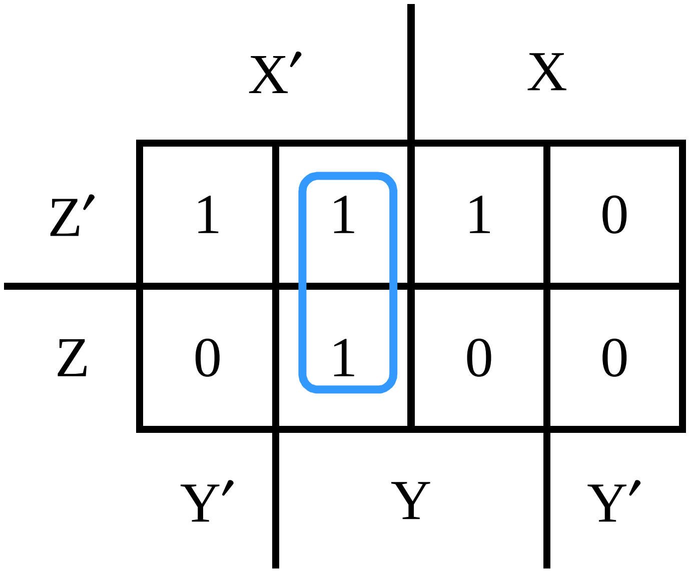
Key to Solution: The "NAND-Wired AND" configuration realizes the *complement* of the NAND-NAND configuration ➔ implement F′

Solution to (a)

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

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

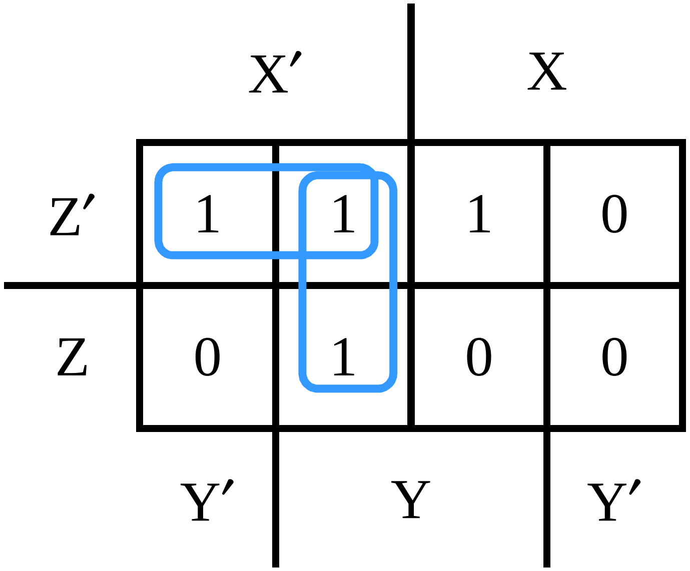
Solution to (a)



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

F(X,Y,Z) = X′•Y

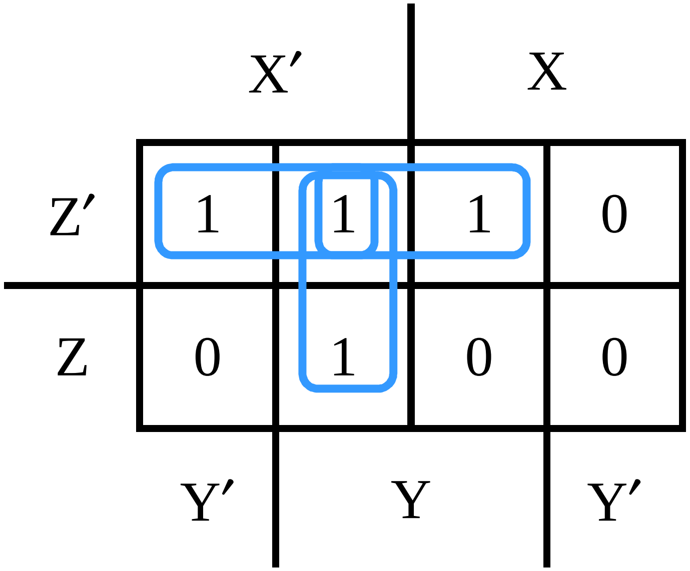
Solution to (a)



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

F(X,Y,Z) = X′•Y + X′•Z′

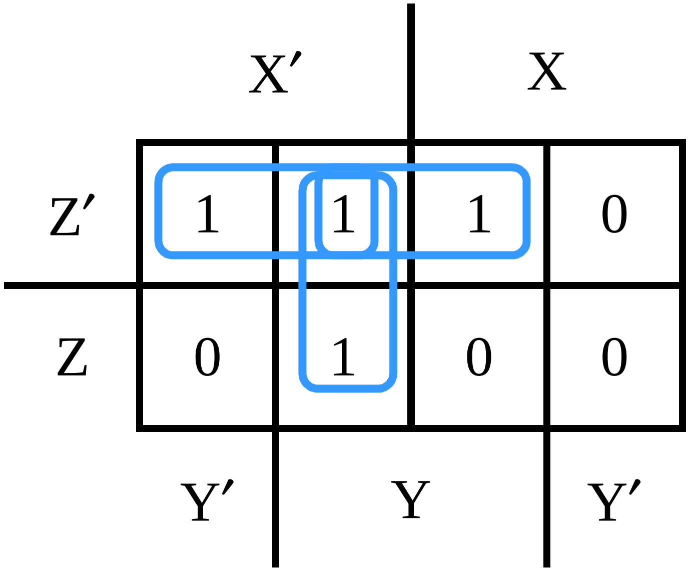
Solution to (a)



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

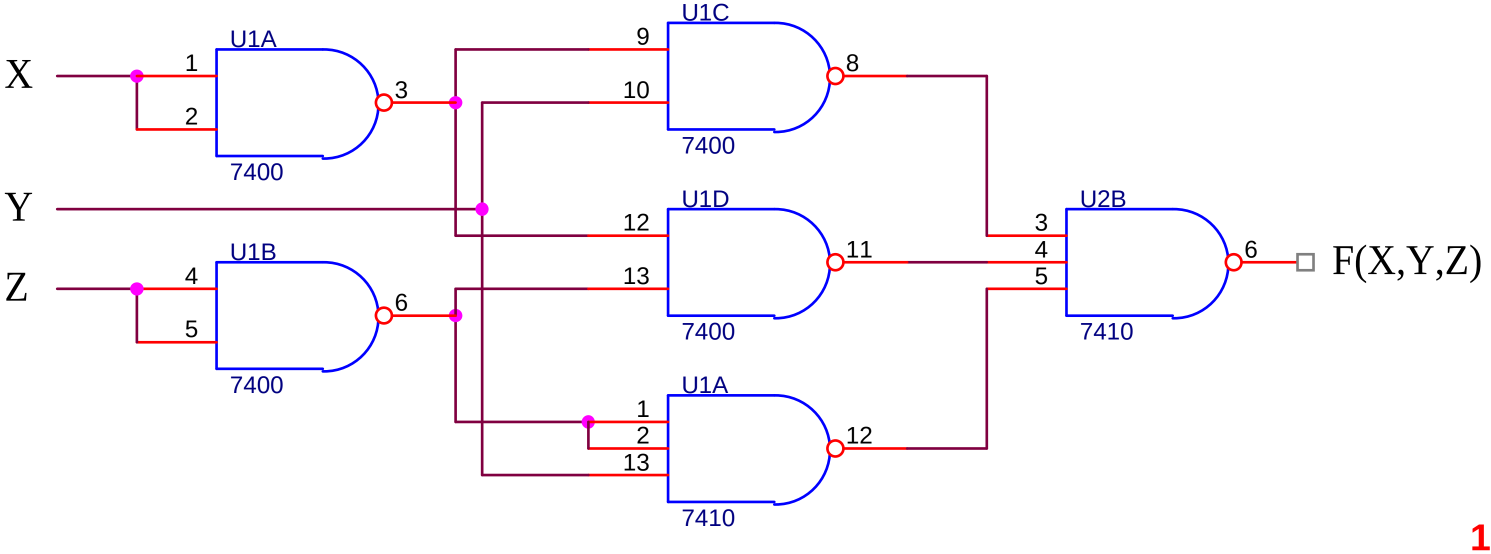
F(X,Y,Z) = X′•Y + X′•Z′ + Y•Z′

Solution to (a)



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

F(X,Y,Z) = X′•Y + X′•Z′ + Y•Z′

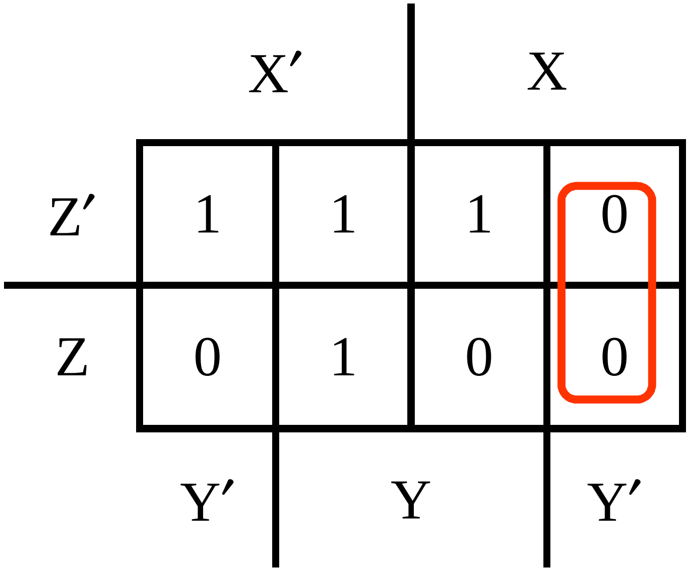


Solution to (b)

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

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

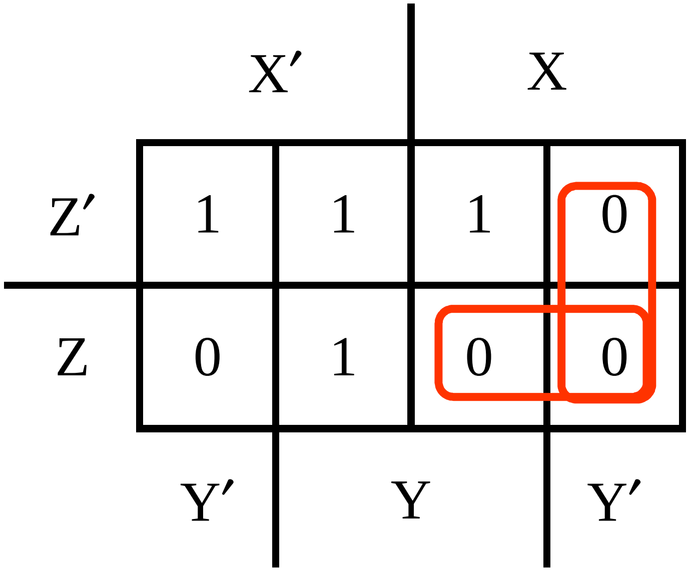
Solution to (b)



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

F′(X,Y,Z) = X•Y′

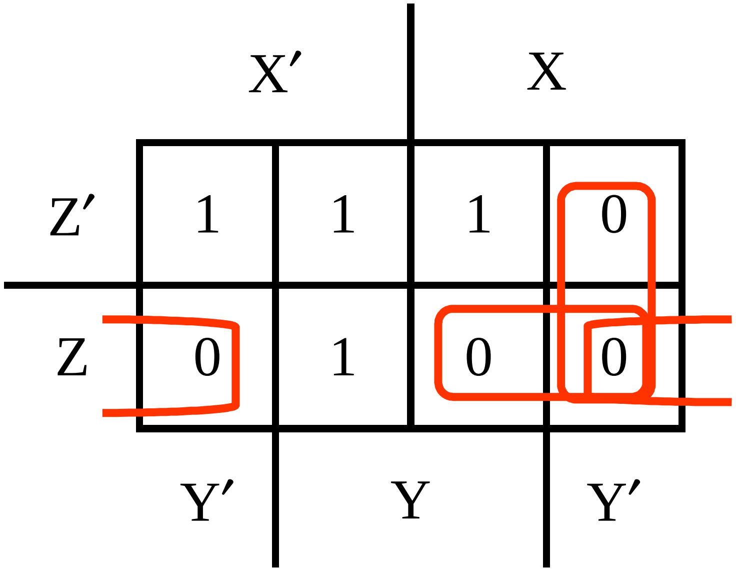
Solution to (b)



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

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

Solution to (b)



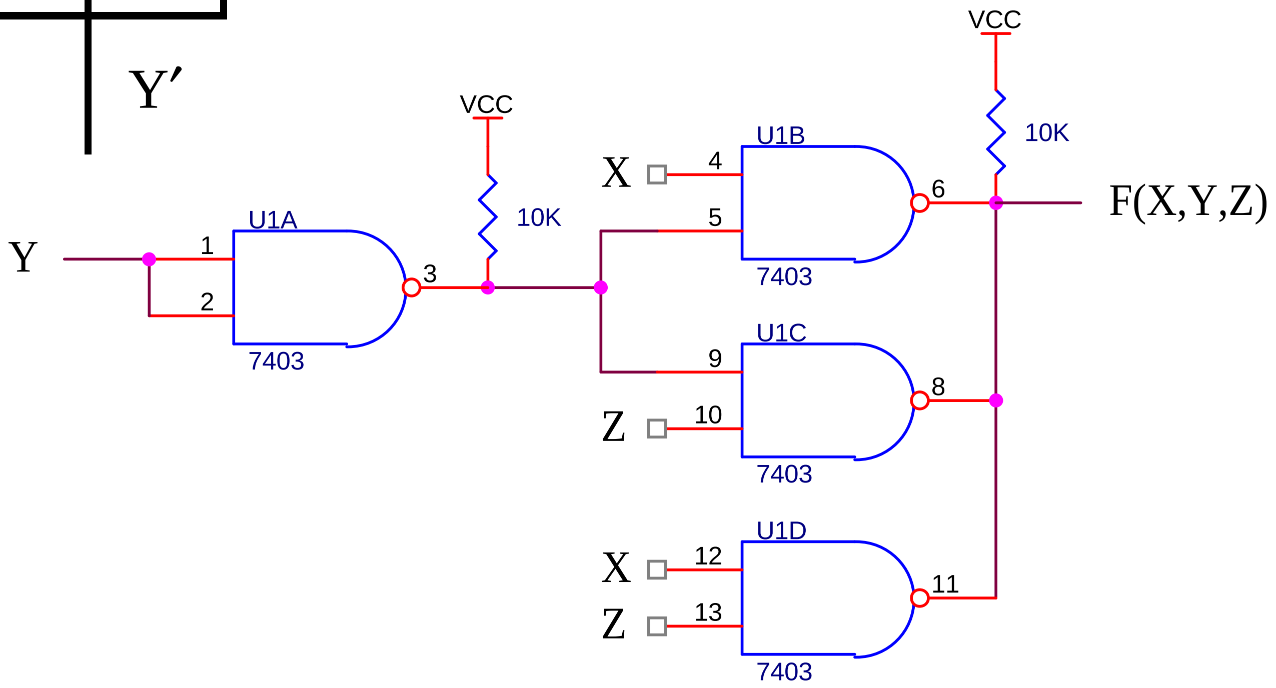
	X′		X	
Z′	1	1	1	0
Z	0	1	0	0
	Y′	Y		Y′

F′(X,Y,Z) = X•Y′ + X•Z + Y′•Z

Solution to (b)

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

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



“Conversion” Example

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

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

→ F′(X,Y,Z) = ______________________________ → Map __________

F(X,Y,Z) = ______________ →

F′(X,Y,Z) in minimal POS form

= ________________________________

“Conversion” Example

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

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

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

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

F(X,Y,Z) = ____________ →

F′(X,Y,Z) in minimal POS form

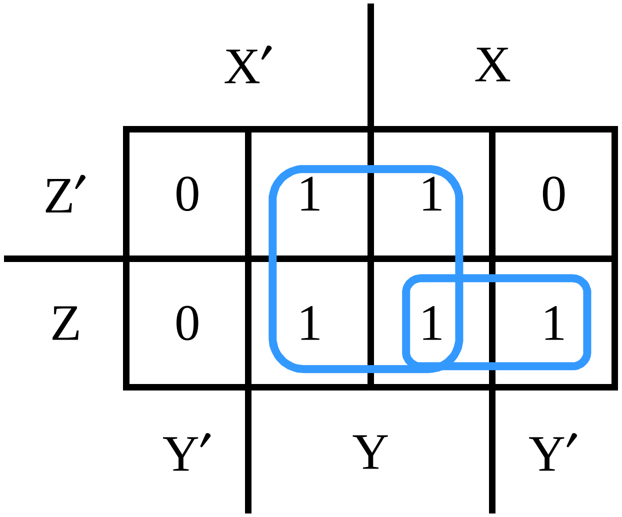
= ______________________________

“Conversion” Example

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

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

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



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

F(X,Y,Z) = Y + X•Z ➔

F′(X,Y,Z) in minimal POS form

= Y′ • (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

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

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

On Set

d-Set

BCD Inequality Detector Example

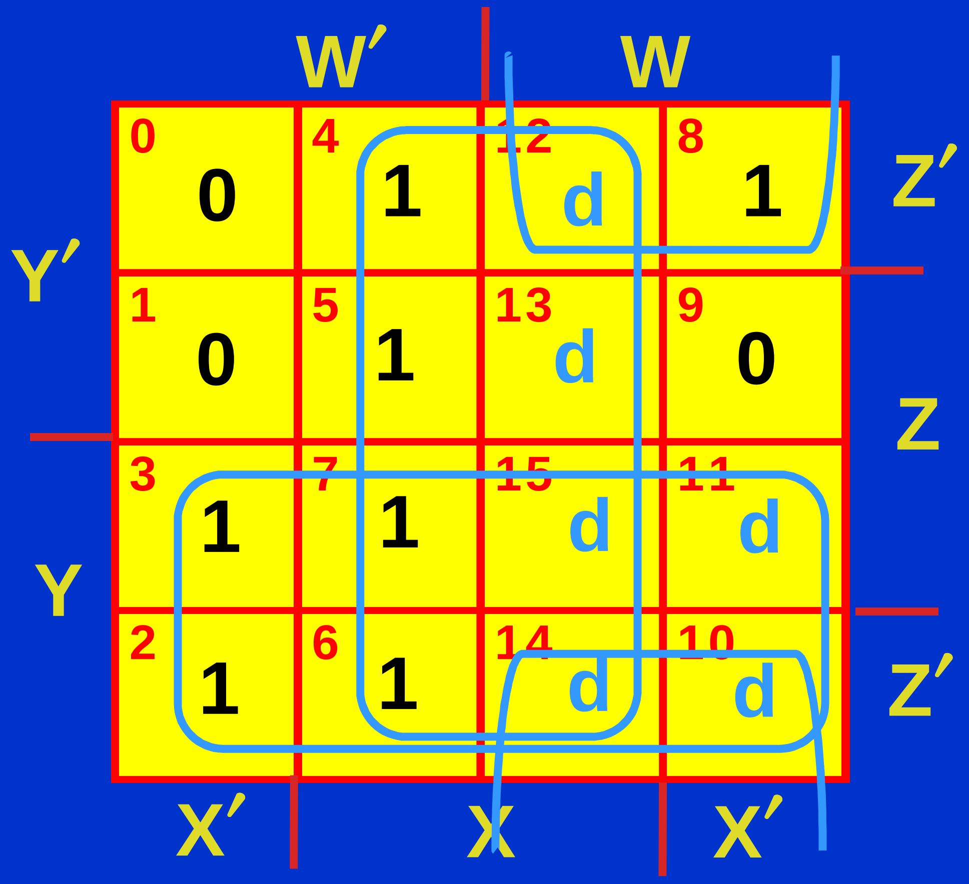
N_{10}	W X Y Z	F(W,X,Y,Z)
0	0 0 0 0	0
1	0 0 0 1	0
2	0 0 1 0	1
3	0 0 1 1	1
4	0 1 0 0	1
5	0 1 0 1	1
6	0 1 1 0	1
7	0 1 1 1	1
8	1 0 0 0	1
9	1 0 0 1	0

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

Some hardware description languages provide a means for the designer to specify don't care inputs

BCD Inequality Detector Example

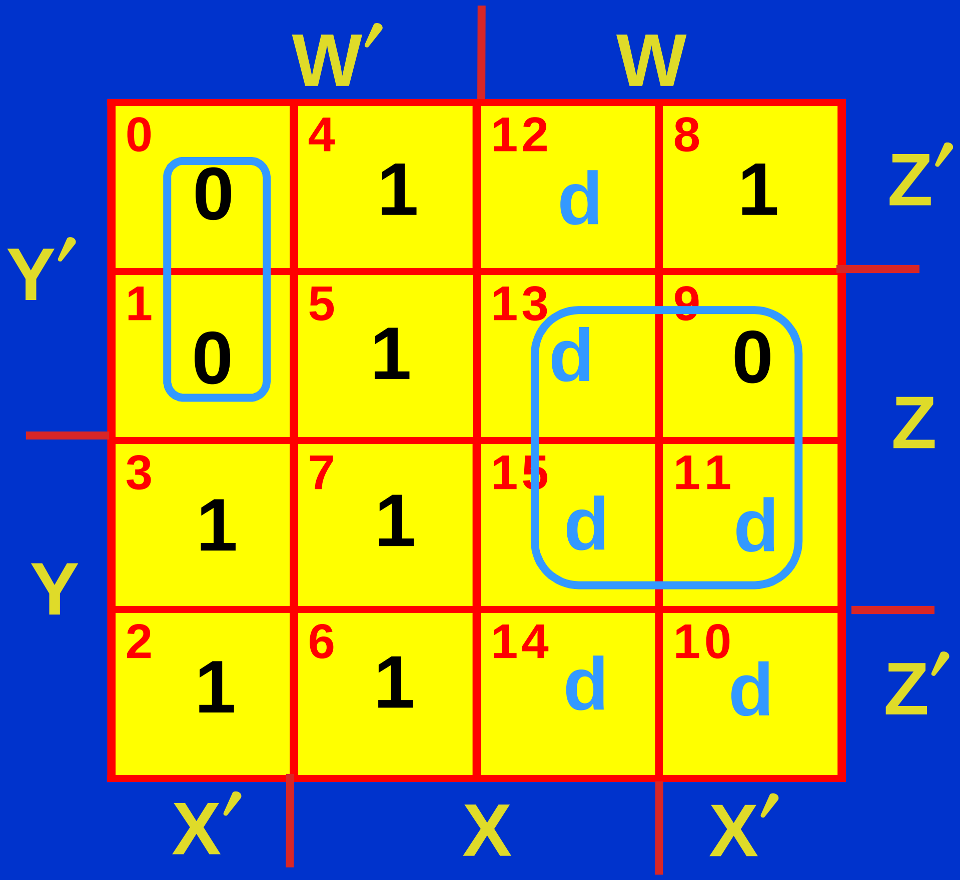


	W′	W′	W	W	
Y′	0 0	4 1	12 d	8 1	Z′
Y′	1 0	5 1	13 d	9 0	Z
Y	3 1	7 1	15 d	11 d	Z
Y	2 1	6 1	14 d	10 d	Z′
	X′	X	X	X′	

Minimum SP: f(W,X,Y,Z) = X + Y + W•Z′

Cost: 5 gate inputs + 2 gate outputs = 7

BCD Inequality Detector Example



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

Minimum PS:

$$f'(W,X,Y,Z) =$$

$$W \bullet Z + W' \bullet X' \bullet Y'$$

$$\rightarrow f(W,X,Y,Z) =$$

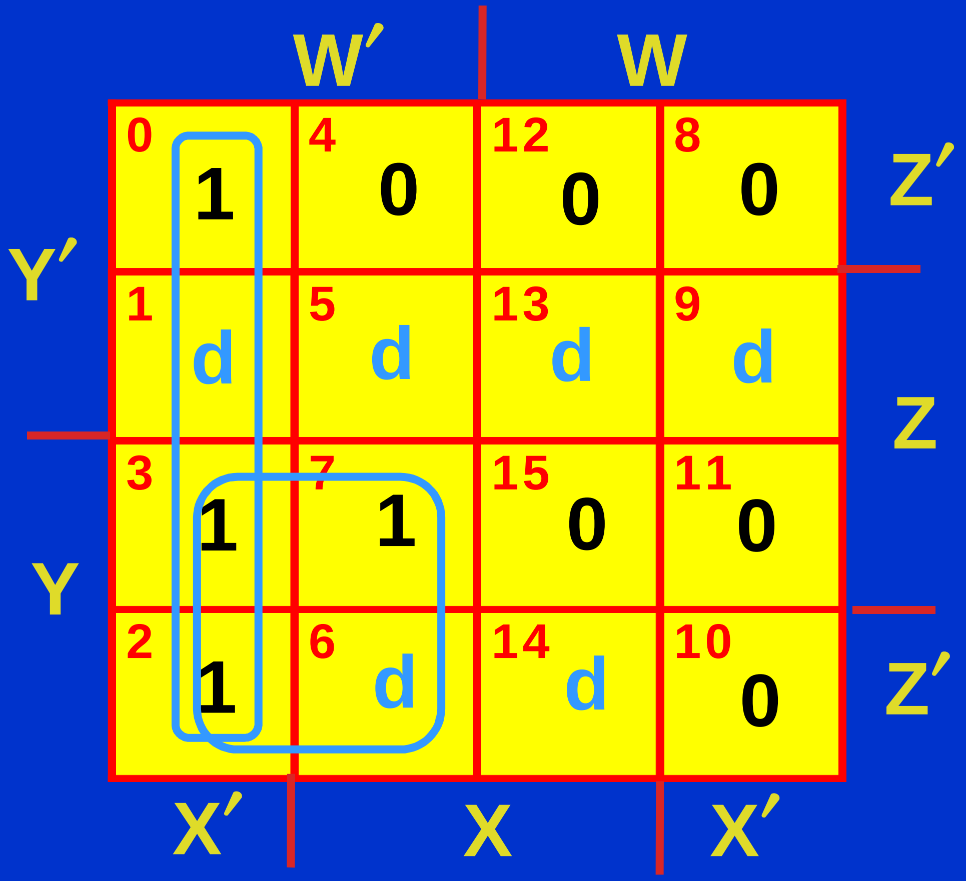
$$(W' + Z') \bullet (W + X + Y)$$

Cost: 7 gate inputs + 3 gate outputs = 10

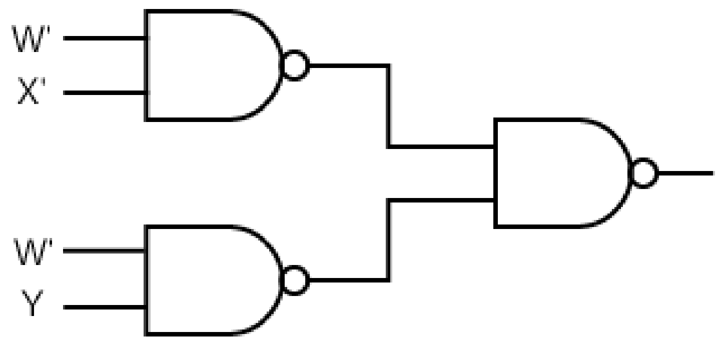
<u>Conclusion</u>: The SP implementation *costs less*

Incompletely Specified Functions

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



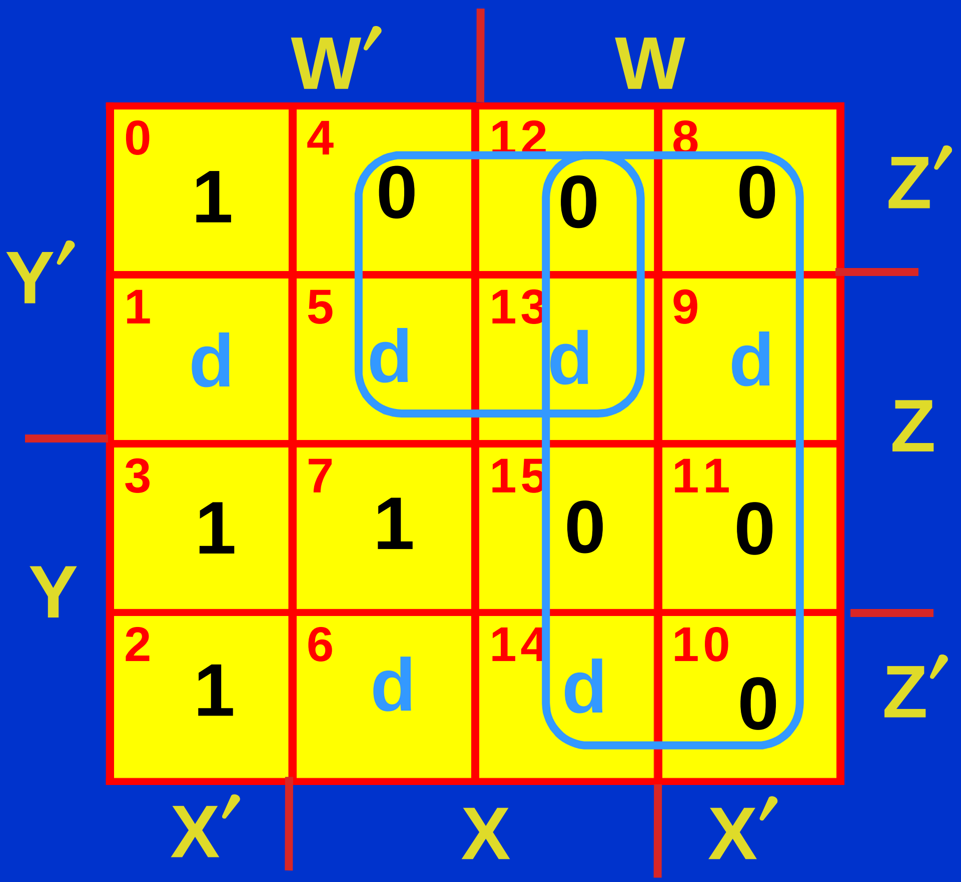
	W′		W		
Y′	0 1	4 0	12 0	8 0	Z′
	1 d	5 d	13 d	9 d	Z
Y	3 1	7 1	15 0	11 0	
	2 1	6 d	14 d	10 0	Z′
	X′	X		X′	



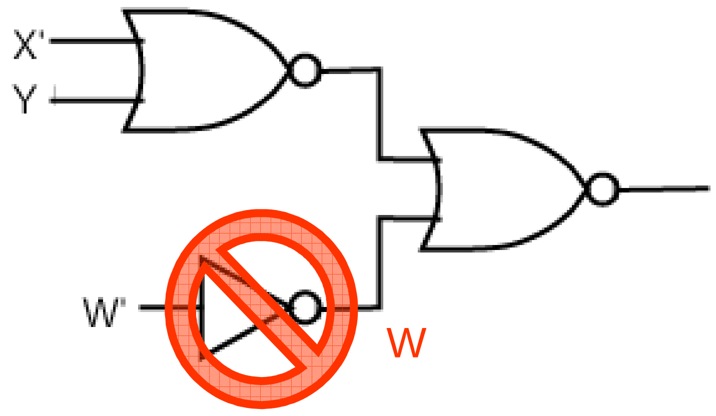
f (W,X,Y,Z) = W′•X′ + W′•Y
Cost: 6 gate inputs +
3 gate outputs
= 9 cost units

Incompletely Specified Functions

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



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



f ′(W,X,Y,Z) = W + X•Y′

f (W,X,Y,Z) = W′ • (X′+Y)

Cost: 4 gate inputs +
2 gate outputs
= 6 cost units

Clicker Quiz

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

1. The **cost** of a **minimal sum of products** realization of this function (assuming **both true and complemented variables** are available) would be:

A. 9 B. 10 C. 11 D. 12 E. none of the above

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

2. The **cost** of a **minimal products of sum** realization of this function (assuming **both true and complemented variables** are available) would be:

A. **9** **B.** **10** **C.** **11** **D.** **12** **E.** **none of the above**

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

3. Assuming the availability of **only true** input variables, the **fewest number of 2-input NAND gates** that are needed to realize this function is:

A. 6 B. 7 C. 8 D. 9 E. none of the above

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

4. Assuming the availability of **only true** input variables, the **fewest number of 2-input NOR gates** that are needed to realize this function is:

A. **6** B. **7** C. **8** D. **9** E. **none of the above**

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

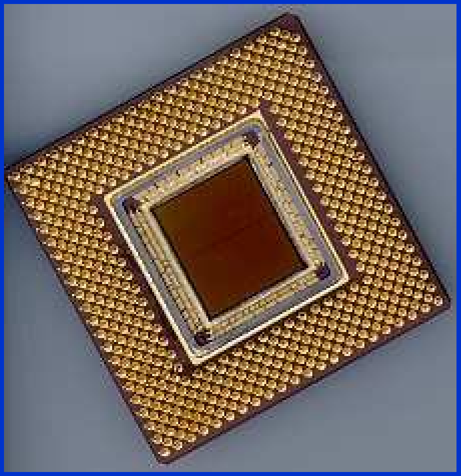
5. Assuming the availability of **only true** input variables, the **fewest number of 2-input open-drain NAND gates** that are needed to realize this function is:

A. **6** B. **7** C. **8** D. **9** E. **none of the above**

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

6. The **number of pull-up resistors** required for realizing this function **using only 2-input open drain NAND gates** (assuming the availability of **only true** input variables) is:

A. 1 B. 2 C. 3 D. 4 E. none of the above



Introduction to Digital System Design

Module 2-C
Timing Hazards

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

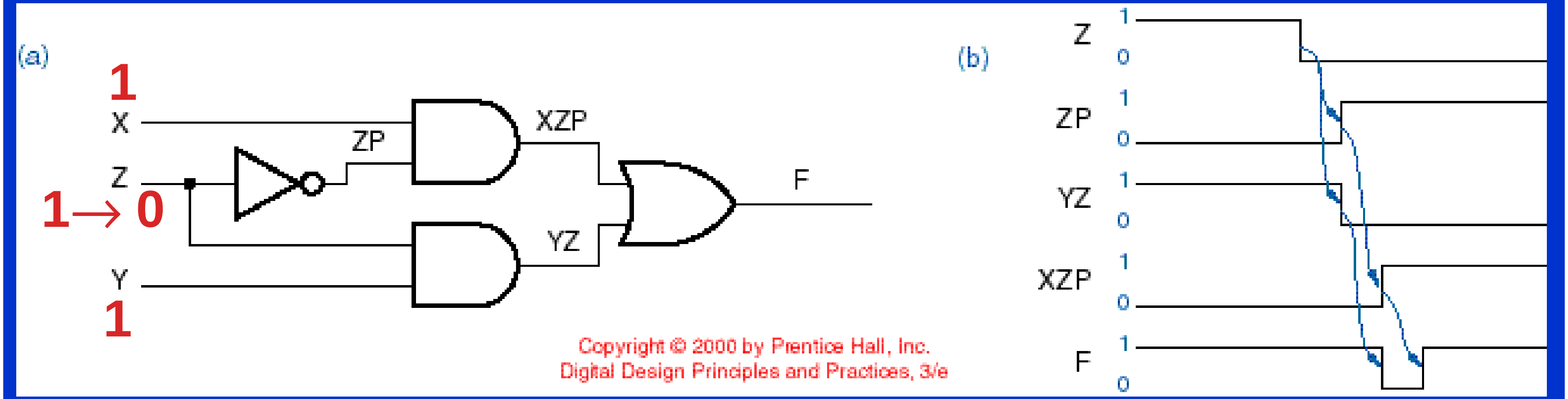
- Timing hazards
  - Static
  - Dynamic
- Elimination of timing hazards
- Clever utilization of timing hazards
- Designing hazard-free circuits

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

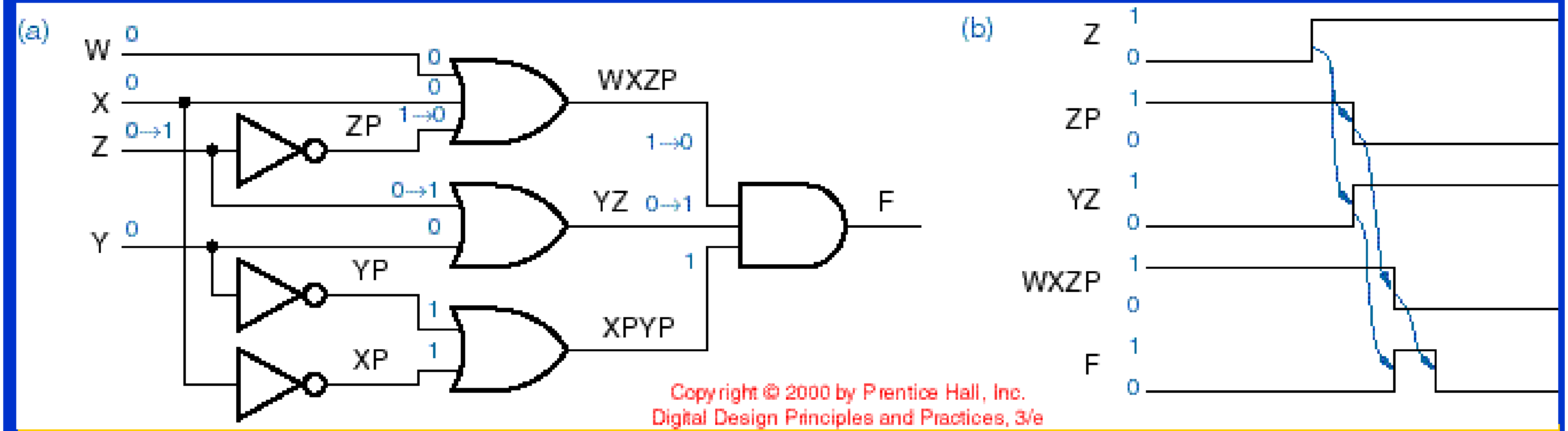
- **<u>Definition</u>: A *static-1 hazard* is a *pair of input combinations* that: (a) differ in only one input variable and (b) both produce a "1" output, such that it is possible for a momentary "0" output to occur during a transition in the differing input variable**



Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e

Timing Hazards

- **<u>Definition</u>: 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**



Copyright © 2000 by Prentice Hall, Inc.
Digital Design Principles and Practices, 3/e

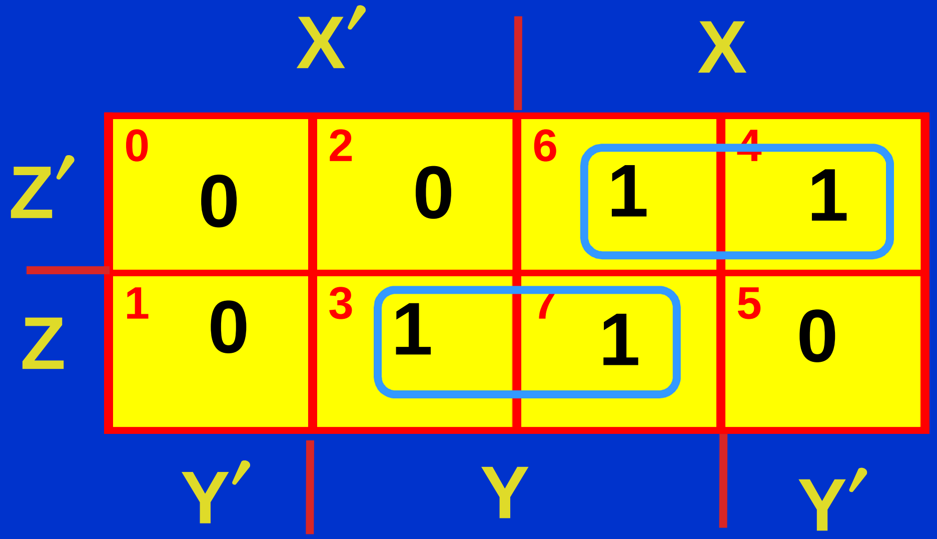
A static-0 hazard is just the *dual* of a static-1 hazard

Timing Hazards

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

Timing Hazards

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



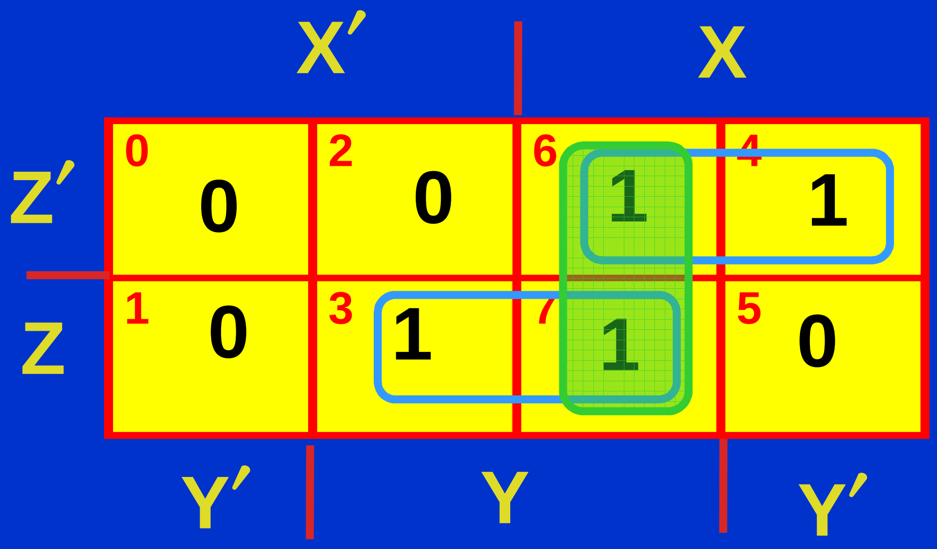
	X′		X	
Z′	0 0	2 0	6 1	4 1
Z	1 0	3 1	7 1	5 0
	Y′	Y		Y′

f (X,Y,Z) = X•Z′+Y•Z

Note: It is possible for the output to momentarily glitch to “0” if the AND gate that covers one of the combinations goes to “0” before the AND gate covering the other input combination goes to “1”

Timing Hazards

- **<u>Solution</u>: Include an extra product term (AND gate) to cover the hazardous input pair**



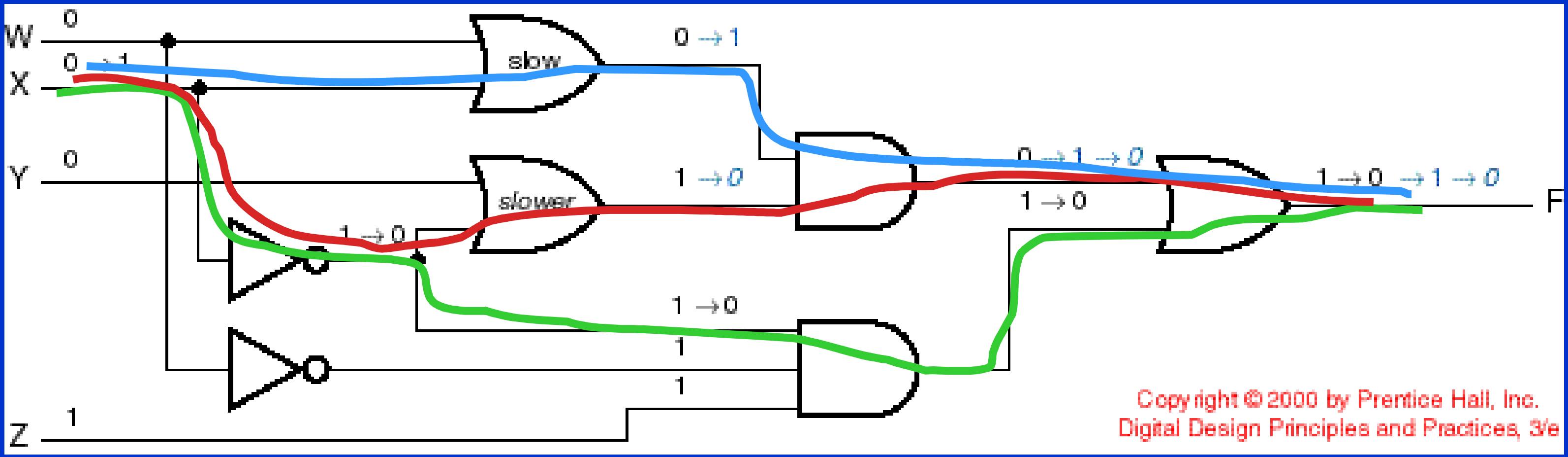
	X′		X	
Z′	0 0	2 0	6 1	4 1
Z	1 0	3 1	7 1	5 0
	Y′	Y		Y′

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

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

Timing Hazards

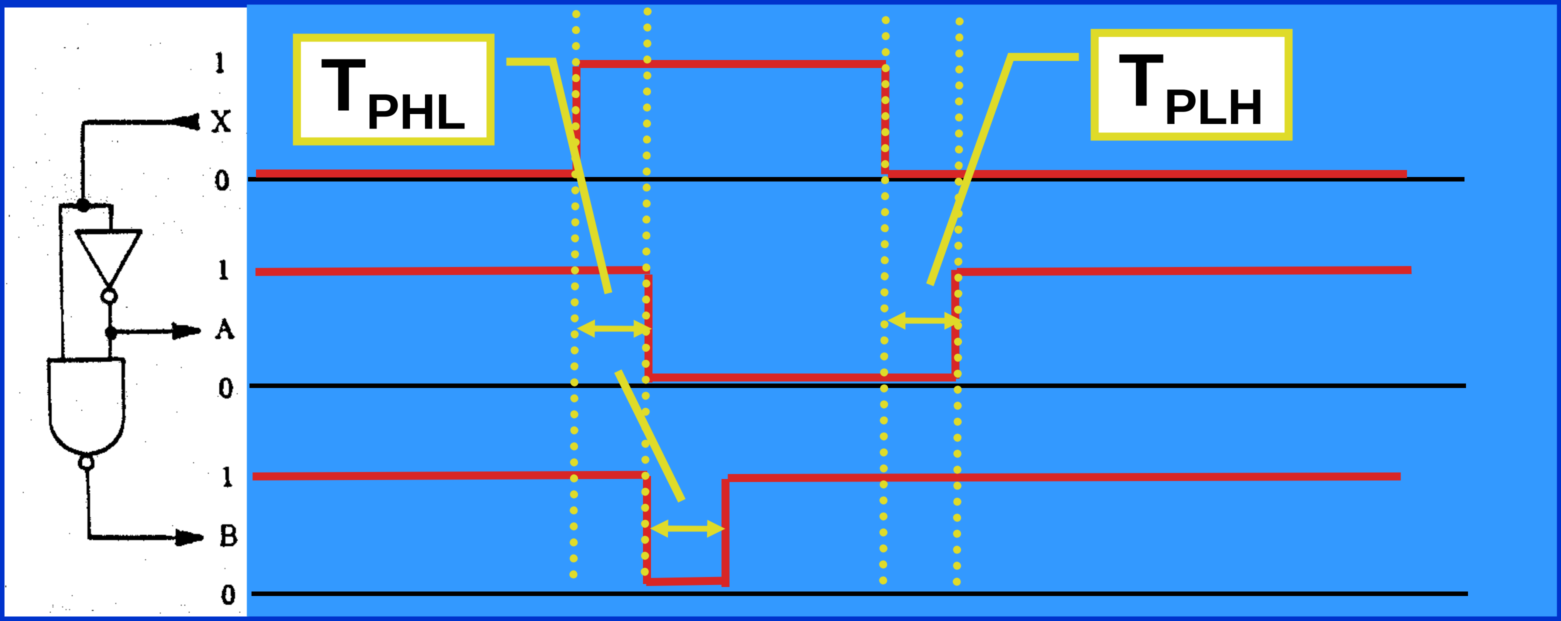
- A ***dynamic hazard*** is the possibility of an output changing ***more than once*** as the result of a single input transition
- Multiple output transitions can occur if there are ***multiple paths*** with ***different delays*** from the changing input to the changing output



Digital Design Principles and Practices, 3/e

Timing Hazards

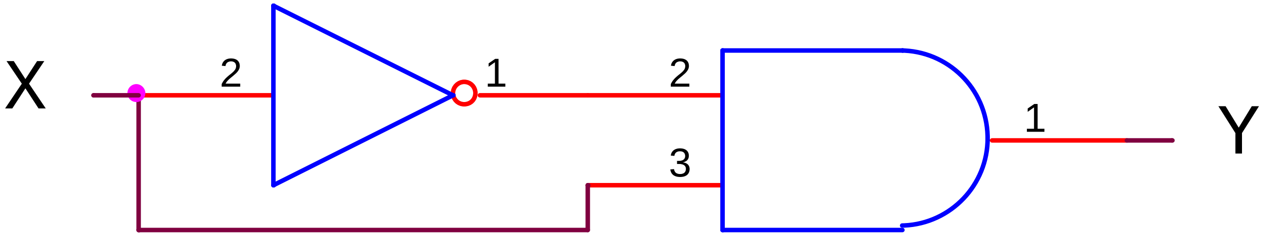
- Important: Not all hazards are hazardous – in fact, some can be quite useful! Consider the case in which we would like to detect a low-to-high transition (the “leading edge”) of a logic signal



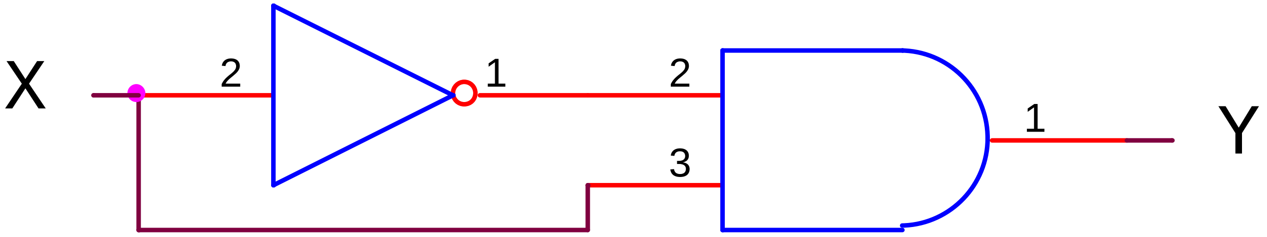
Designing Hazard-Free Circuits

- ***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., <u>all</u> 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:

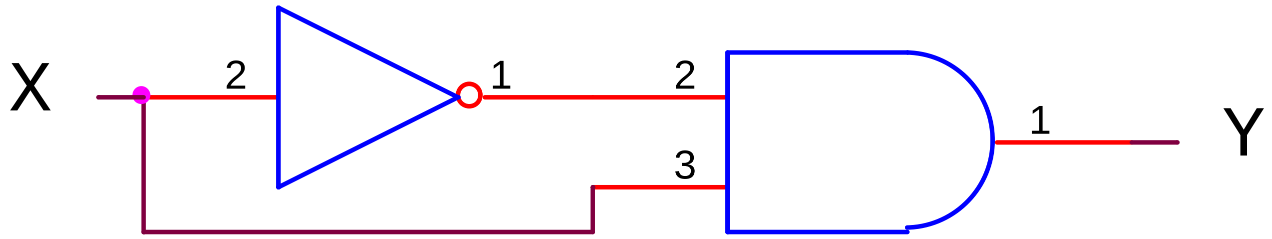
A. static-0

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:

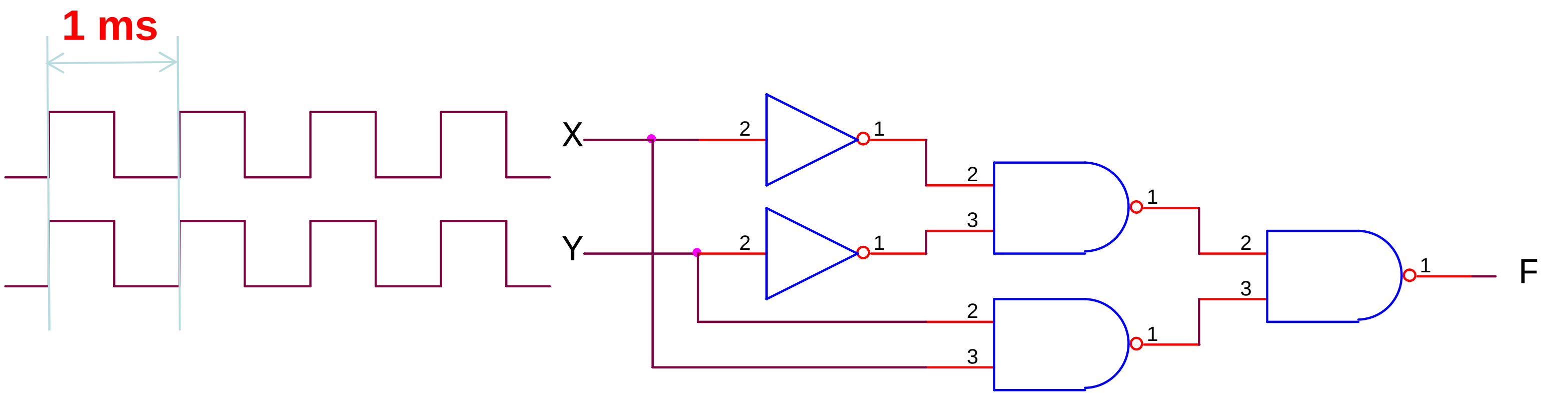
   A. static-0

   B. static-1

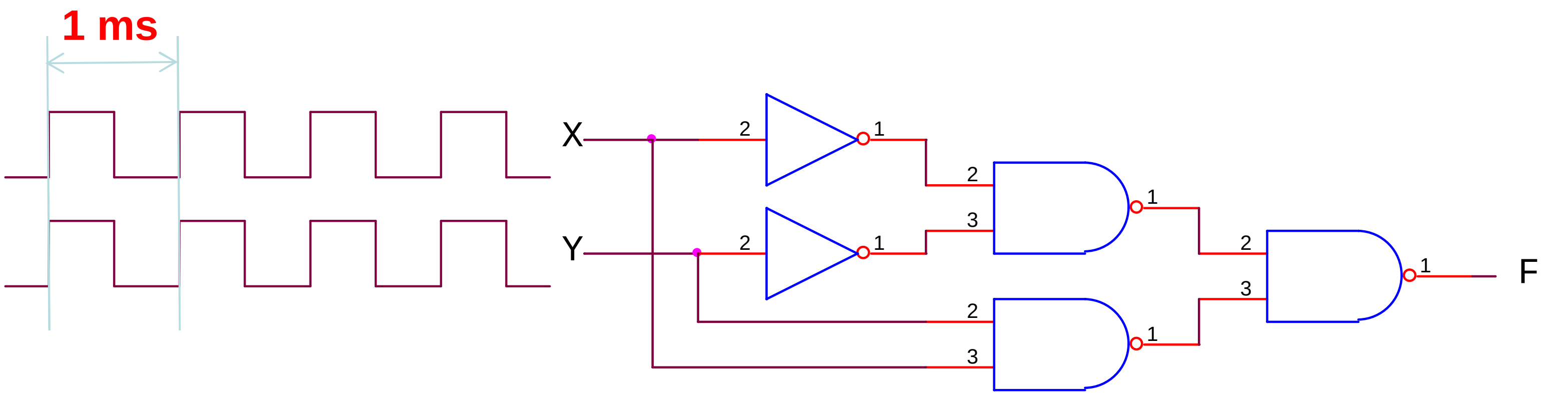
   C. dynamic

   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:**

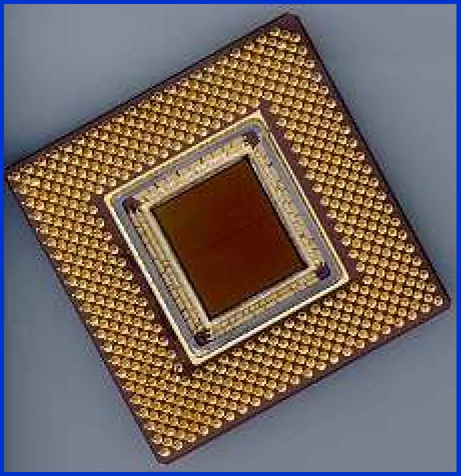
A. **a static "0" hazard will be generated in response to low-to-high transitions of the input waveform**

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**



Introduction to Digital System Design

Module 2-D
XOR/XNOR Functions

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

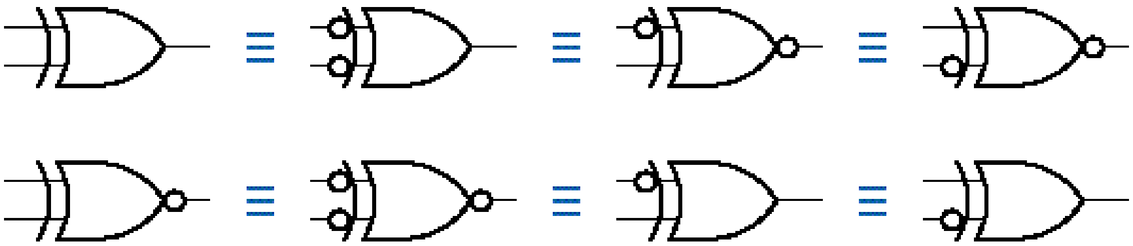
- **The *“ring sum” operator* $\oplus$ is often used to denote the XOR function: $X \oplus Y = X' \bullet Y + X \bullet Y'$**
- **The XNOR function can be thought of as either the *dual* or the *complement* of the XOR function**

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

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

XOR Operator $\oplus$ Properties

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



XOR and XNOR Equivalent Symbols

XOR K-Map

- **K-map of 2-variable XOR function**

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

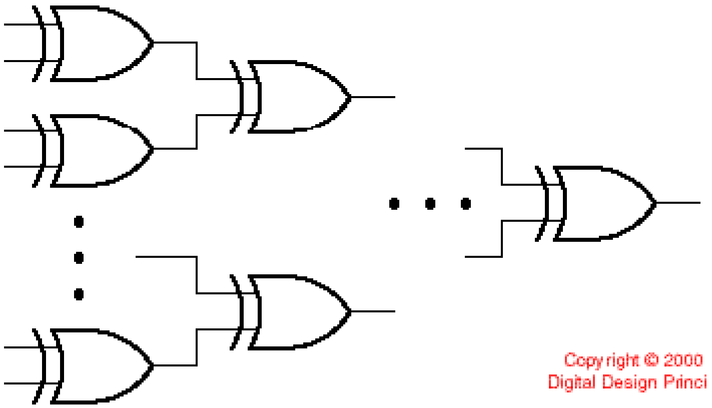
	X′	X
Y′	0 0	2 1
Y	1 1	3 0

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

XOR N-Variable Functions

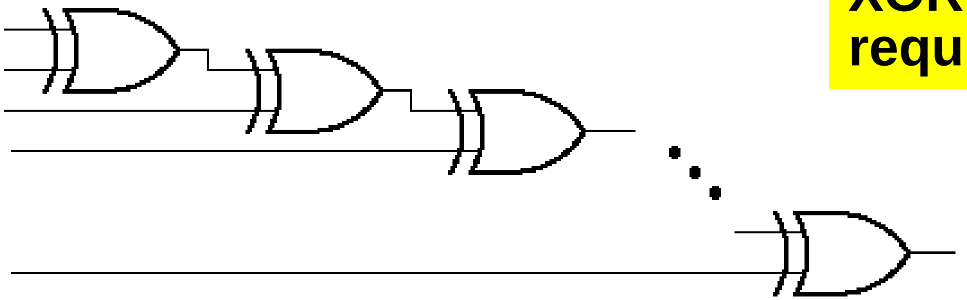
- The XOR (or XNOR) of *N* variables can be realized with *tree* or *cascade* circuits

  - tree XOR circuit (*N* is a power of 2)



Copyright © 2000
Digital Design Princi

  - cascade XOR circuit



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

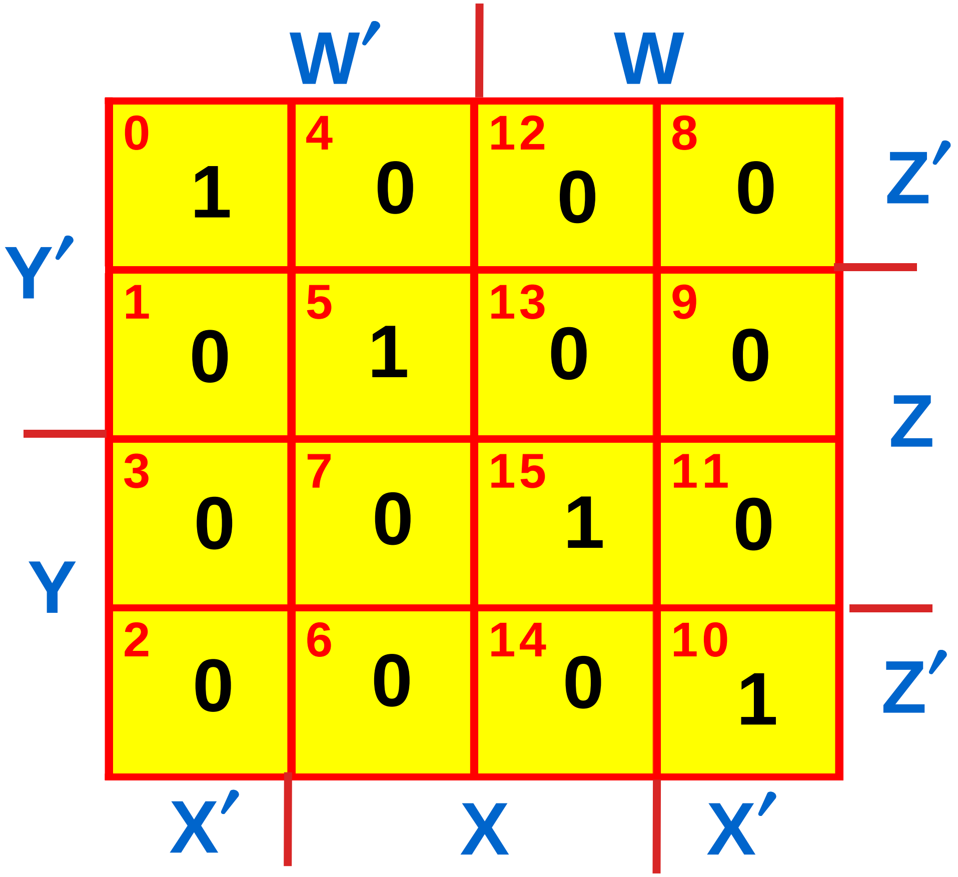
The output of an n-variable XNOR function is 1 if an even number of inputs are 1

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

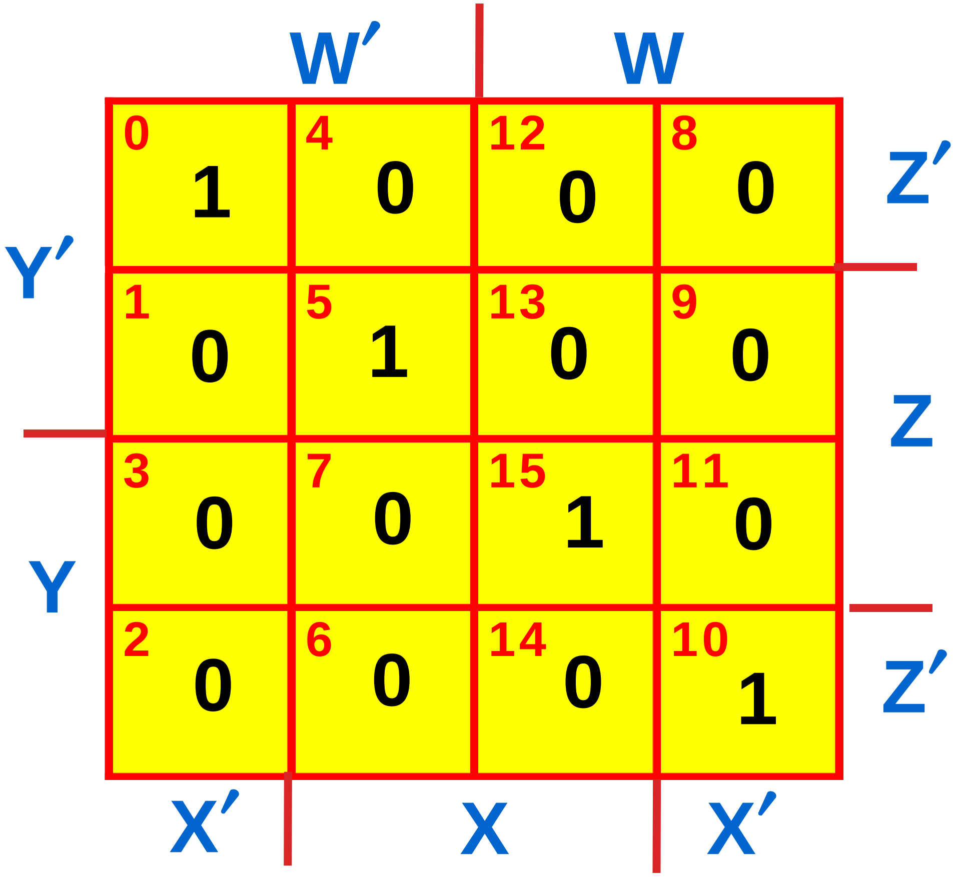
Example – “Diagonal” K-map



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

Example – “Diagonal” K-map

- **Minimize function to the extent possible**

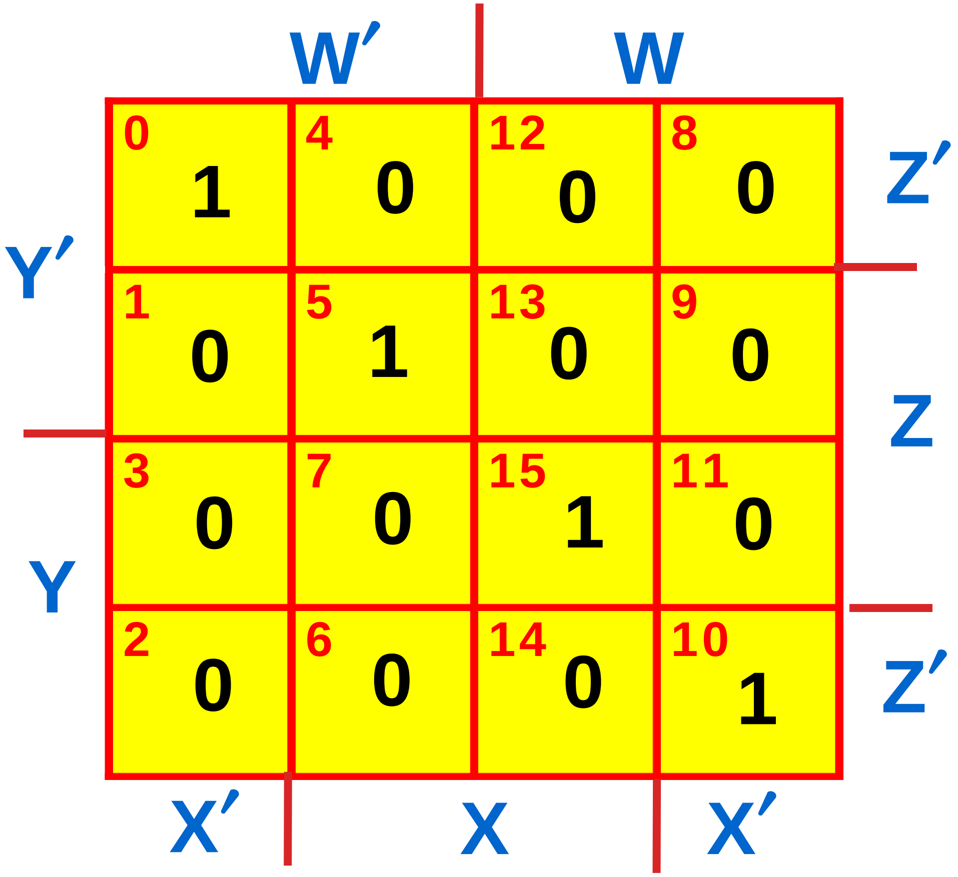


	W′		W		
Y′	0 1	4 0	12 0	8 0	Z′
	1 0	5 1	13 0	9 0	Z
Y	3 0	7 0	15 1	11 0	
	2 0	6 0	14 0	10 1	Z′
	X′	X		X′	

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

Example – “Diagonal” K-map

- **Factor out XOR/XNOR expressions**



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

$F(W,X,Y,Z) =$

$W'\bullet X'\bullet Y'\bullet Z' + W'\bullet X\bullet Y'\bullet Z$

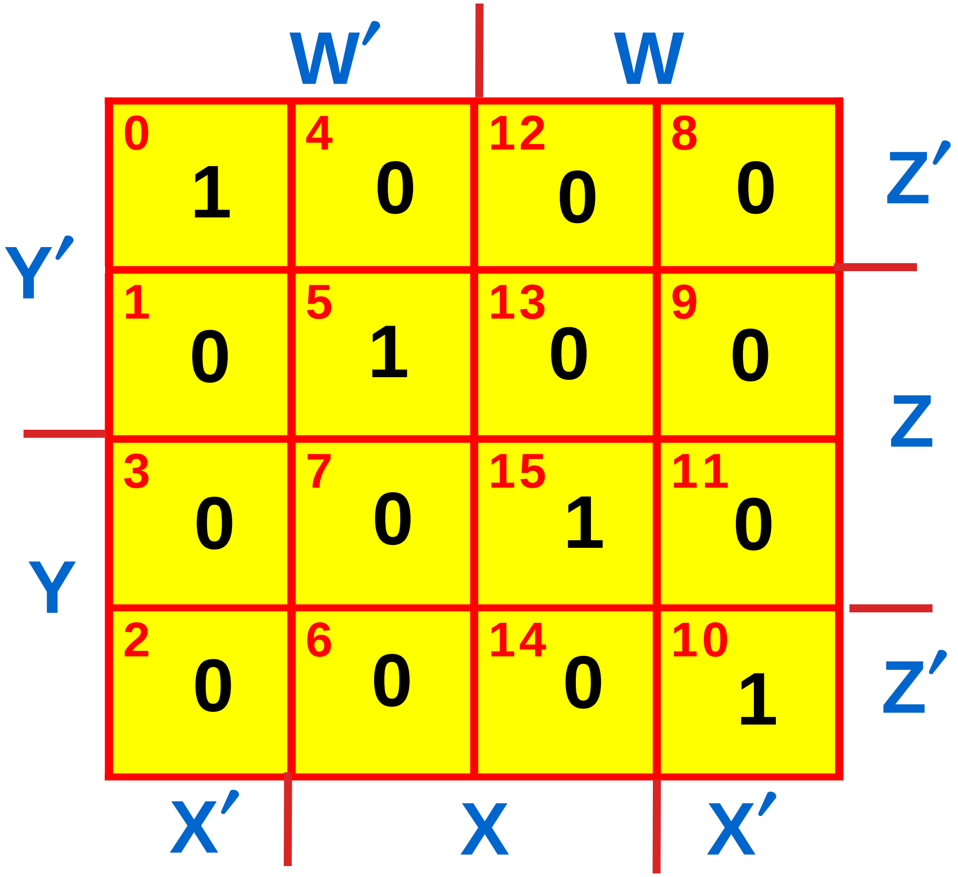
$+ W\bullet X\bullet Y\bullet Z + W\bullet X'\bullet Y\bullet Z'$

$= X'\bullet Z' \bullet (W'\bullet Y' + W\bullet Y)$

$+ X\bullet Z \bullet (W'\bullet Y' + W\bullet Y)$

Example – “Diagonal” K-map

- **Factor out XOR/XNOR expressions**



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

$F(W,X,Y,Z) =$

$W'\bullet X'\bullet Y'\bullet Z' + W'\bullet X\bullet Y'\bullet Z$

$+ W\bullet X\bullet Y\bullet Z + W\bullet X'\bullet Y\bullet Z'$

$= X'\bullet Z' \bullet (W'\bullet Y' + W\bullet Y)$

$+ X\bullet Z \bullet (W'\bullet Y' + W\bullet Y)$

$= (X'\bullet Z' + X\bullet Z)\bullet(W'\bullet Y' + W\bullet Y)$

Example – “Diagonal” K-map

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

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

$F(W,X,Y,Z) =$

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

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

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

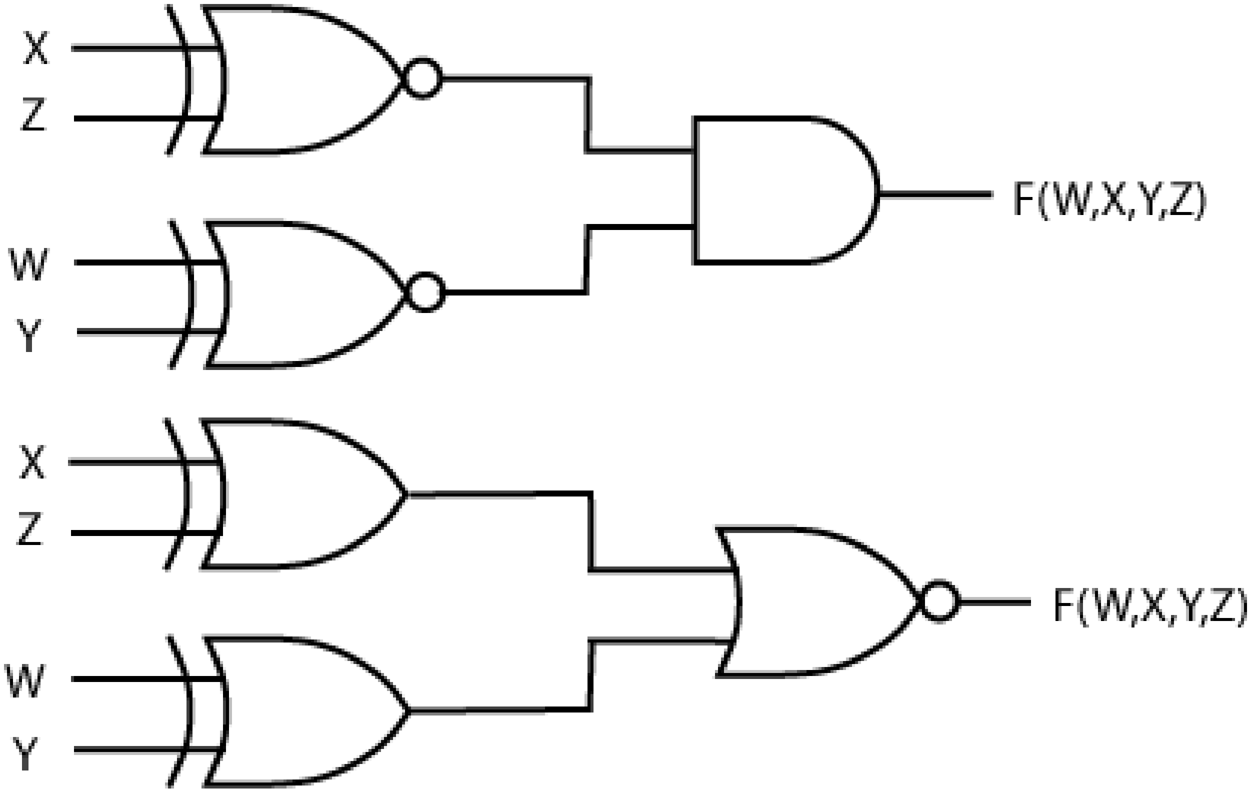
$+ X\cdot Z \cdot (W'\cdot Y' + W\cdot Y)$

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

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

Example – “Diagonal” K-map

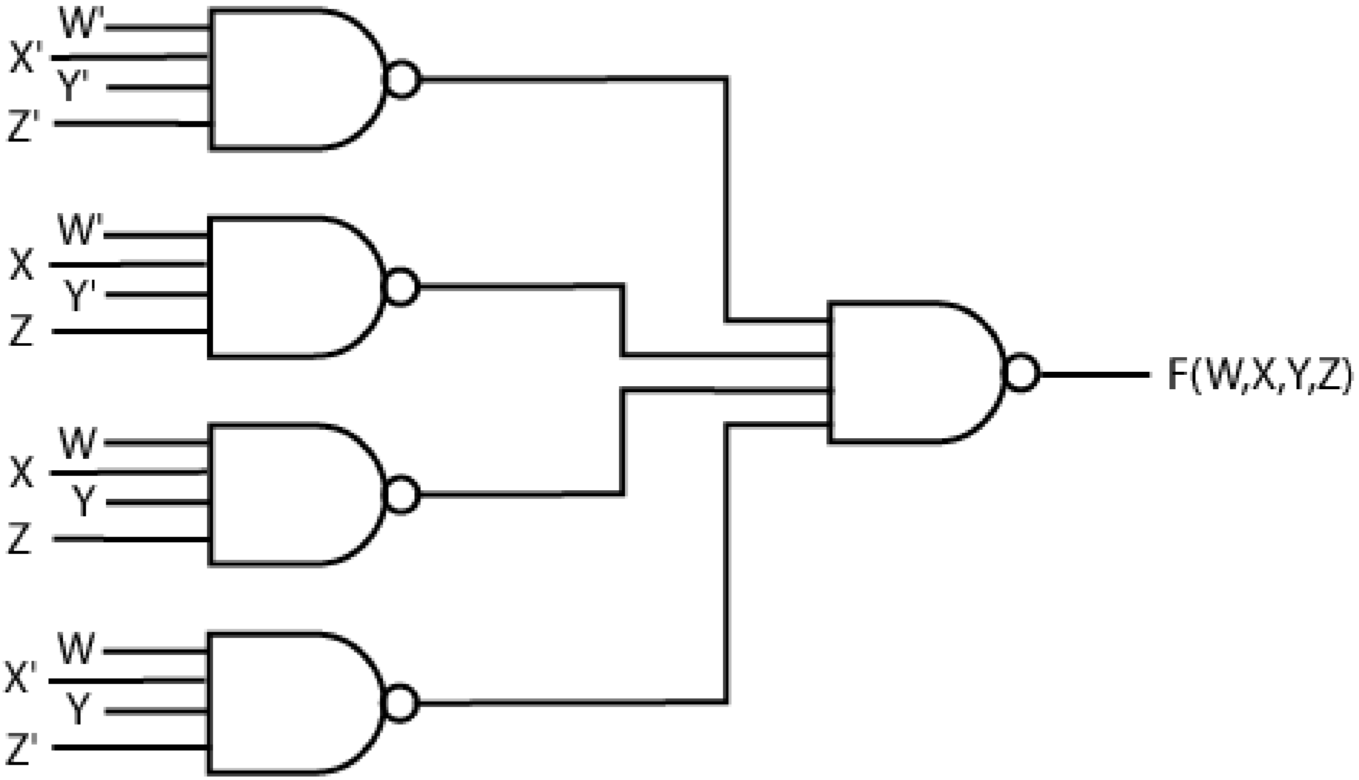
- **Realize using XOR/XNOR gates**



COST = 6 inputs + 3 outputs = 9

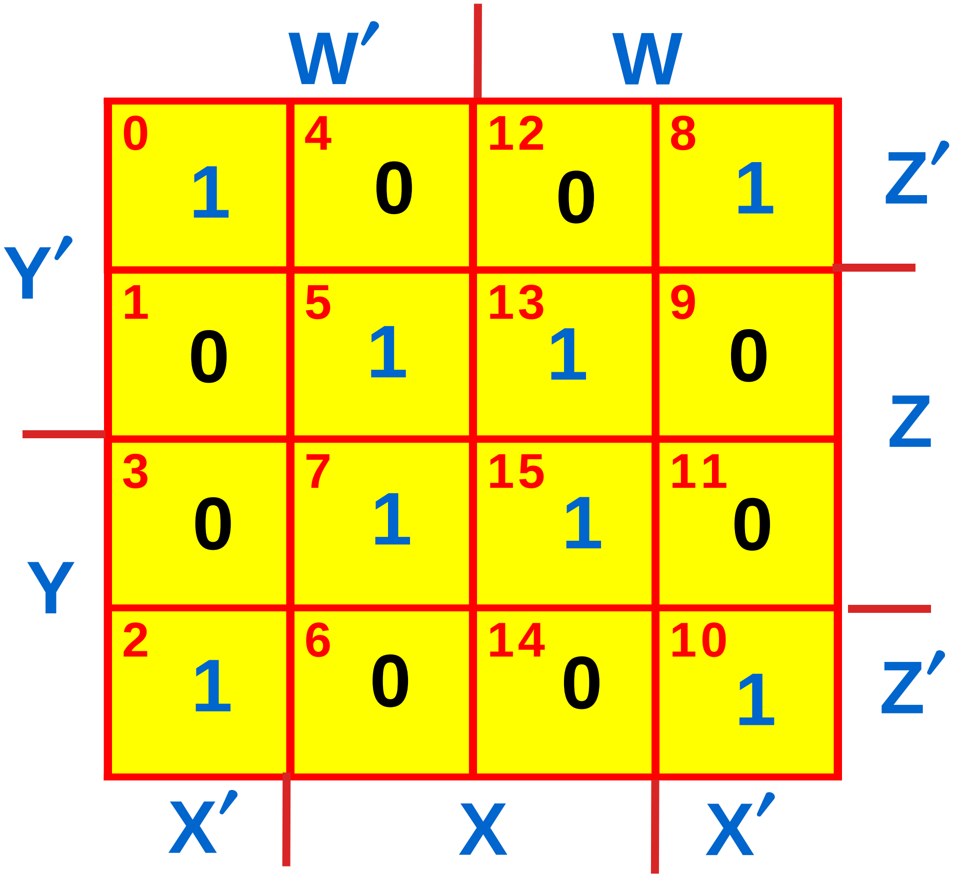
Example – “Diagonal” K-map

- **Compare with minimal SoP realization**



COST = 20 inputs + 5 outputs = 25

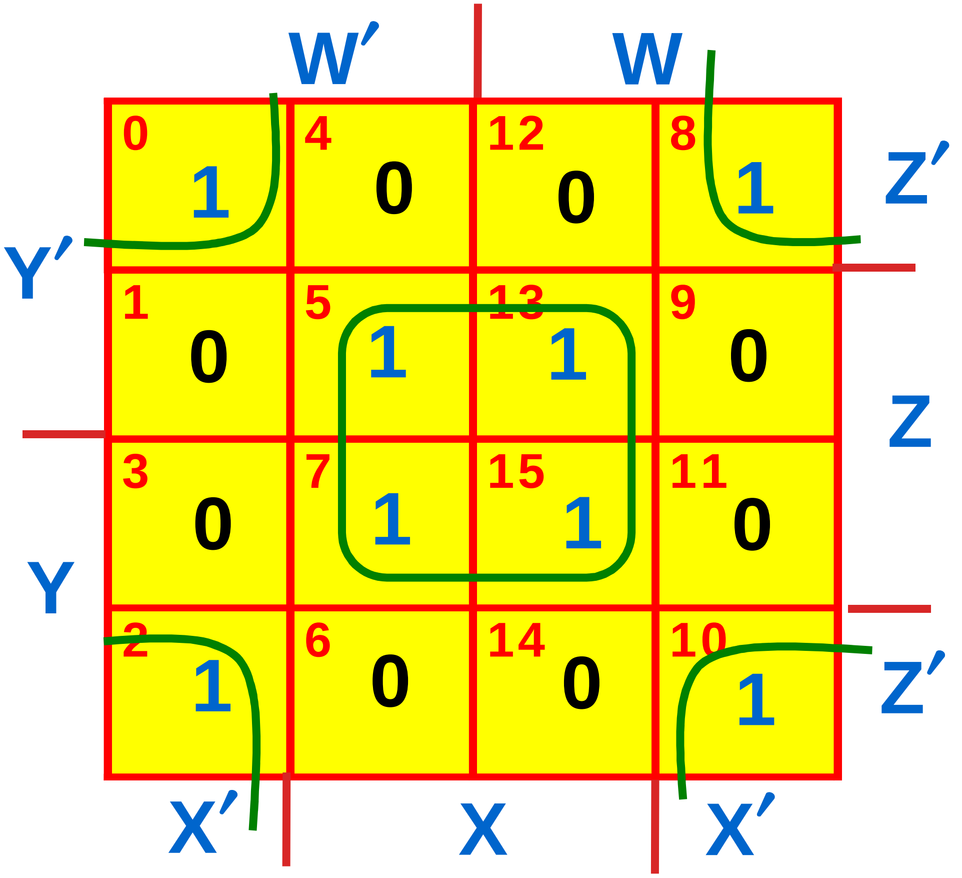
Example – “X”-map



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

Example – “X”-map

- Minimize function to the extent possible

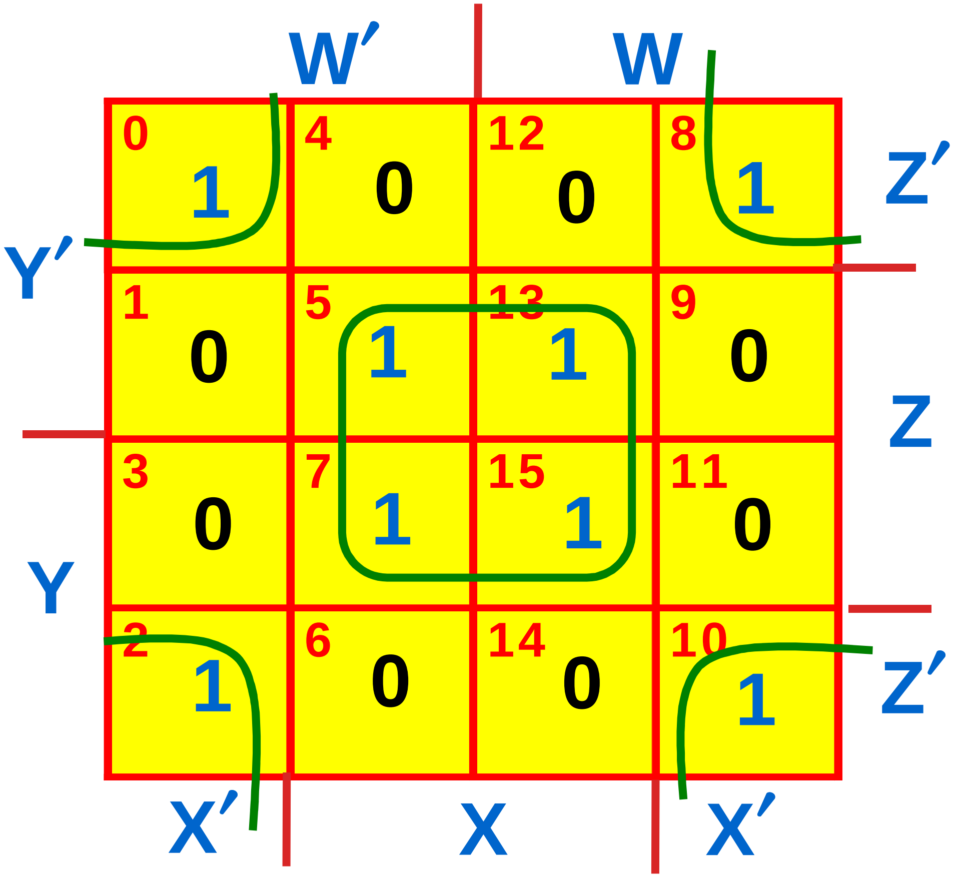


0 1	4 0	12 0	8 1
1 0	5 1	13 1	9 0
3 0	7 1	15 1	11 0
2 1	6 0	14 0	10 1

F(W,X,Y,Z) =
X′•Z′ + X•Z

Example – “X”-map

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



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

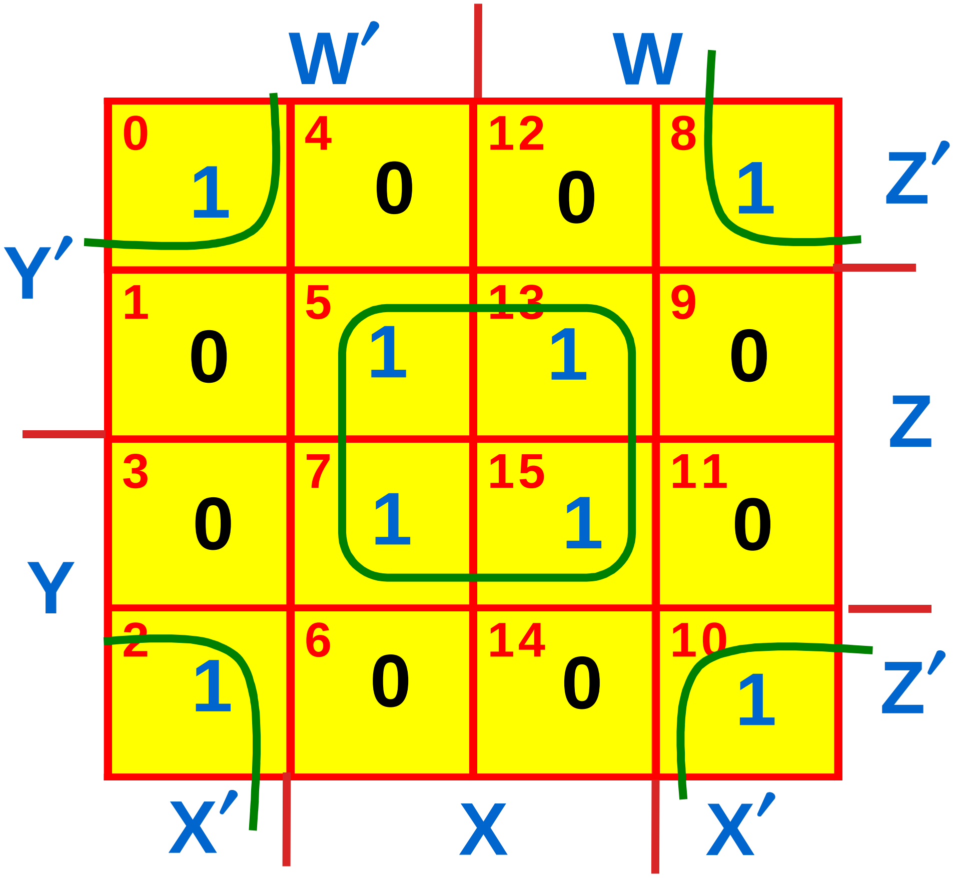
F(W,X,Y,Z) =

X′•Z′ + X•Z

= (X ⊕ Z)′

Example – “X”-map

- **Compare costs**



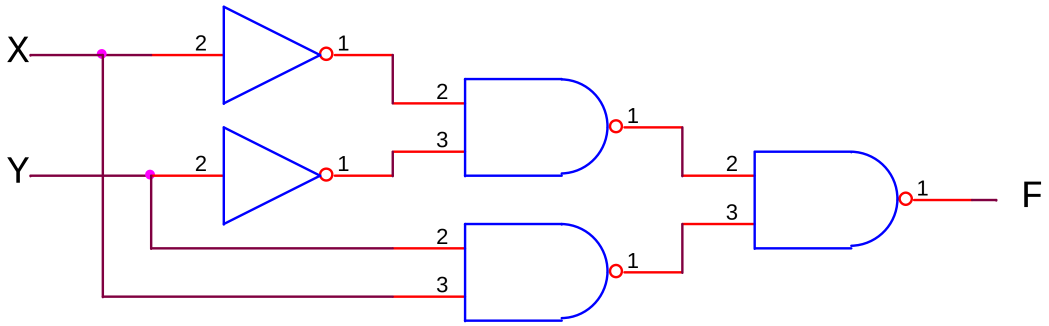
0 1	4 0	12 0	8 1
1 0	5 1	13 1	9 0
3 0	7 1	15 1	11 0
2 1	6 0	14 0	10 1

F(W,X,Y,Z) =

X′•Z′ + X•Z   Cost=9

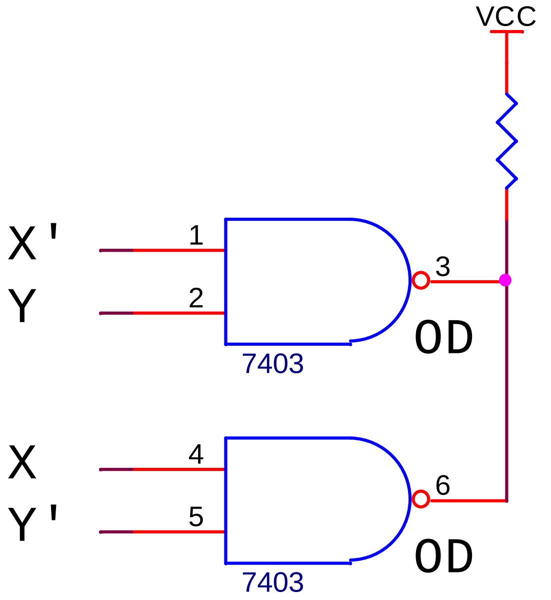
= (X ⊕ Z)′   Cost=3

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:

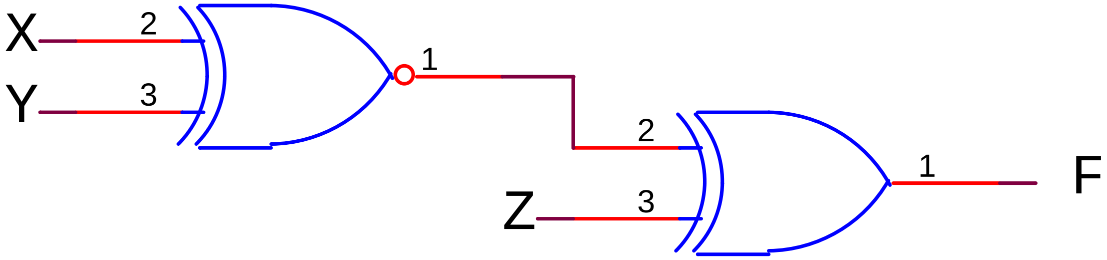
A. $\sum_{X,Y}(0,2)$

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. $\sum_{X,Y,Z}$**(0,3,4,7)**

B. $\sum_{X,Y,Z}$**(1,2,5,6)**

C. $\sum_{X,Y,Z}$**(0,3,5,6)**

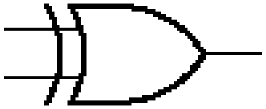
D. $\sum_{X,Y,Z}$**(1,2,4,7)**

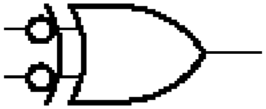
E. none of the above

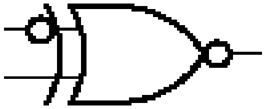
4. The XOR property listed below that is NOT true is:

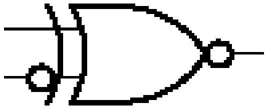
A. $X \oplus 0 = X$
B. $X \oplus 1 = X'$
C. $X \oplus X = X$
D. $X \oplus X' = 1$
E. none of the above

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

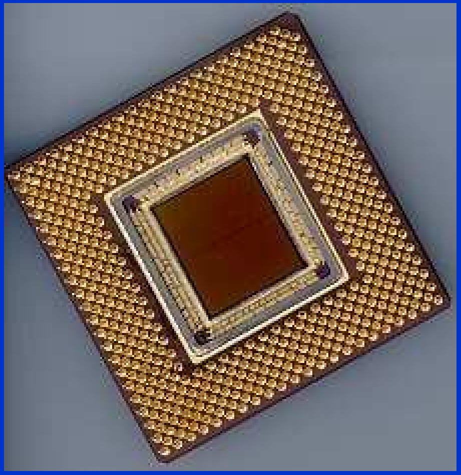
A. 

B. 

C. 

D. 

E. none of the above



Introduction to Digital System Design

Module 2-E
Programmable Logic Devices

Reading Assignment:

DDPP 4th Ed., pp. 370-383, 840-859

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*)**

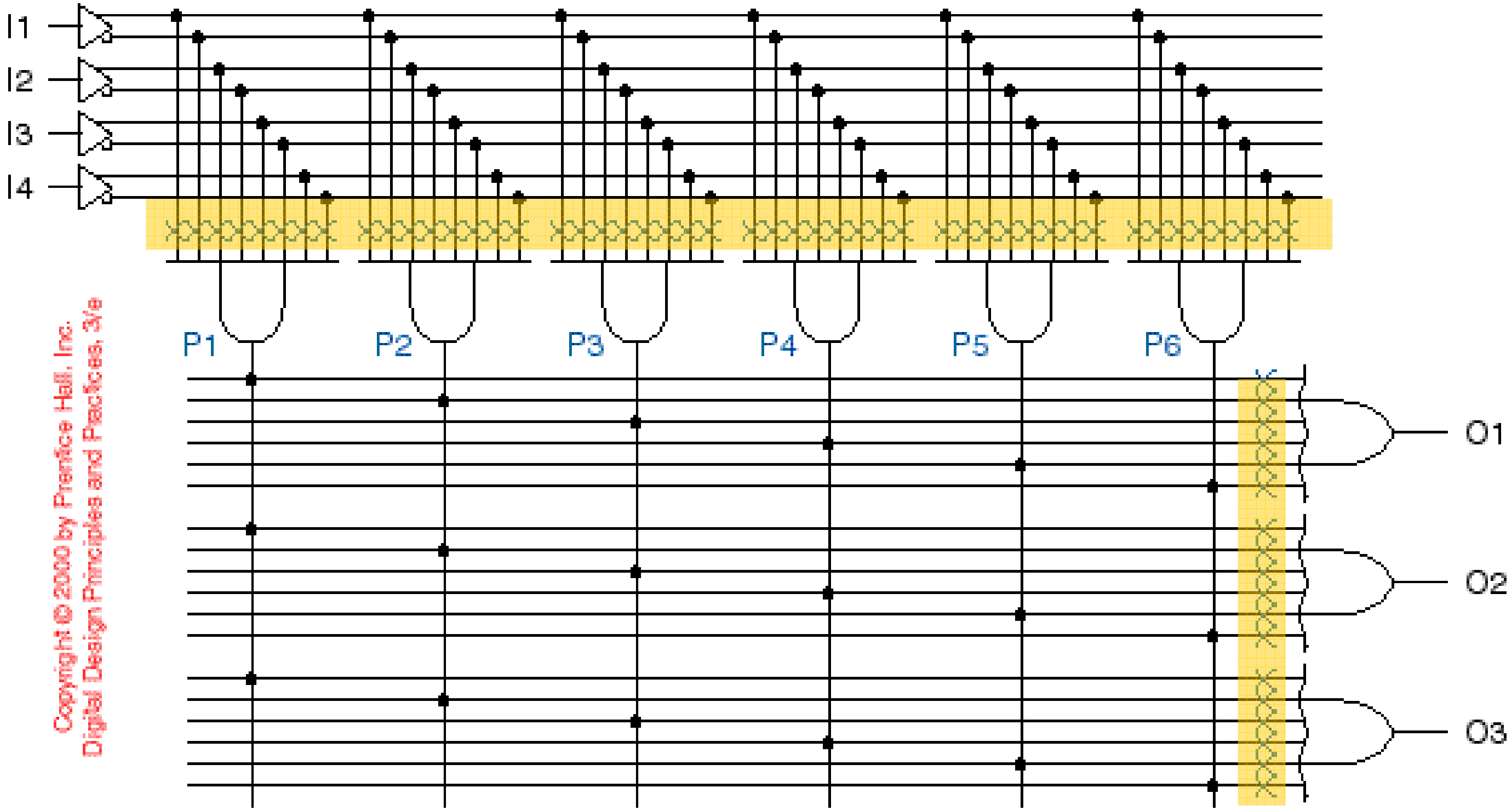
Such a device might be described as an *n* x *m* PLA with *p* product terms

Overview

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

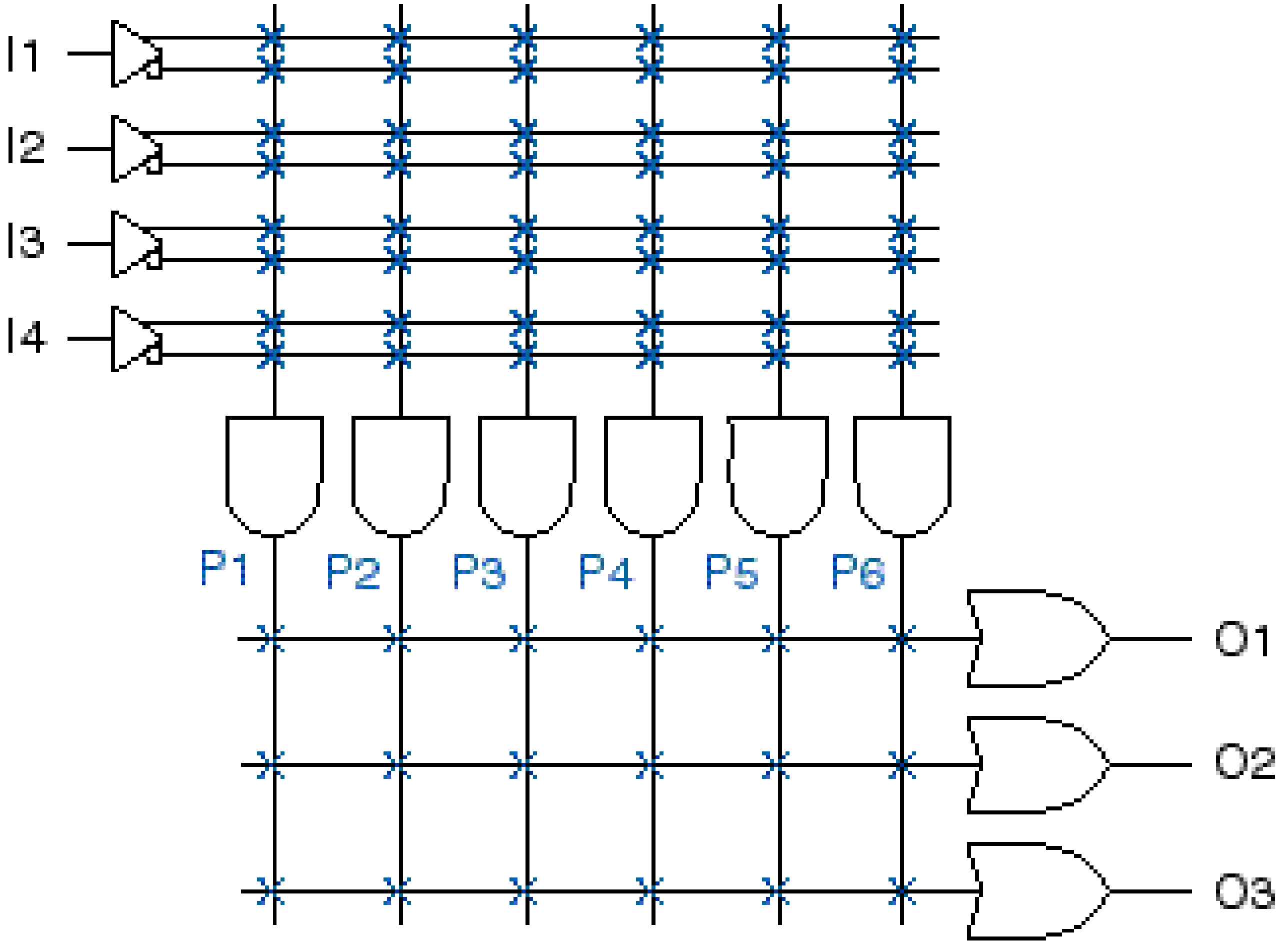
- **4 x 3 PLA with 6 product terms**



Potential connections indicated by "X"

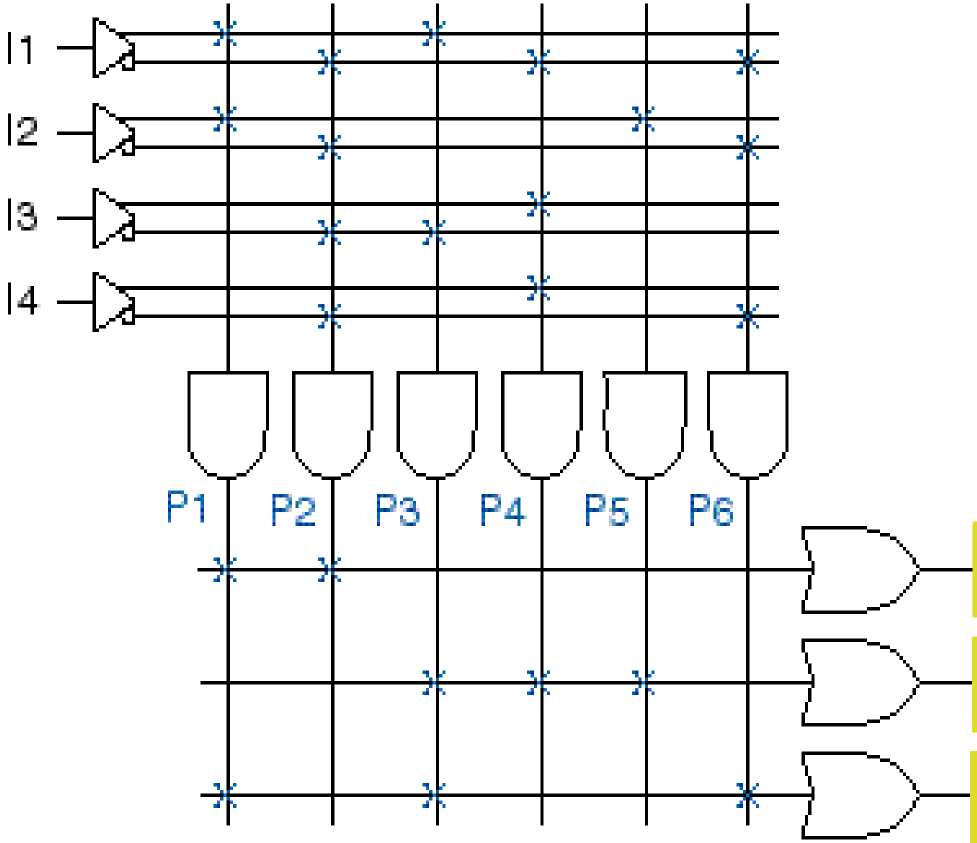
Programmable Logic Array

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



Programmable Logic Array

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



I1•I2 + I1′•I2′•I3′•I4′

I1•I3′ + I1′•I3•I4 + I2

I1•I2 + I1•I3′ + I1′•I2′•I4′

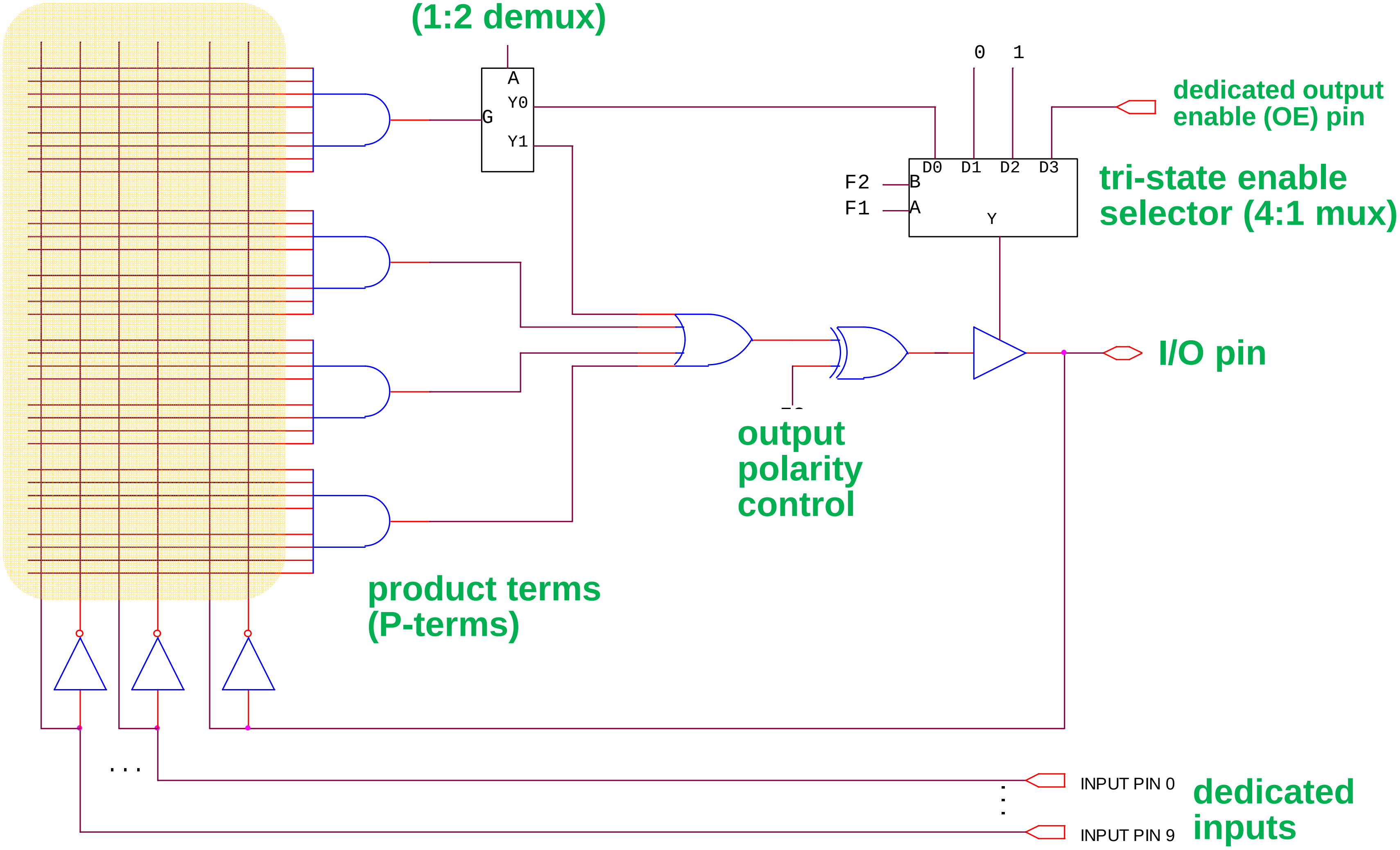
Programmable Array Logic

- 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 can not be shared)
- Each output has an individual tri-state enable, controlled by a dedicated AND gate
- There is an inverter between the output of the OR gate and the external pin
- Some of the output pins may also be used as inputs (called **“I/O pins”**)
  - tri-state buffer OFF, *input only*
  - tri-state buffer ON, either *output-only*, output *cascaded* to another function input, or *feedback* to create a sequential circuit

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)

dedicated output enable (OE) pin

tri-state enable selector (4:1 mux)

I/O pin

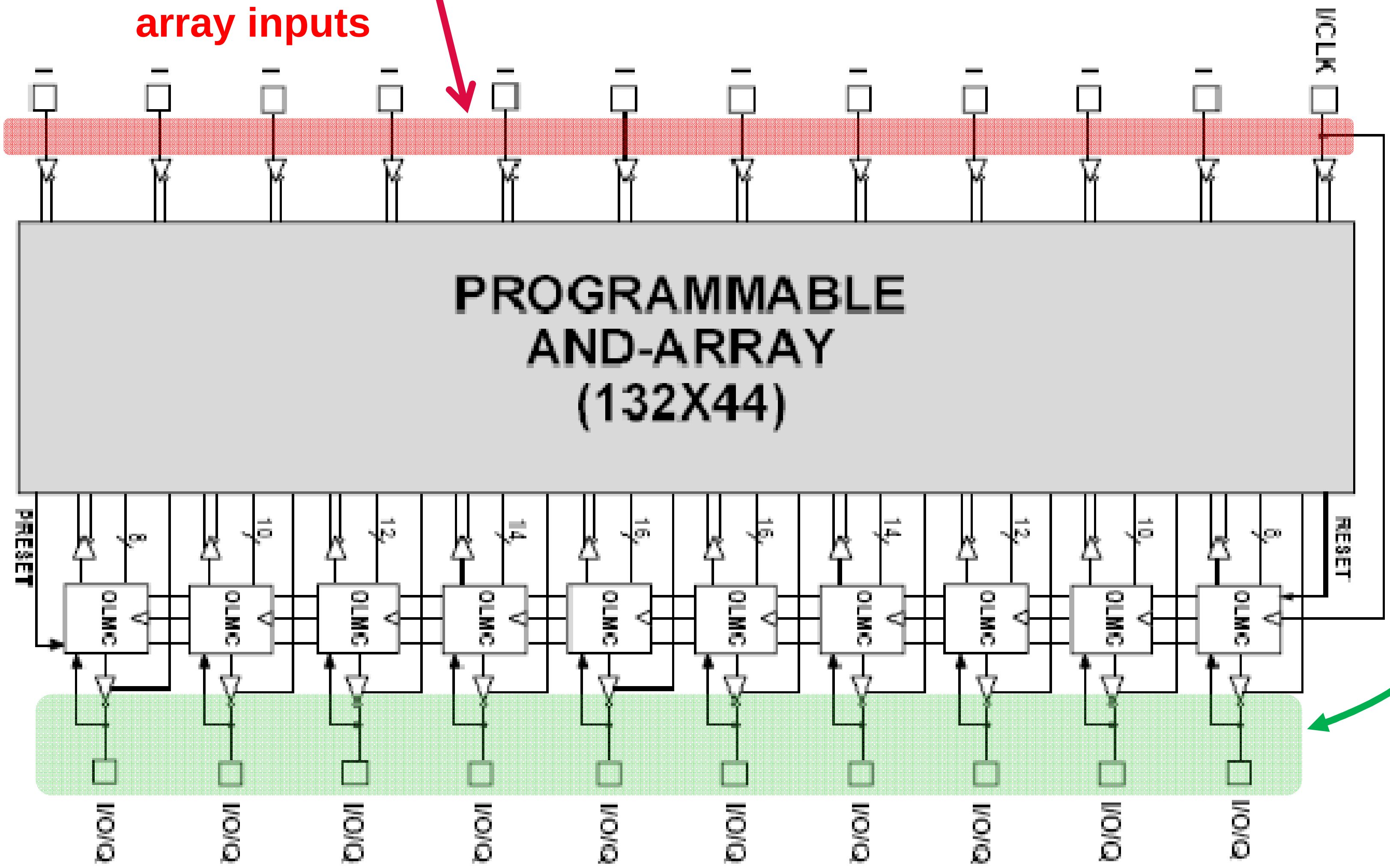
output polarity control

product terms (P-terms)

dedicated inputs

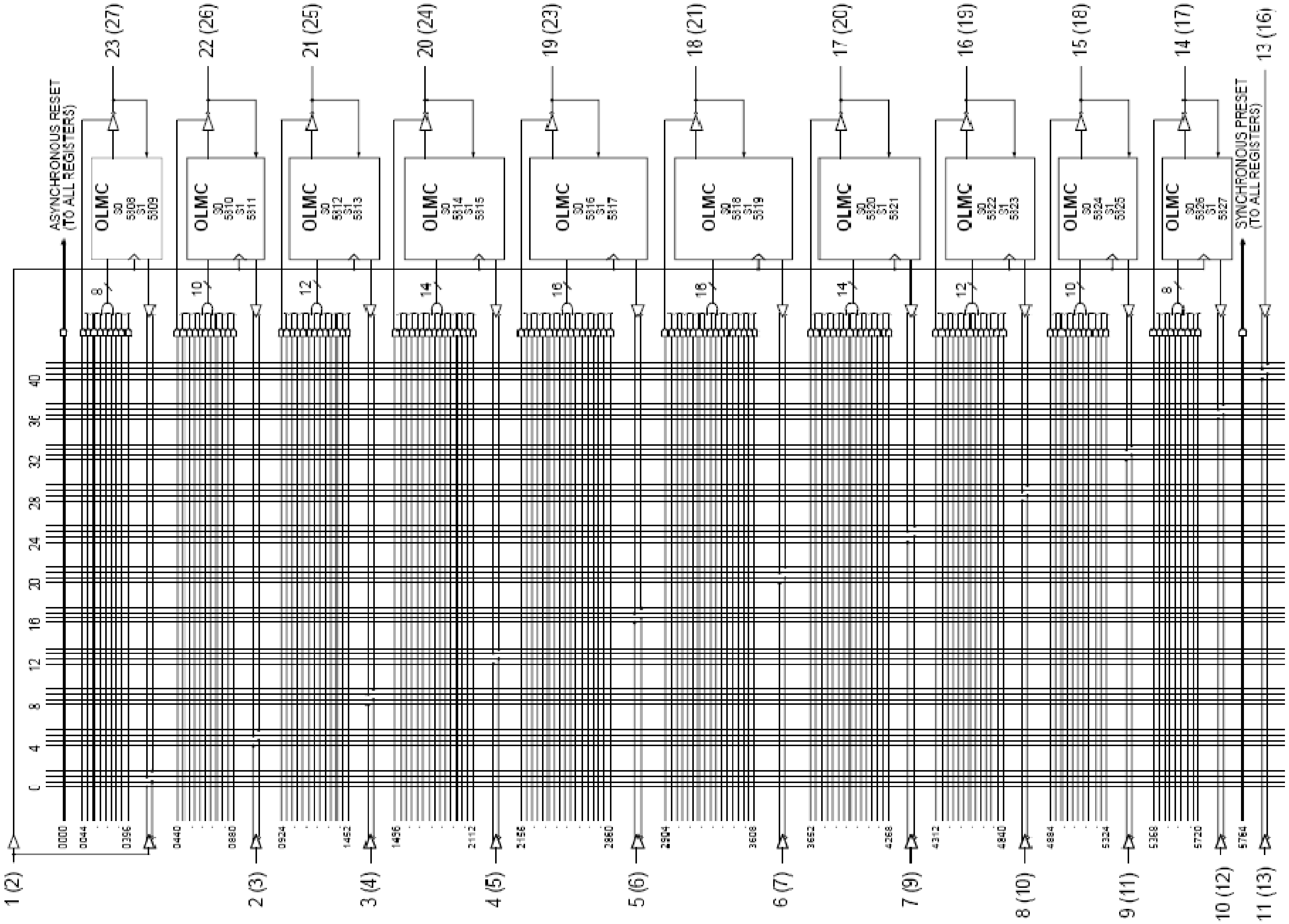
GAL22V10 Block Diagram

number of AND array inputs

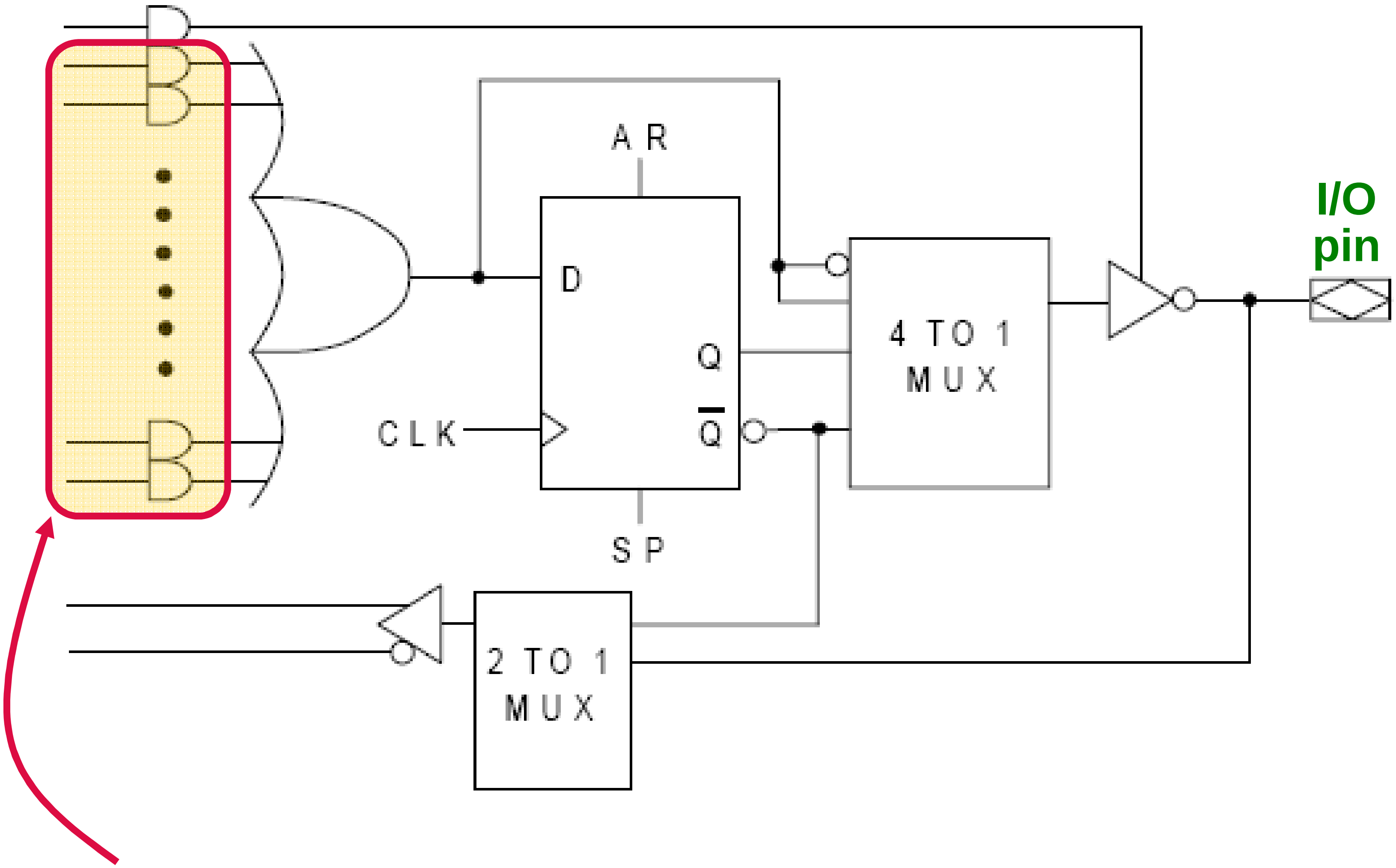


number of macrocells and associated I/O pins

GAL22V10 AND Array (“Fuse Matrix”)

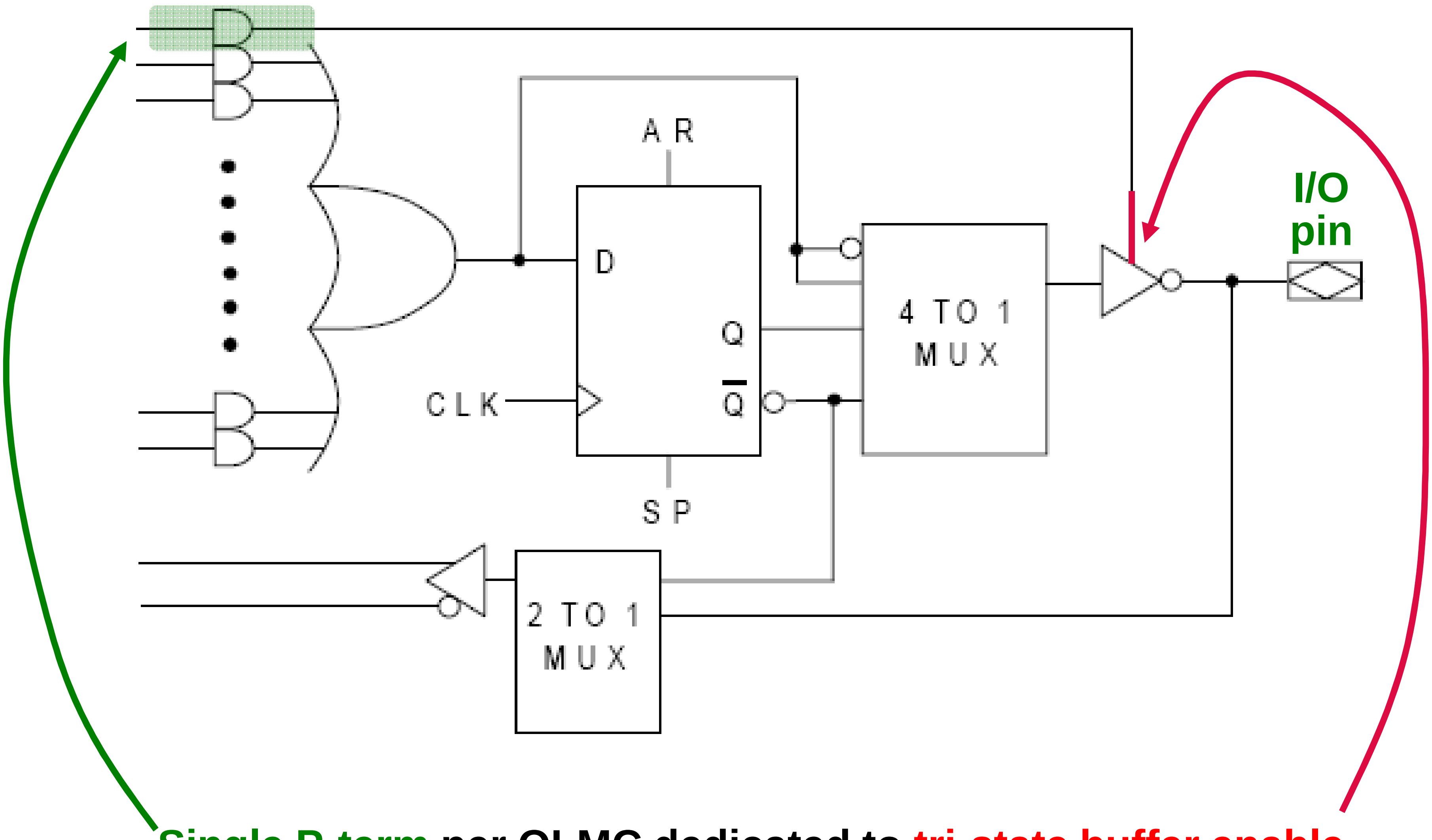


GAL22V10 Output Logic Macrocell ("OLMC")



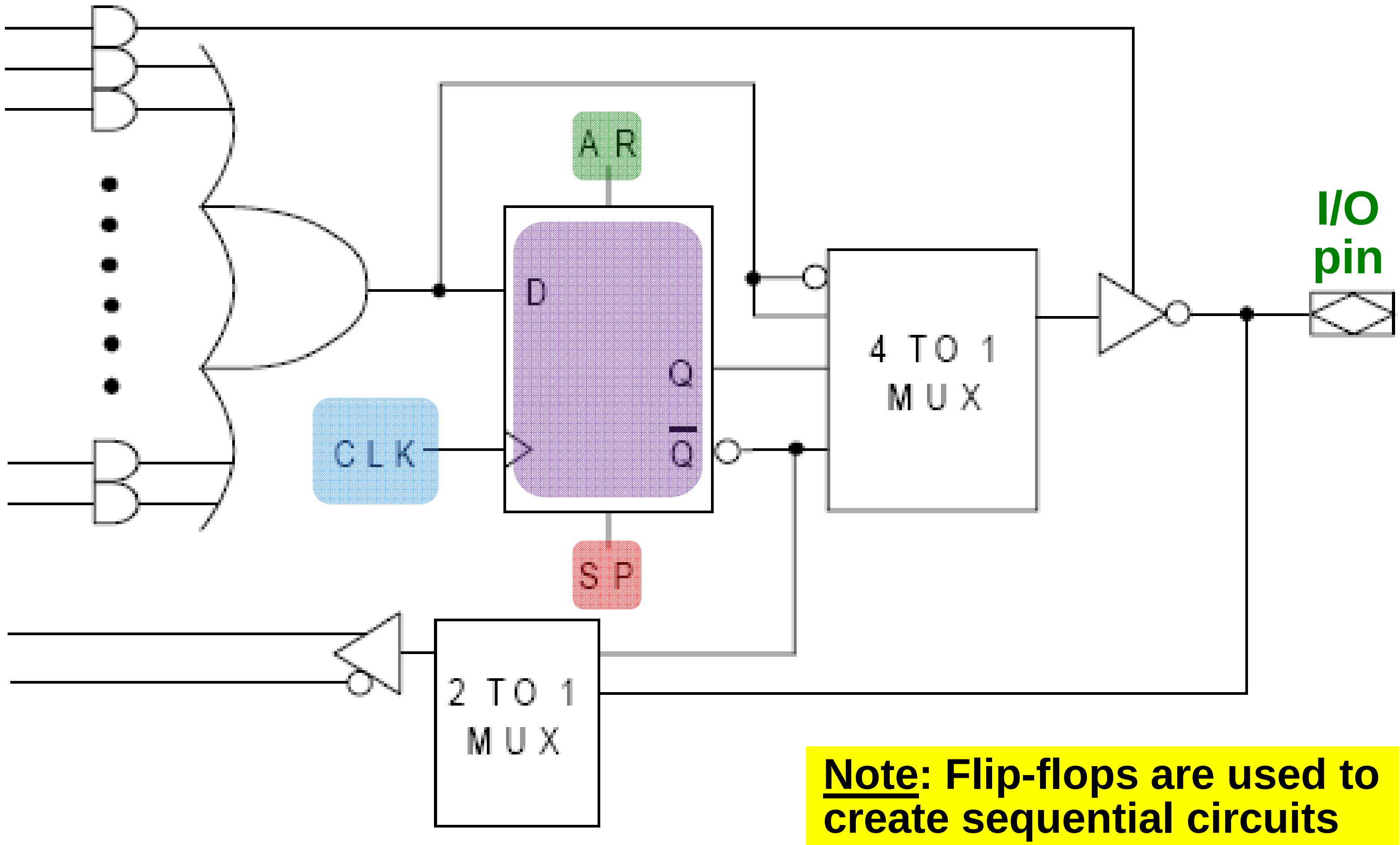
Number of P-terms per OLMC ranges from 8 to 16

GAL22V10 Output Logic Macrocell (“OLMC”)



Single P-term per OLMC dedicated to tri-state buffer enable

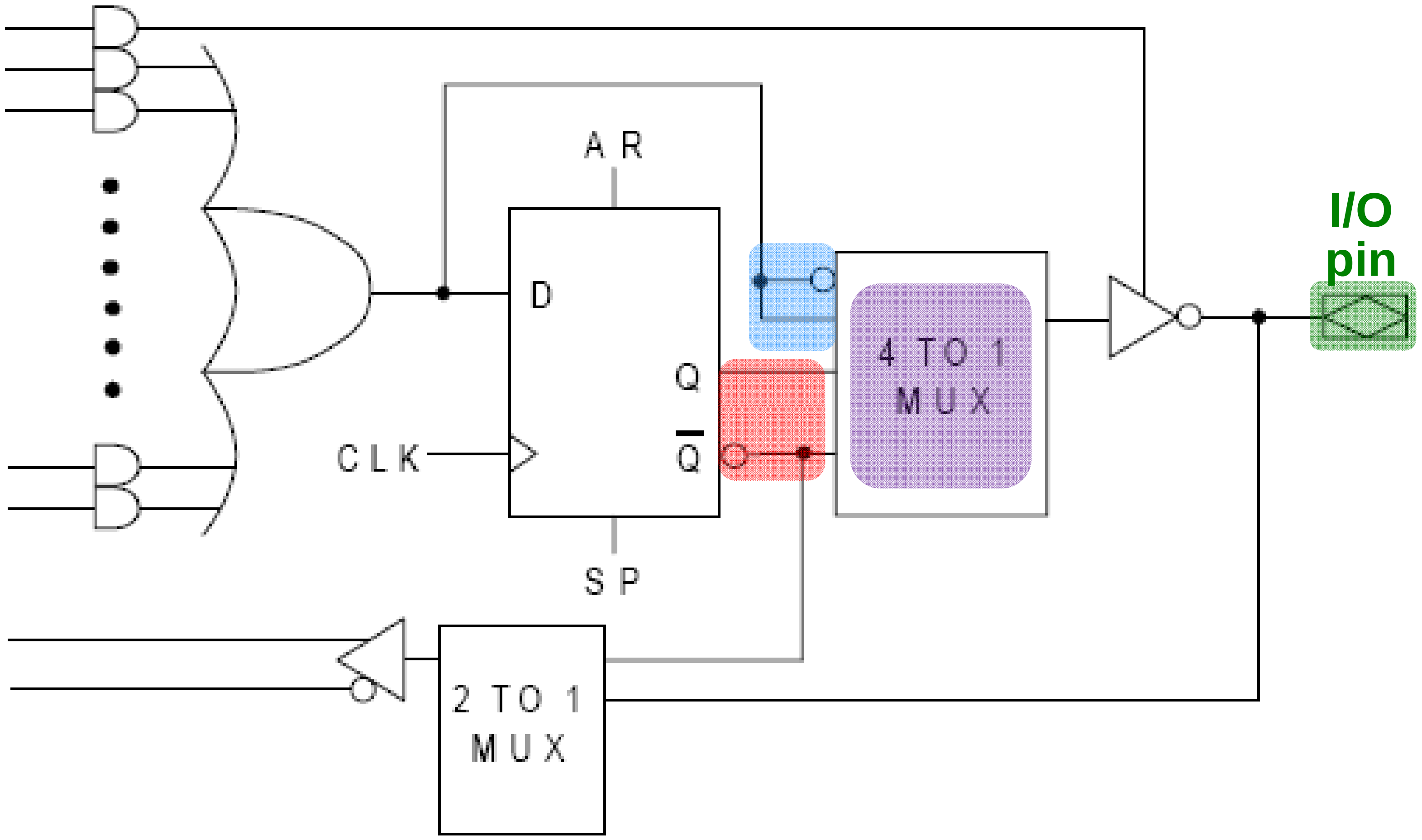
GAL22V10 Output Logic Macrocell (“OLMC”)



Note: Flip-flops are used to create sequential circuits

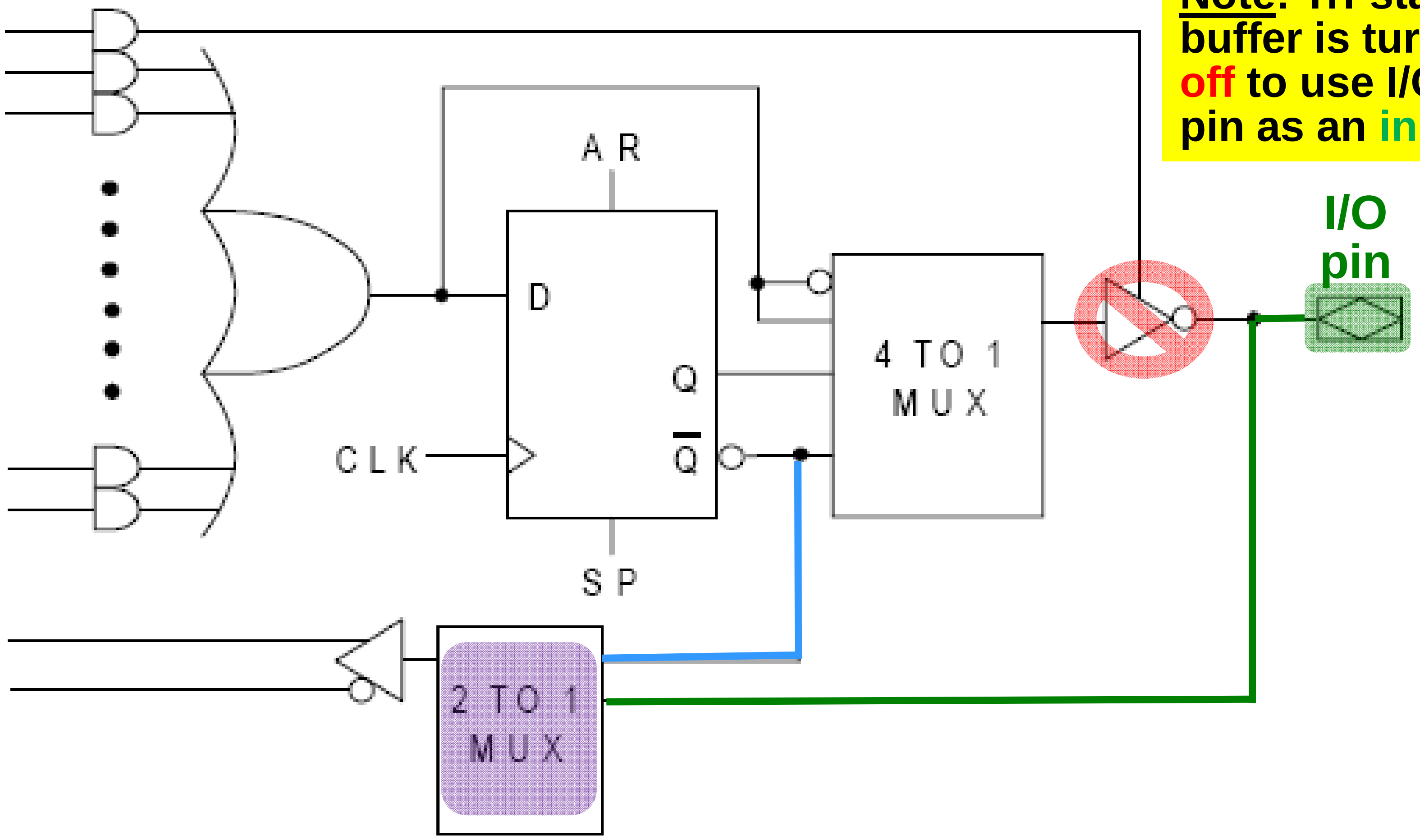
All OLMC edge-triggered D flip-flops utilize common clock (CLK) , asynchronous reset (AR), and asynchronous preset (SP) signals

GAL22V10 Output Logic Macrocell (“OLMC”)



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

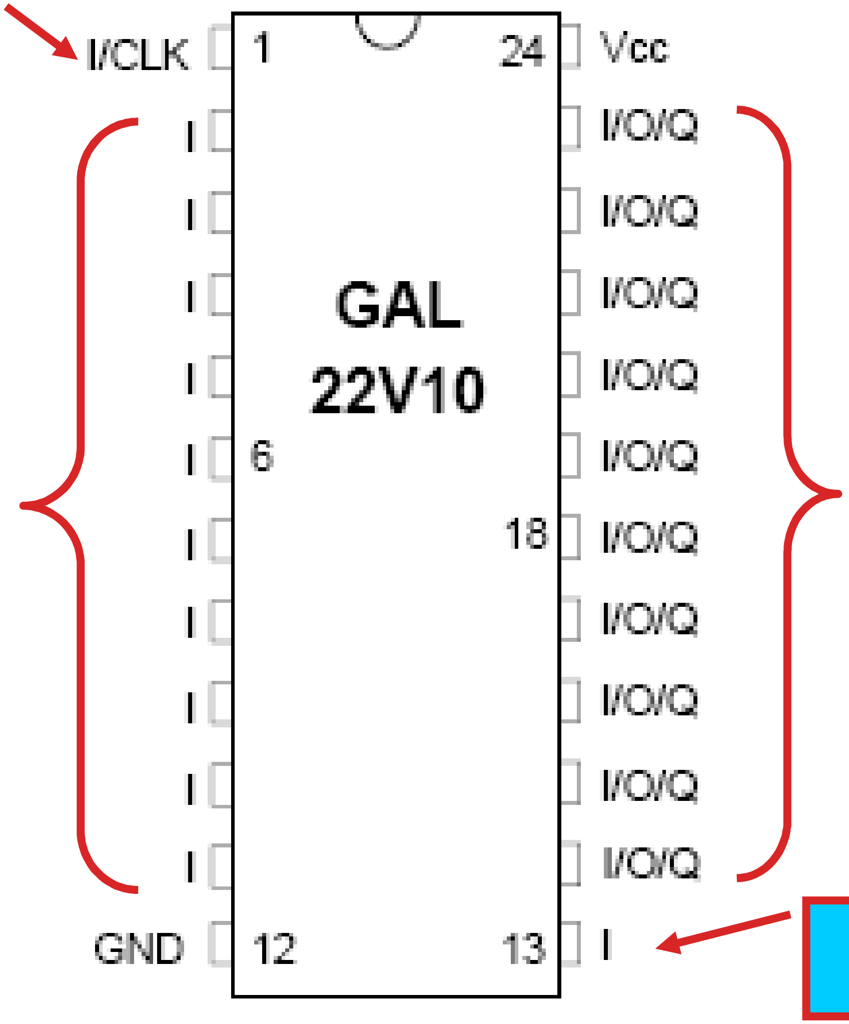
GAL22V10 Output Logic Macrocell (“OLMC”)



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

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

GAL22V10 Pinout



clock or data input

data inputs

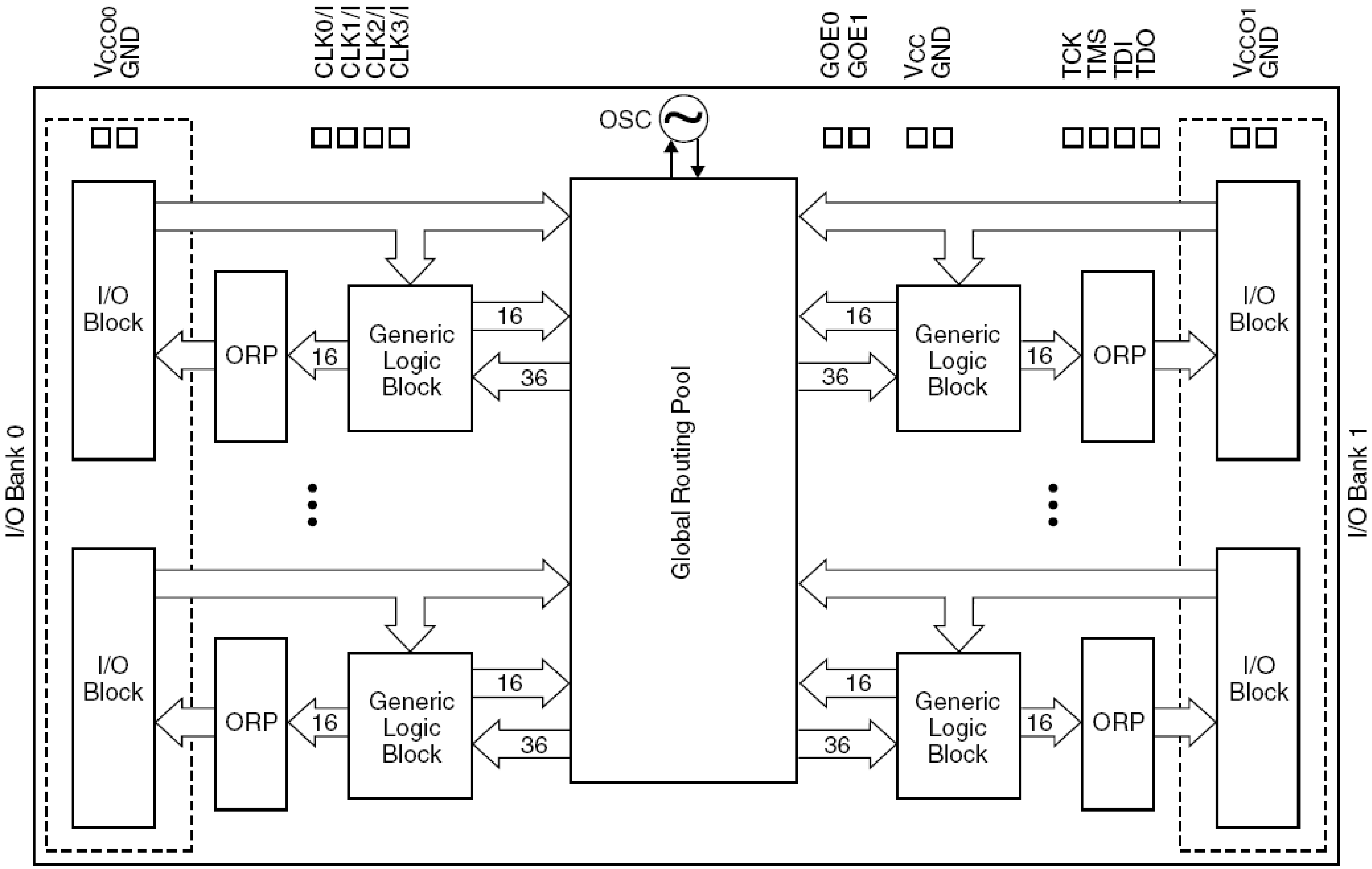
macrocell I/O pins (inputs or outputs)

data input

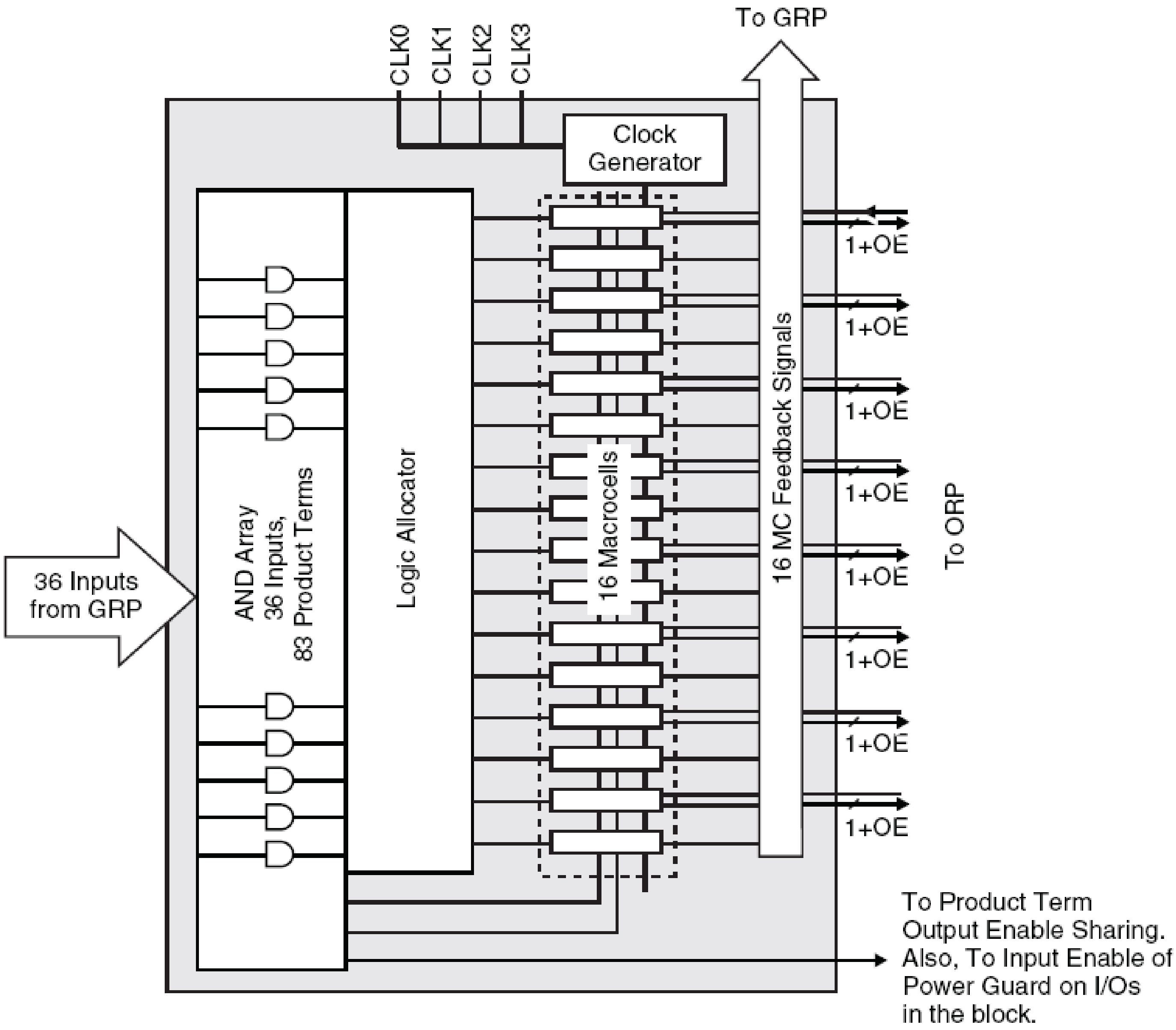
Complex PLDs (CPLDs)

- **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*)**
- **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**
- **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**

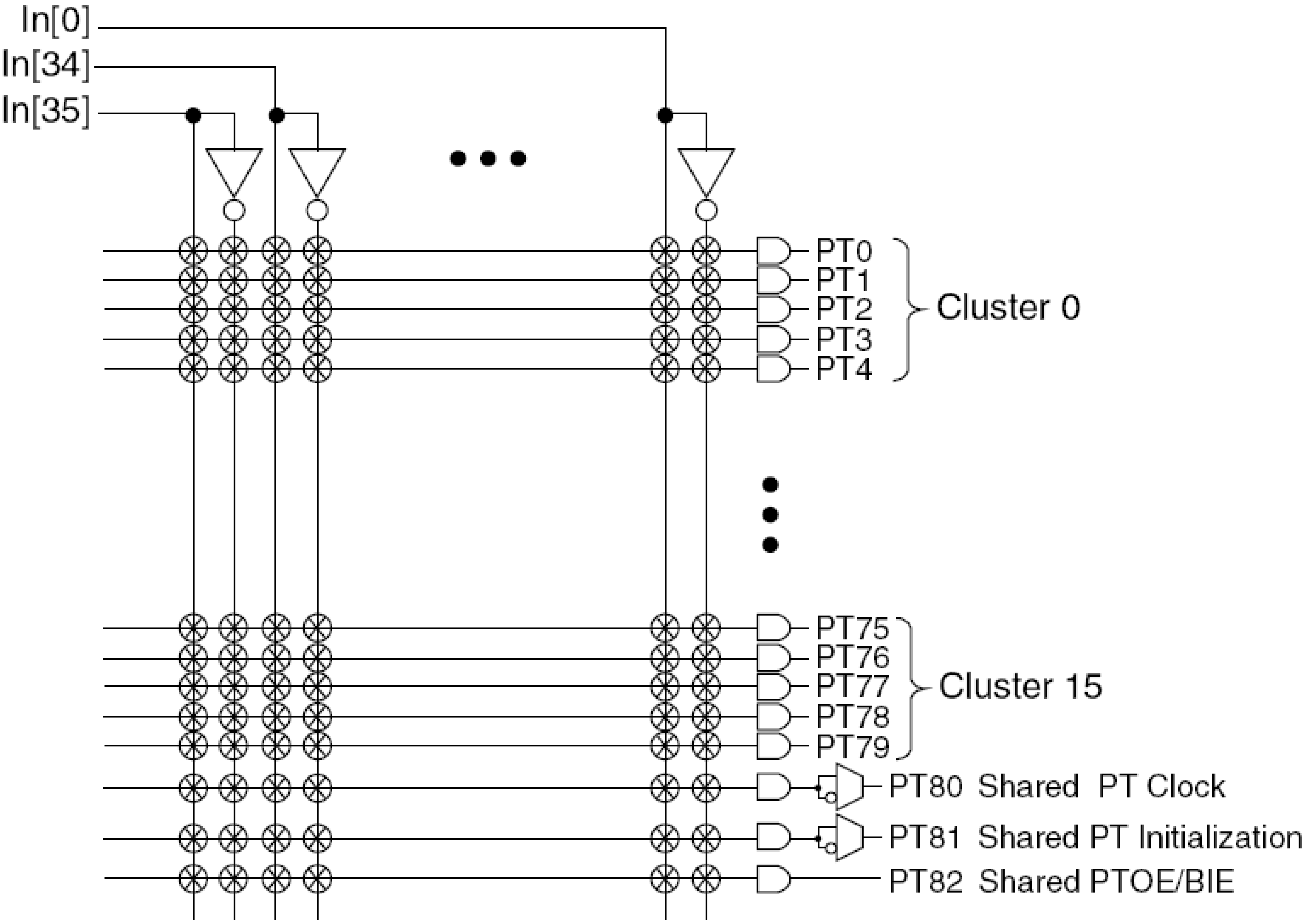
ispMACH 4000ZE Block Diagram



ispMACH 4000ZE Generic Logic Block



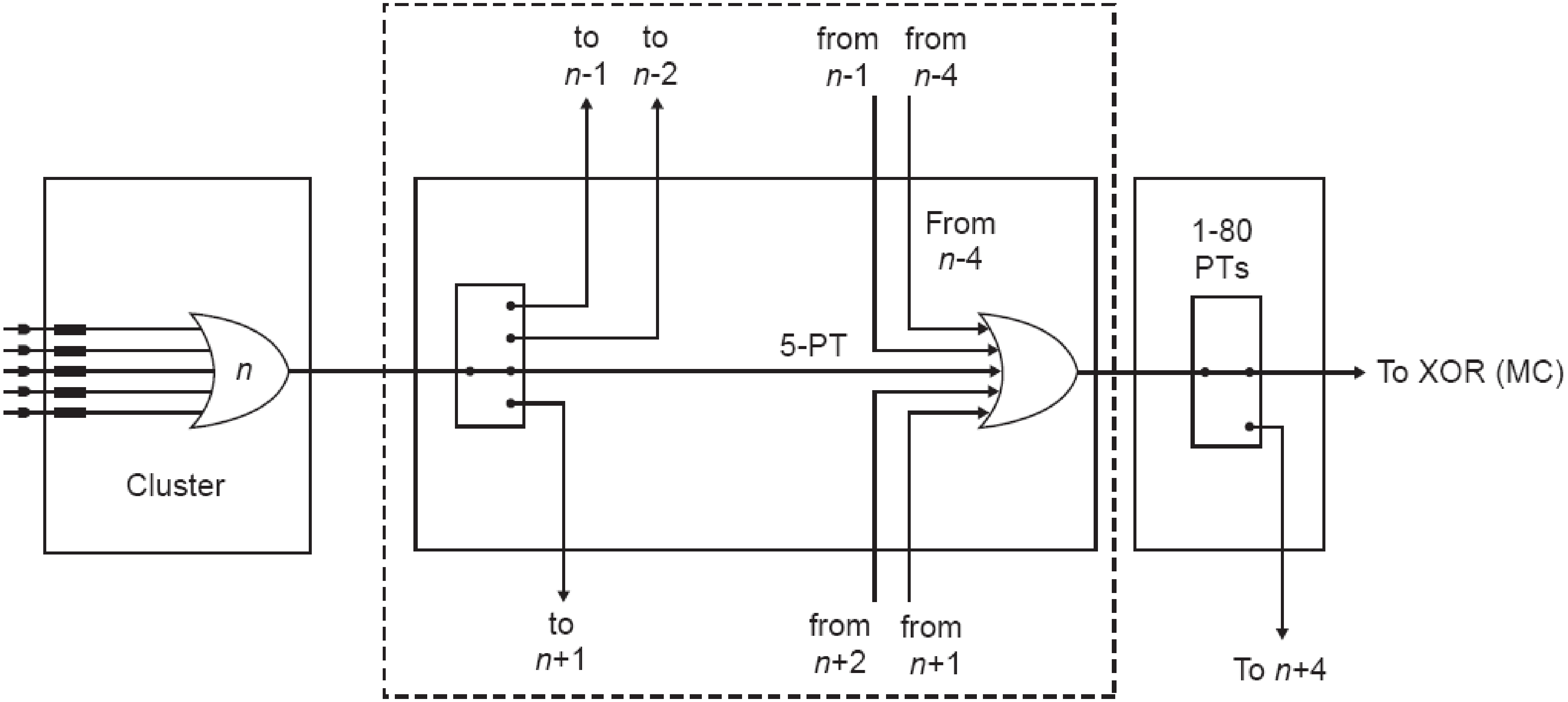
ispMACH 4000ZE 36-Input AND Array



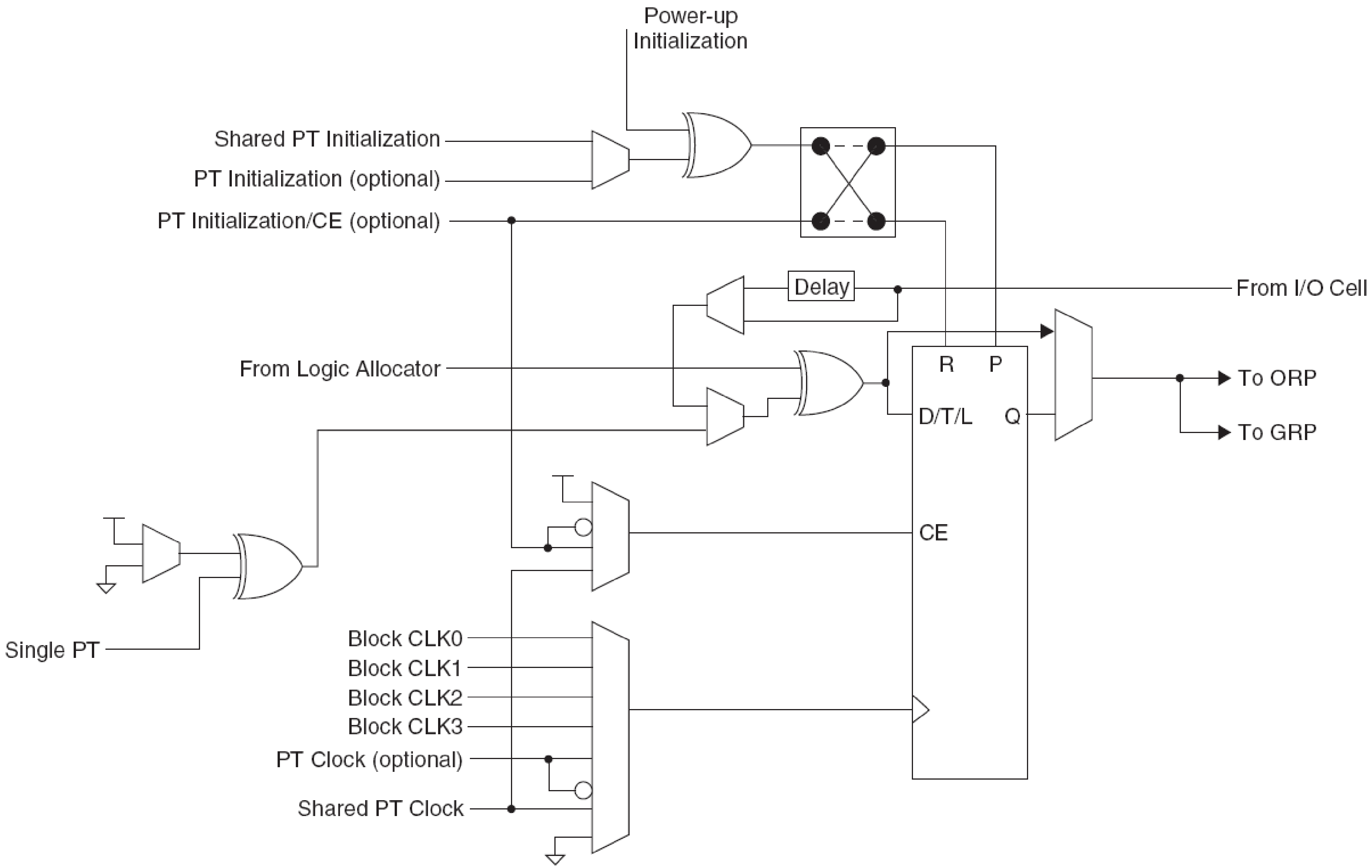
Note:

⊗ Indicates programmable fuse.

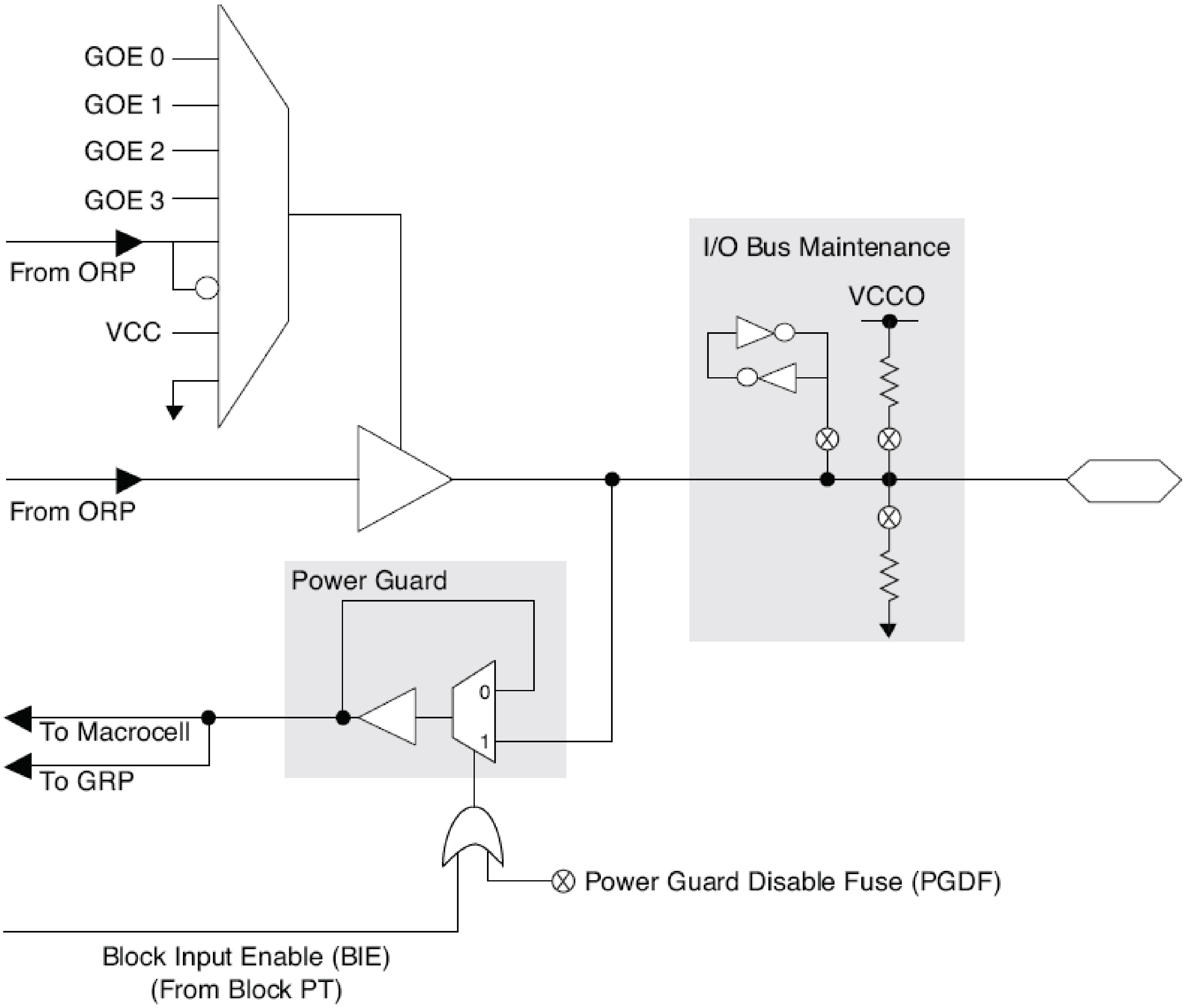
ispMACH 4000ZE Logic Allocator (Slice)



ispMACH 4000ZE Macrocell



ispMACH 4000ZE I/O Cell

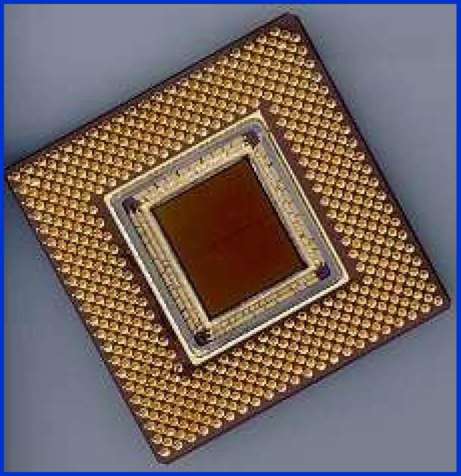


Field Programmable Gate Arrays

- 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**



Introduction to Digital System Design

Module 2-F
Hardware Description Languages

Reading Assignment:

DDPP 4th Ed., pp. 237-255

Learning Objectives:

- **List the basic features and capabilities of a hardware description language**
- **List the structural components of an ABEL program**
- **Identify operators and keywords used to create ABEL programs**
- **Write equations using ABEL syntax**
- **Define functional behavior using the `truth_table` operator in ABEL**

Outline

- Overview
- ABEL and ispLever^{TM}
- ABEL program semantics
- ABEL program structure
- ABEL symbols for logical operations
- Sample ABEL programs

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)**

ABEL and ispLever^{TM}

- To avoid the potential pitfall of using too high a level of abstraction at the introductory level, our HDL focus will be on using ABEL
- ABEL is a **hardware description language** that allows designers to specify logic functions for realization in both legacy PLDs (like the 22V10) as well as current generation CPLDs (like the ispMACH 4256ZE)
- We will use the Lattice ispLever Classic 1.5 software package in lab, which includes support for ABEL
- You can obtain your own free copy of this software from the Lattice Semiconductor web site (www.latticesemi.com)

ABEL and ispLever™

- An ABEL program 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
  - [optionally] "test vectors" that specify expected functional outputs for certain input combinations (note – code for isp devices can be recompiled and programmed into the device *so quickly* that it is often not necessary to use test vectors)
- ABEL 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)

ABEL Program Semantics

- ***identifiers*** **must begin with a letter or underscore**
- **a program file begins with a Module statement**
- **the *title statement* specifies a title string**
- ***comments*** **begin with a double quote and end with another double quote (or end of line, whichever comes first)**
- ***pin declarations*** **tell the compiler about symbolic names associated with the external pins of the device**
- **the *istype* keyword precedes a list of one or more properties, separated by commas, that tells the compiler the *type of output signal* (combinational, registered, active high, active low)**

ABEL Program Structure

- the Equations statement indicates that logic equations defining output signals as functions of input signals will follow
- *equations* are written like assignment statements (X = Y;) in a conventional programming language, with each equation terminated by a *semicolon*
- the Truth_Table statement indicates that a truth table function specification follows
- the [optional] Test_Vectors statement indicates that *test vectors* follow
- *test vectors* associate input combinations with expected output values, and are used for simulation and testing
- the End statement marks the end of a module

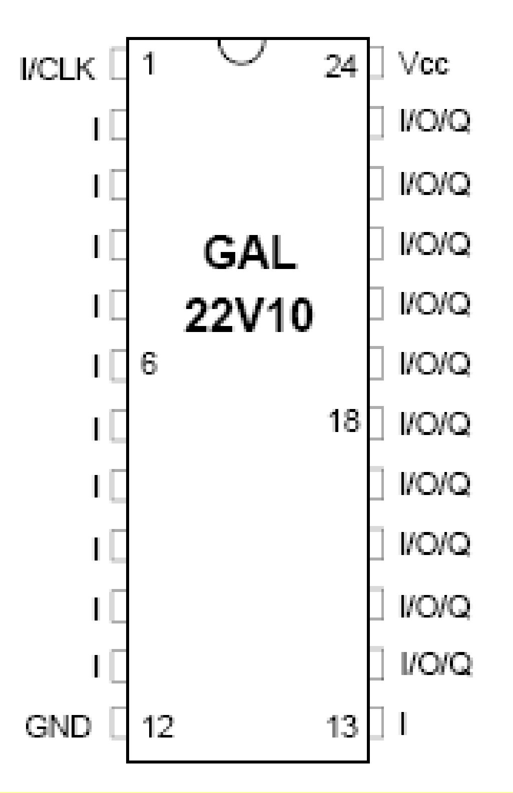
ABEL Symbols for Logical Ops

&	AND
#	OR
!	NOT
$	XOR
!$	XNOR

Note: AND (&) has precedence over OR (#)

Example ABEL Program #1A

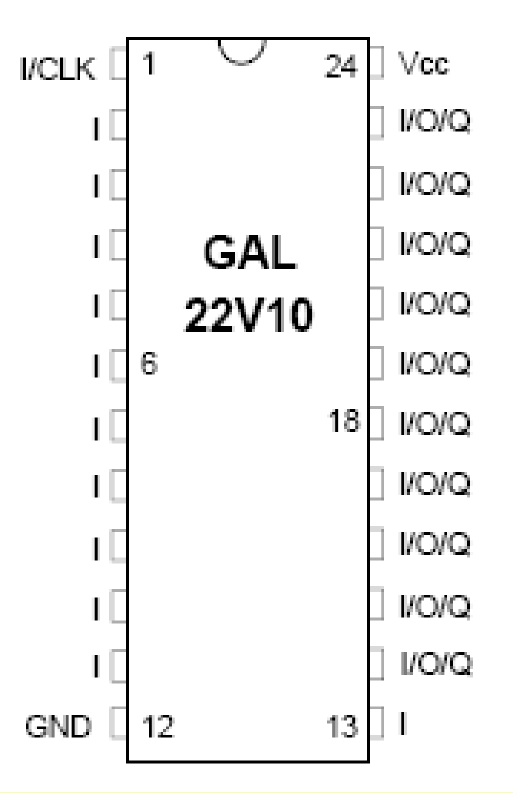
```
MODULE  abel_exA

TITLE  'ABEL Combinational Example for GAL22V10'

DECLARATIONS
" active high input pins
A   pin   2;
B   pin   3;
C   pin   4;
D   pin   5;

" active high output pins
X   pin 14 istype 'com';
Y   pin 15 istype 'com';
Z   pin 16 istype 'com';

EQUATIONS
X = A&B # !C&D;
Y = !B&D # !A&B&D;
Z = A & !B&C&!D;

END
```



<u>Note</u>: Explicit pin declarations can be omitted and automatically assigned

Example ABEL Program #1B

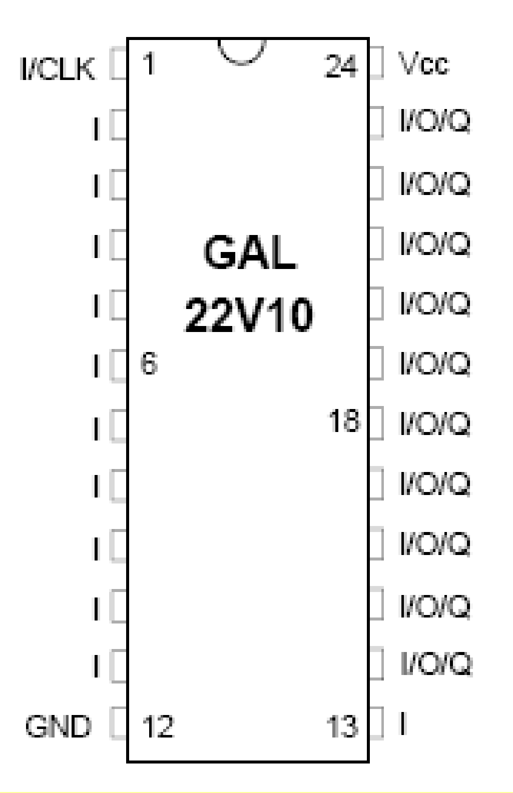
```
MODULE  abel_exB

TITLE  'ABEL Combinational Example for GAL22V10'

DECLARATIONS
" active low input pins
!A   pin   2;
!B   pin   3;
!C   pin   4;
!D   pin   5;

" active high output pins
X   pin 14 istype 'com';
Y   pin 15 istype 'com';
Z   pin 16 istype 'com';

EQUATIONS
X = A&B # !C&D;
Y = !B&D # !A&B&D;
Z = A & !B&C&!D;

END
```



Note: Explicit pin declarations can be omitted and automatically assigned

Example ABEL Program #1C

```
MODULE  abel_exC

TITLE  'ABEL Combinational Example for GAL22V10'

DECLARATIONS
" active low input pins
!A   pin   2;
!B   pin   3;
!C   pin   4;
!D   pin   5;

" active low output pins
!X   pin 14 istype 'com';
!Y   pin 15 istype 'com';
!Z   pin 16 istype 'com';

EQUATIONS
X = A&B # !C&D;
Y = !B&D # !A&B&D;
Z = A & !B&C&!D;

END
```



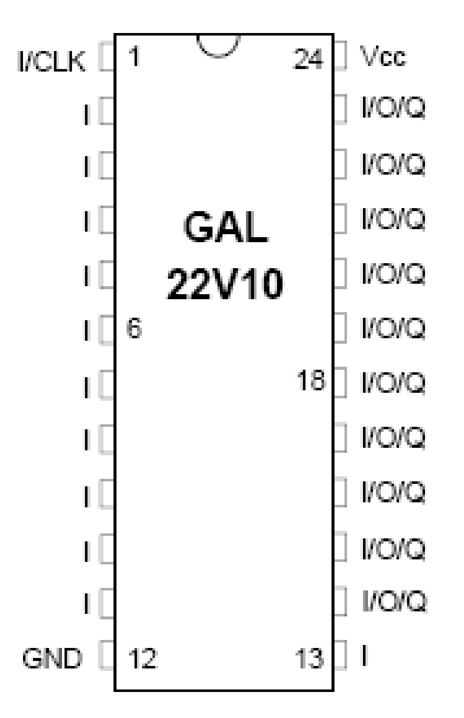
Note: Explicit pin declarations can be omitted and automatically assigned

Example ABEL Program #2

```
MODULE ttex

TITLE 'Example ABEL Truth Table'

DECLARATIONS
E, R, S, T pin 2..5; " input pins
A, B, C, D, F pin 14..18 istype 'com'; " output pins

TRUTH_TABLE ([E,R,S,T]->[A,B,C,D,F])
             [0,0,0,0]->[0,1,0,0,0];
             [0,0,0,1]->[0,0,0,1,0];
             [0,0,1,0]->[0,0,1,0,0];
             [0,0,1,1]->[0,0,0,1,0];
             [0,1,0,0]->[1,0,0,0,0];
             [0,1,0,1]->[1,0,0,0,0];
             [0,1,1,0]->[0,0,1,0,0];
             [0,1,1,1]->[1,0,0,0,0];
             [1,0,0,0]->[0,1,0,0,0];
             [1,0,0,1]->[0,1,0,0,0];
             [1,0,1,0]->[0,0,1,0,0];
             [1,0,1,1]->[0,0,0,0,1];
             [1,1,0,0]->[1,0,0,0,0];
             [1,1,0,1]->[1,0,0,0,0];
             [1,1,1,0]->[0,0,1,0,0];
             [1,1,1,1]->[1,0,0,0,0];
END
```

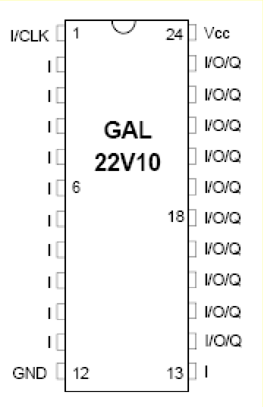
range

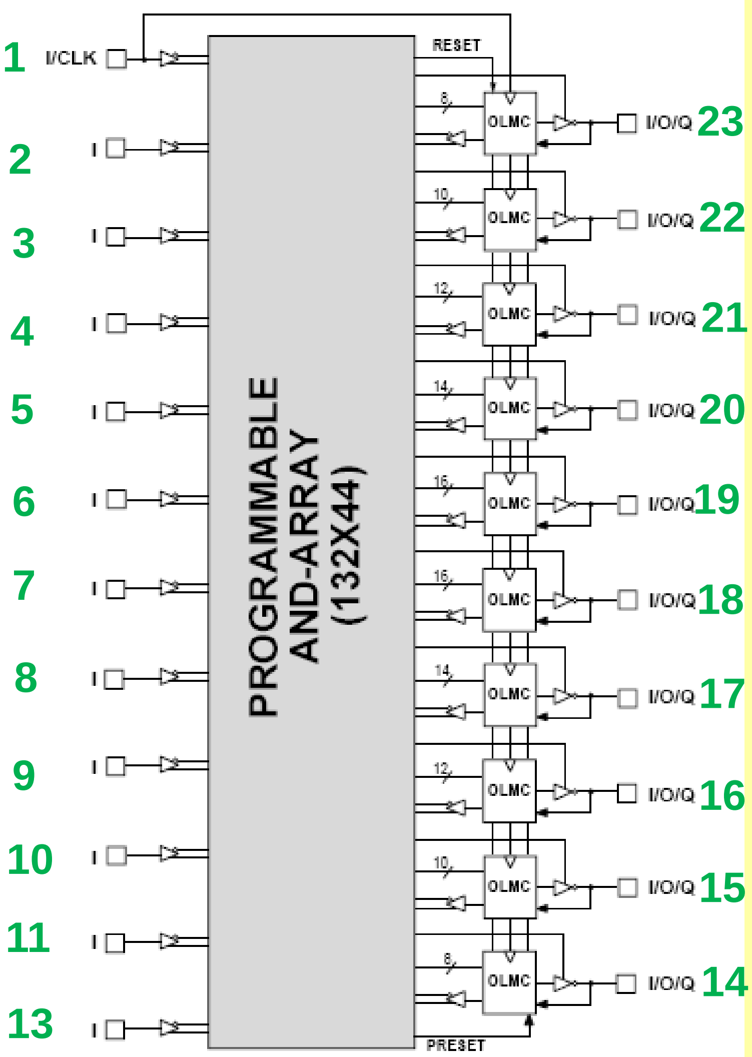
truth table declaration



Example ABEL Program #3A

```
MODULE  xor_exA

TITLE '4-variable XOR on 22V10'

DECLARATIONS
" Inputs
I1..I4 pin;

" Outputs
X  pin ___ istype 'com';

EQUATIONS
X = I1 $ I2 $ I3 $ I4;

END
```

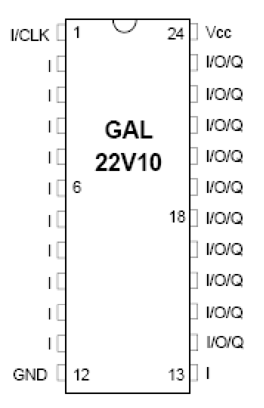


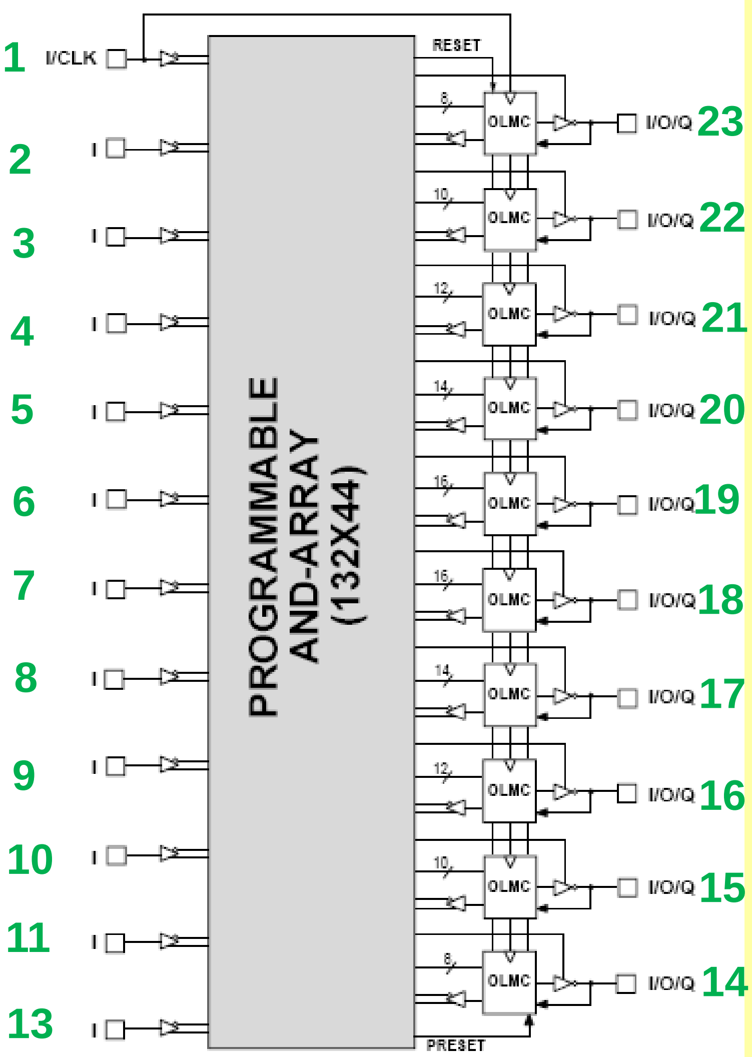


Equation requires 8 P-terms → can be realized on any 22V10 macrocell (any I/O pin)

Example ABEL Program #3B

```
MODULE  xor_exB

TITLE '5-variable XOR on 22V10'

DECLARATIONS
" Inputs
I1..I5 pin;

" Outputs
X  pin ___ istype 'com';

EQUATIONS
X = I1 $ I2 $ I3 $ I4 $ I5;

END
```

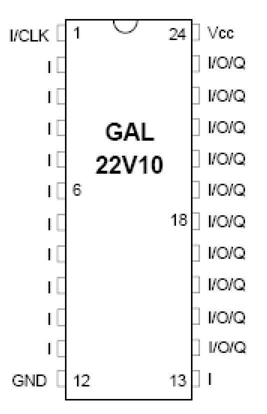


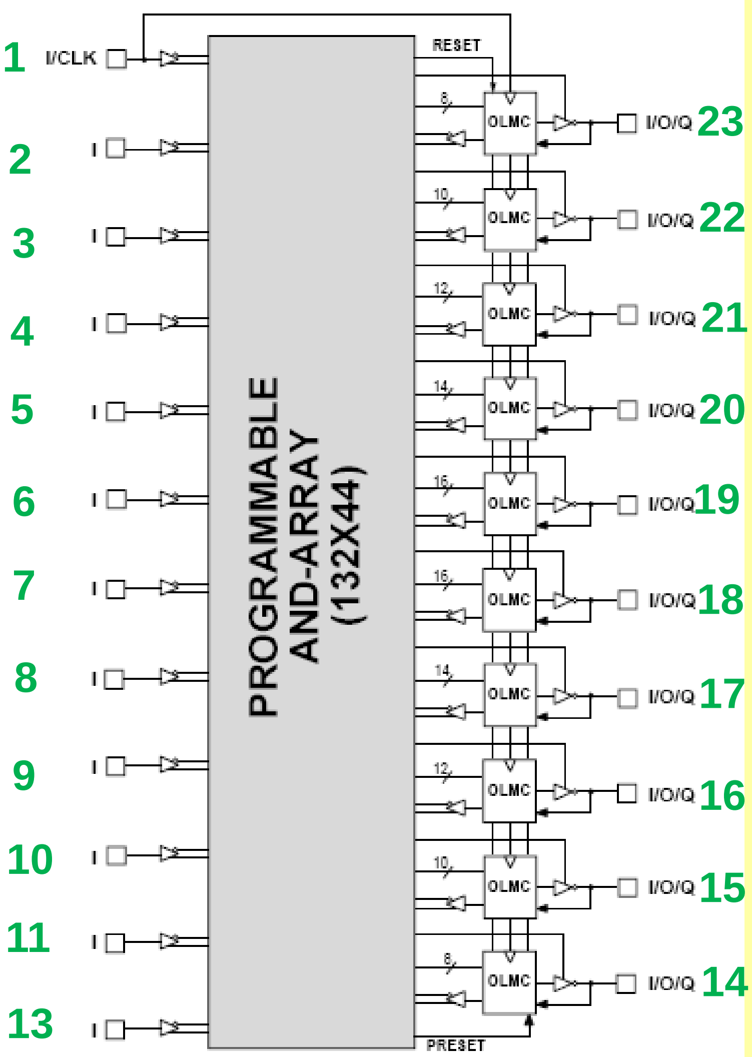


Equation requires 16 P-terms → can be realized on macrocells associated with I/O pins 18 & 19

Example ABEL Program #3C

```
MODULE  xor_exC

TITLE '10-variable XOR on 22V10'

DECLARATIONS
" Inputs
I1..I10 pin;

" Outputs
X  pin 18 istype 'com';
Y  pin 19 istype 'com';
Z  pin 23 istype 'com';

EQUATIONS
X = I1$I2$I3$I4$I5; " 16 P-terms
Y = I6$I7$I8$I9$10; " 16 P-terms
Z = X $ Y;          "  2 P-terms
END
```





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

Clicker Quiz

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

A. **X2**

B. **2X**

C. **XY**

D. **_XY**

E. none of the above

2. Which of the following is an example of a **range?**

A. **X0..X3**

B. **[X0..X3]**

C. **GA..GE**

D. **[GA..GE]**

E. none of the above

3. For **pin declarations**, which of the following statements is **not** true?

A. pin declarations associate symbolic names with the device’s physical pins

B. pin numbers are optional

C. if the pin number is omitted, 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 **equations** are placed in the EQUATIONS section of an ABEL program does not matter.

A. true

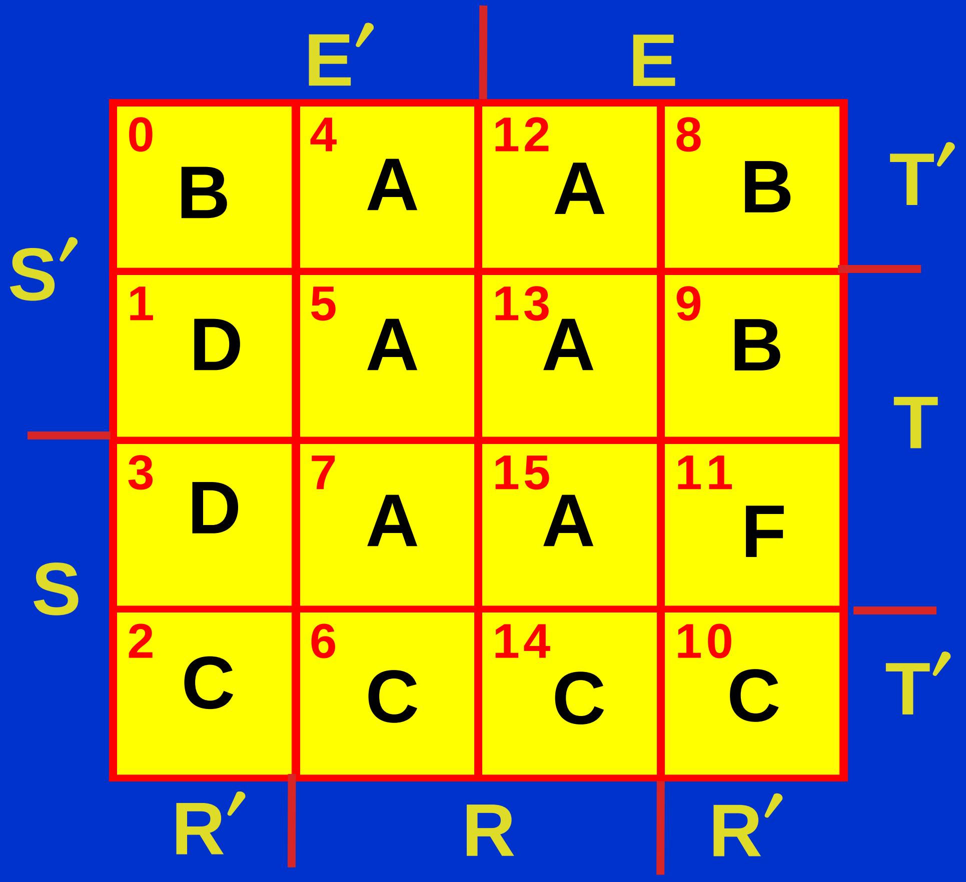
B. false

Example–Raul's "Crazy Grader"

You have 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”

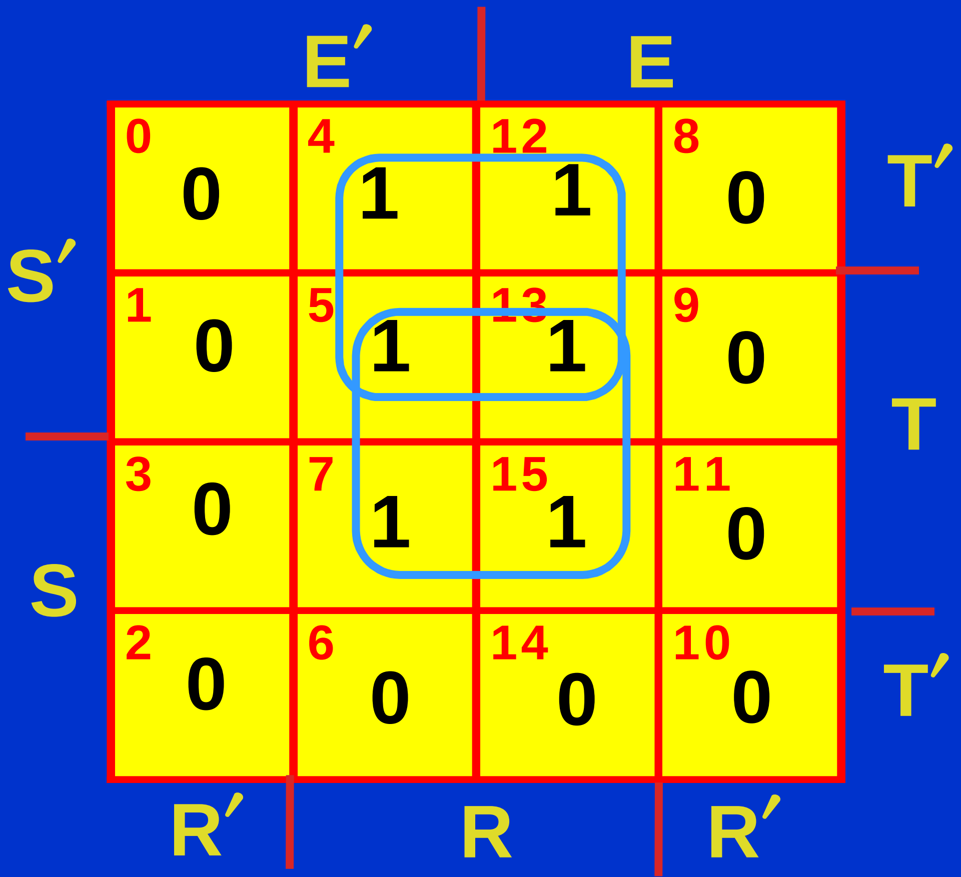


	E′		E		
S′	0 B	4 A	12 A	8 B	T′
	1 D	5 A	13 A	9 B	T
S	3 D	7 A	15 A	11 F	
	2 C	6 C	14 C	10 C	T′
	R′	R		R′	

Options

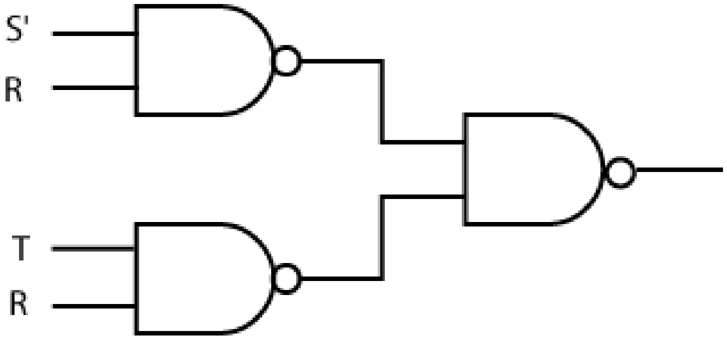
- Map and minimize all 5 functions, implement with several discrete CMOS ICs, subject to the following limitations:
  - only “true” variables are available
  - only SSI chips in digital kit can be used
    - 7400 quad 2-input NAND
    - 7402 quad 2-input NOR
    - 7404 hex inverter
    - 7410 triple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

Working K-Map for “A” – SoP



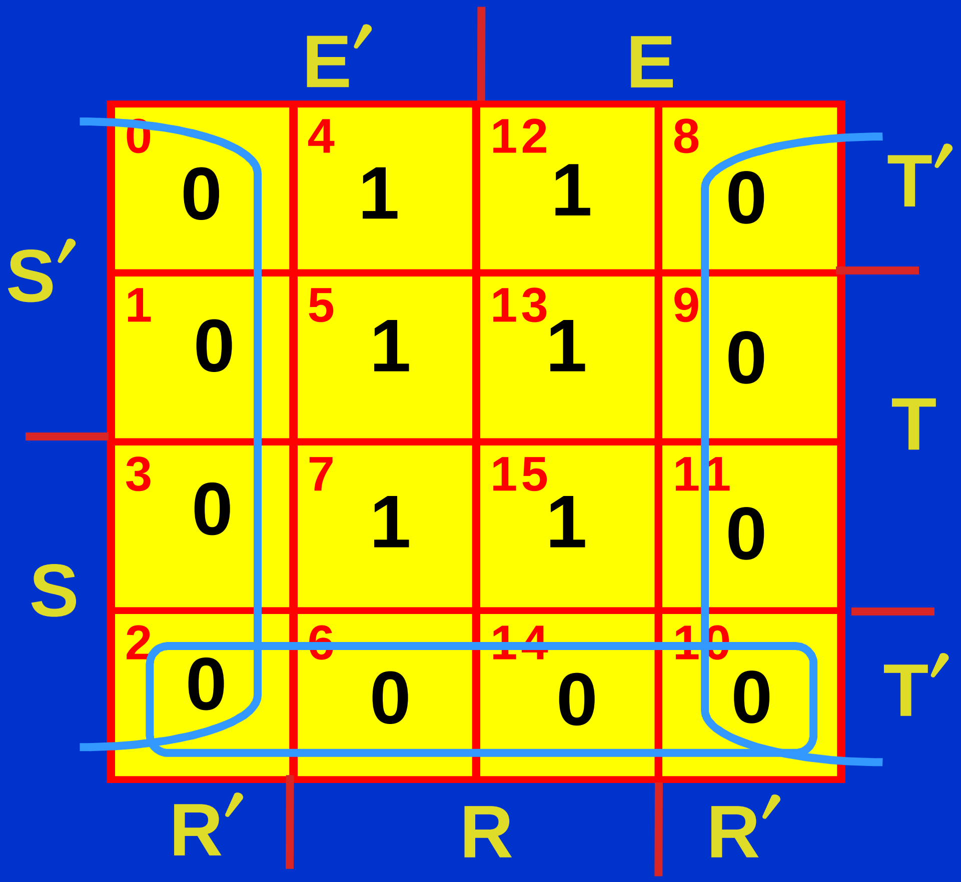
	E′	E′	E	E	
S′	0 0	4 1	12 1	8 0	T′
S′	1 0	5 1	13 1	9 0	T
S	3 0	7 1	15 1	11 0	T
S	2 0	6 0	14 0	10 0	T′
	R′	R	R	R′	

A = S′•R + T•R



COST = 6 inputs + 3 outputs = 9

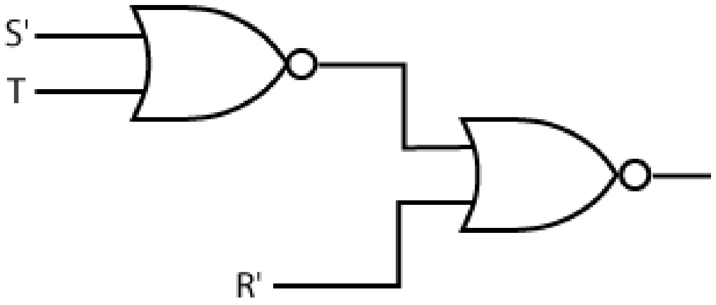
Working K-Map for “A” – PoS



	E′		E		
S′	0 0	4 1	12 1	8 0	T′
	1 0	5 1	13 1	9 0	T
S	3 0	7 1	15 1	11 0	
	2 0	6 0	14 0	10 0	T′
	R′	R		R′	

A′ = R′ + S•T′

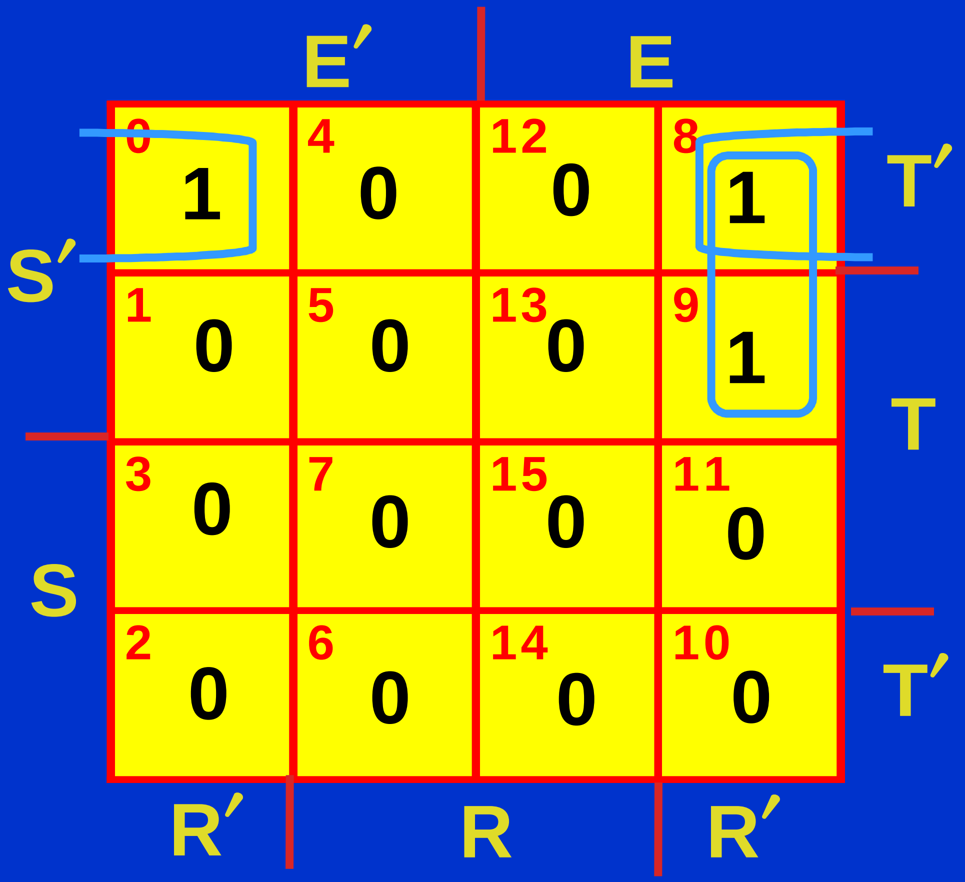
A = R • (S′ + T)



COST = 4 inputs + 2 outputs = 6

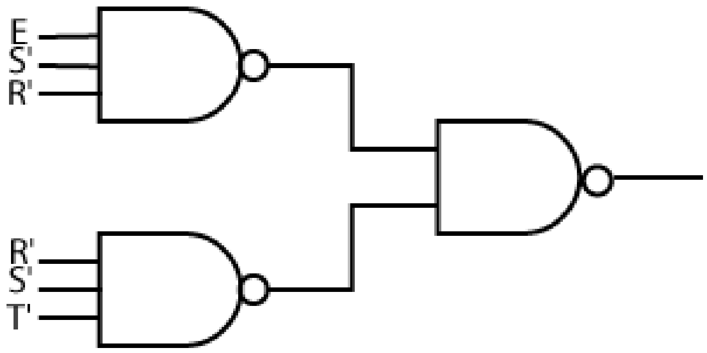
Cheaper than SoP

Working K-Map for “B” – SoP



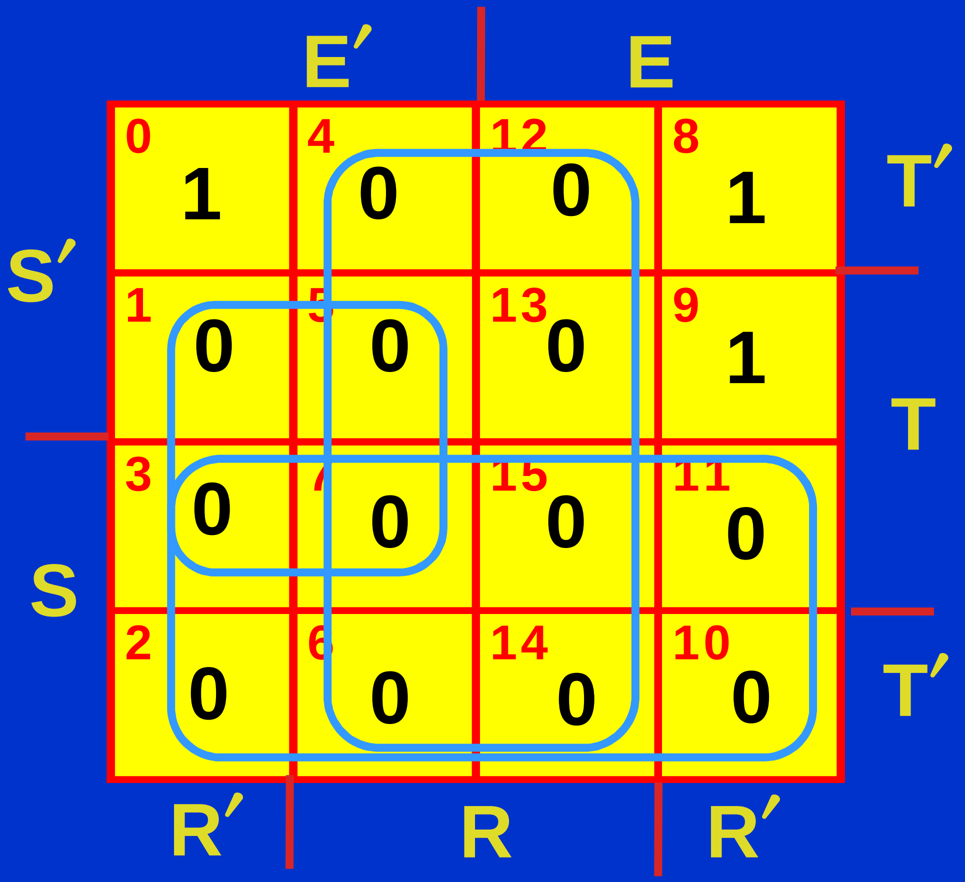
		E′		E	
		R′	R	R	R′
S′	T′	0 1	4 0	12 0	8 1
S′	T	1 0	5 0	13 0	9 1
S	T	3 0	7 0	15 0	11 0
S	T′	2 0	6 0	14 0	10 0

B = E•S′•R′ + R′•S′•T′



COST = 8 inputs + 3 outputs = 11

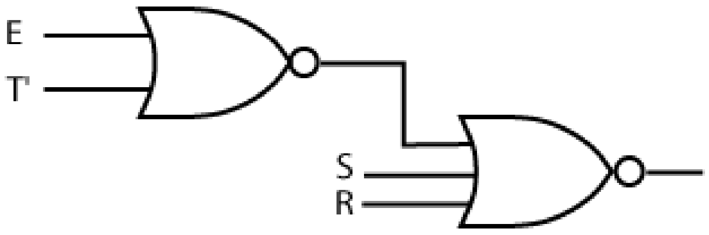
Working K-Map for “B” – PoS



		E′		E	
		R′	R	R	R′
S′	T′	0 1	4 0	12 0	8 1
	T	1 0	5 0	13 0	9 1
S	T	3 0	7 0	15 0	11 0
	T′	2 0	6 0	14 0	10 0

B′ = S + R + E′•T

B = S′• R′•(E+T′)



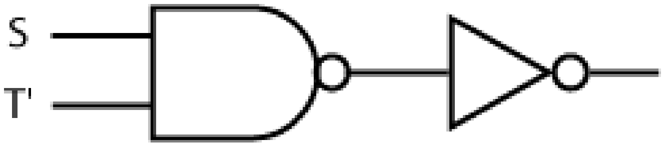
COST = 5 inputs
+ 2 outputs = 7

Cheaper than SoP

Working K-Map for “C” – SoP

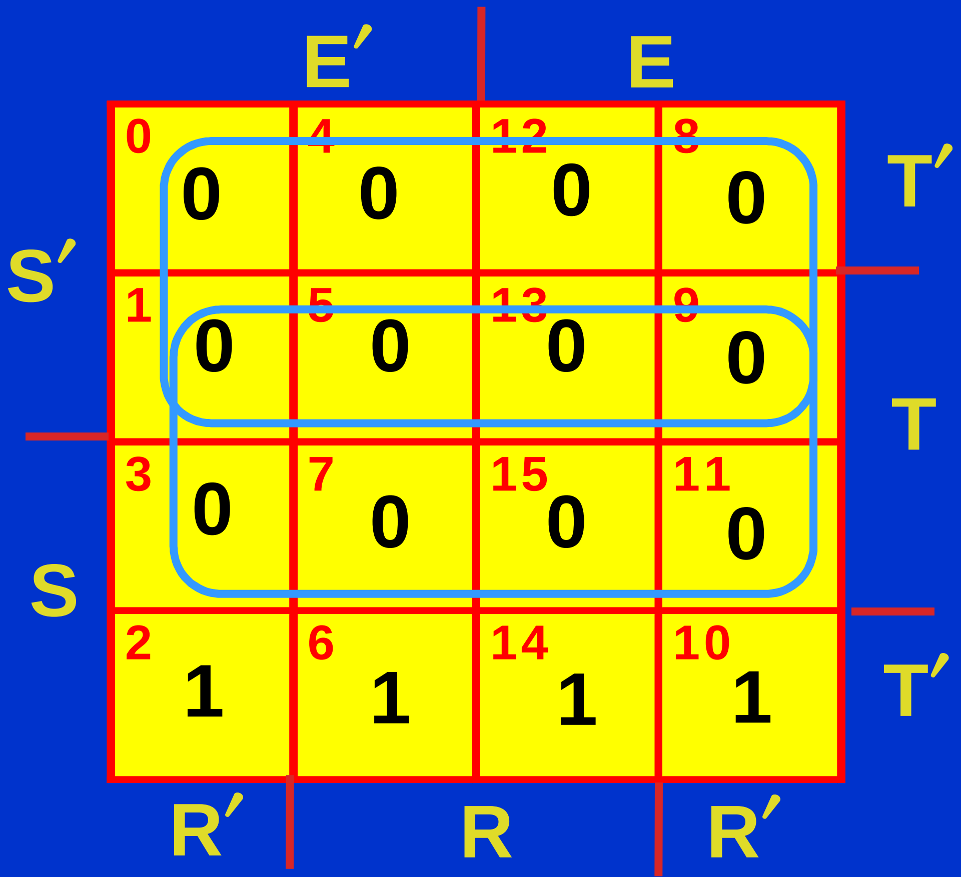
	E′		E		
S′	0 0	4 0	12 0	8 0	T′
	1 0	5 0	13 0	9 0	T
S	3 0	7 0	15 0	11 0	
	2 1	6 1	14 1	10 1	T′
	R′	R		R′	

C = S•T′



COST = 3 inputs + 2 outputs = 5

Working K-Map for “C” – PoS

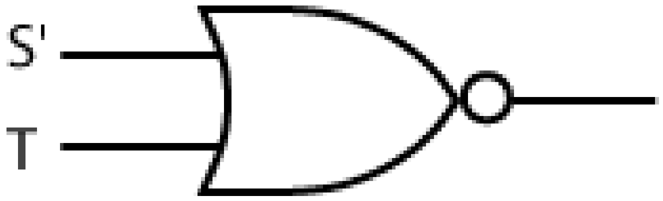


	E′		E		
S′	0 0	4 0	12 0	8 0	T′
	1 0	5 0	13 0	9 0	T
S	3 0	7 0	15 0	11 0	
	2 1	6 1	14 1	10 1	T′
	R′	R		R′	

Cheaper than SoP

$C' = S' + T$

$C = S \bullet T'$

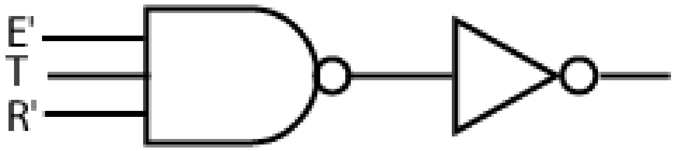


COST = 2 inputs + 1 output = 3

Working K-Map for “D” – SoP

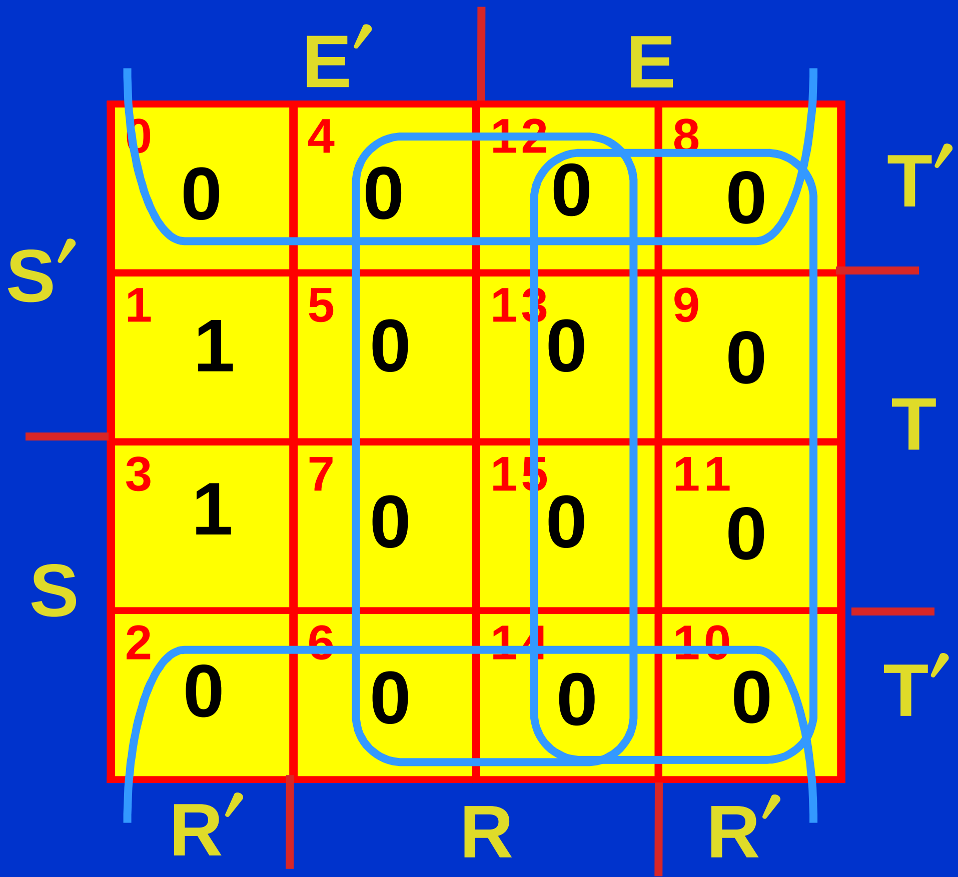
	E′		E		
S′	0 0	4 0	12 0	8 0	T′
	1 1	5 0	13 0	9 0	T
S	3 1	7 0	15 0	11 0	
	2 0	6 0	14 0	10 0	T′
	R′	R		R′	

D = E′•T•R′



COST = 4 inputs + 2 outputs = 6

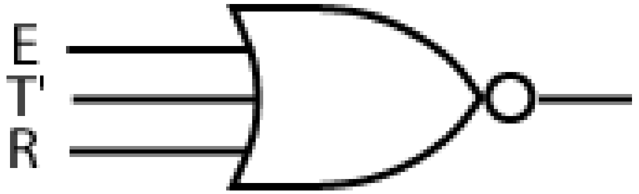
Working K-Map for “D” – PoS



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

Cheaper than SoP

D = E′•T•R′

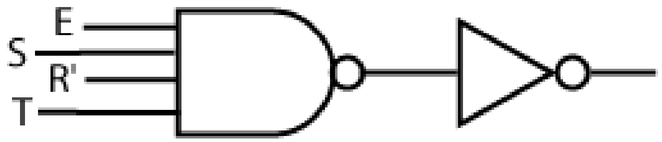


COST = 3 inputs + 1 output = 4

Working K-Map for “F” - SoP

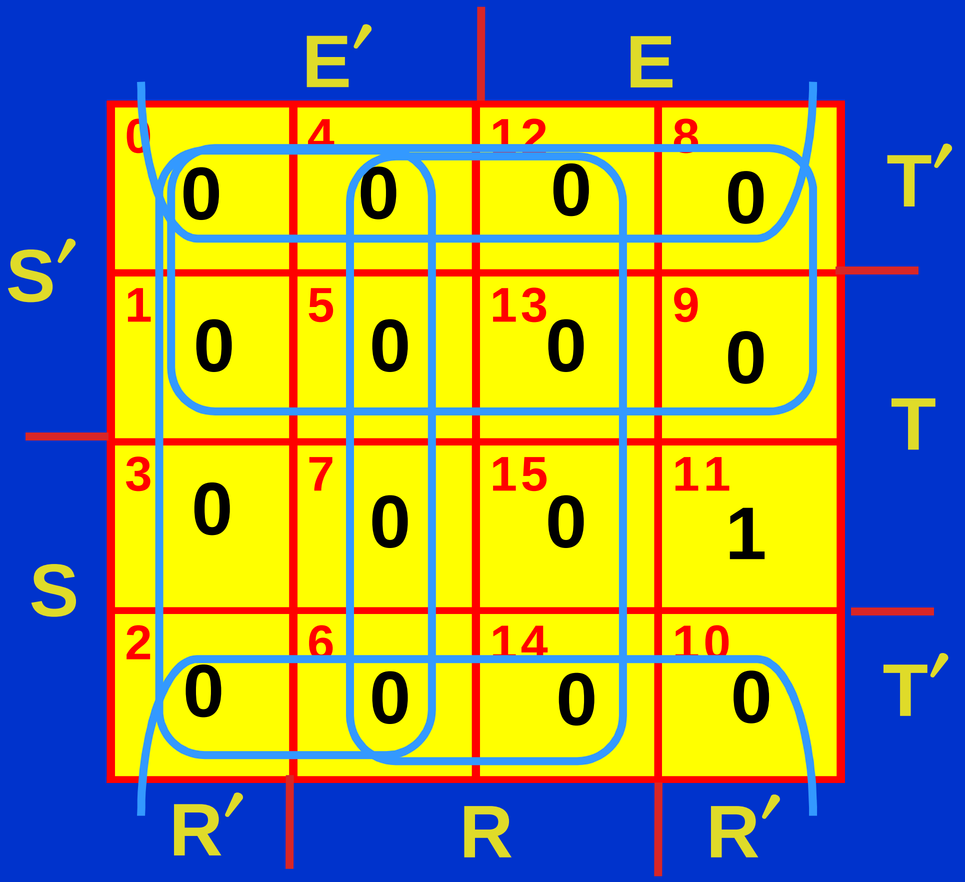
		E′		E		
S′		0 0	4 0	12 0	8 0	T′
S′		1 0	5 0	13 0	9 0	T
S		3 0	7 0	15 0	11 1	T
S		2 0	6 0	14 0	10 0	T′
		R′	R	R	R′	

F = E•S•R′•T



COST = 5 inputs
+ 2 outputs = 7

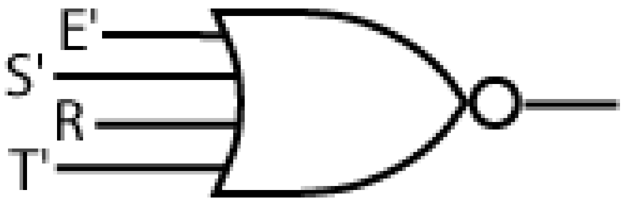
Working K-Map for “F” - PoS



		E′		E		
S′		0 0	4 0	12 0	8 0	T′
		1 0	5 0	13 0	9 0	T
S		3 0	7 0	15 0	11 1	
		2 0	6 0	14 0	10 0	T′
		R′		R		R′

F′ = E′+S′+R+T′

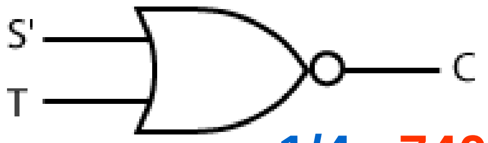
F = E•S•R′•T



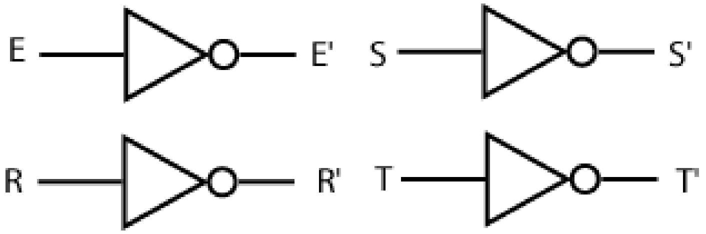
COST = 4 inputs + 1 output = 5

Cheaper than SoP

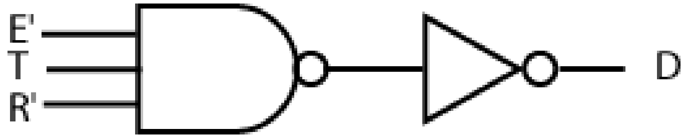
SSI “Final Answer”…



1/4 - 7402

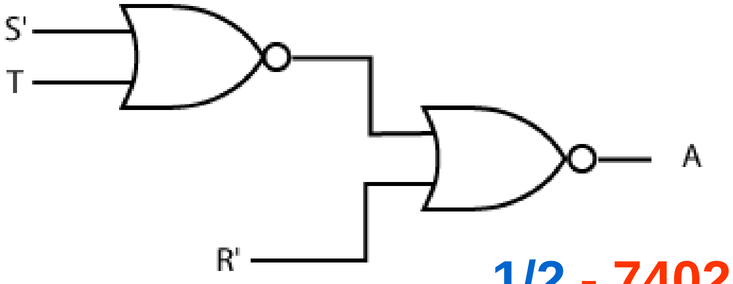


2/3 - 7404

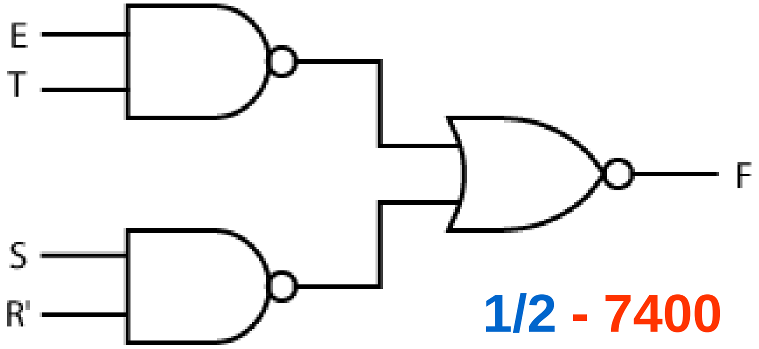


1/3 - 7410

1/6 - 7404

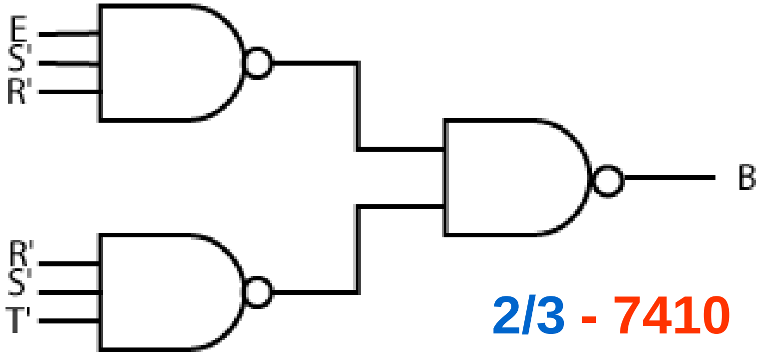


1/2 - 7402



1/2 - 7400

1/4 - 7402



2/3 - 7410

1/4 - 7400

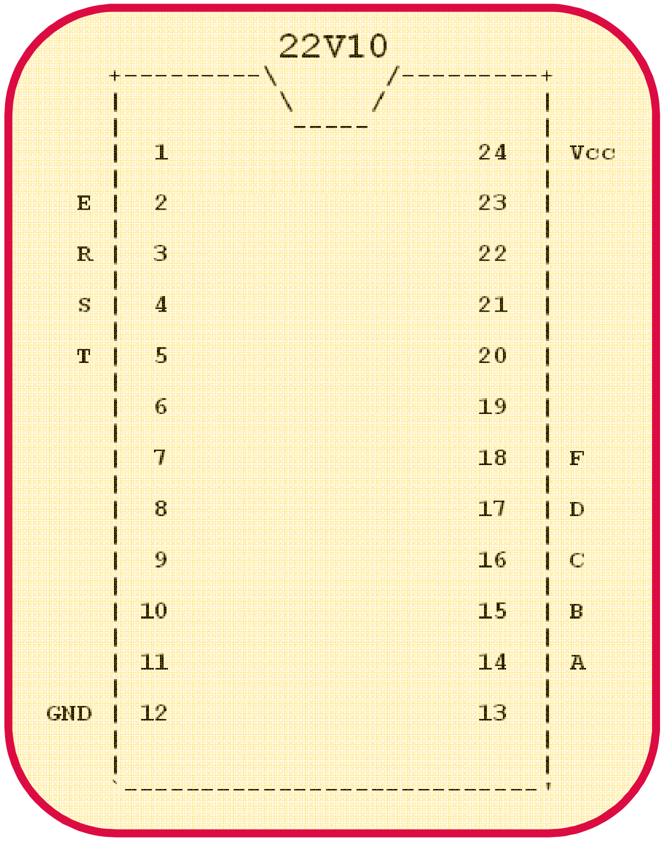
3/4 - 7400 5/6 - 7404

1 - 7402 1 - 7410

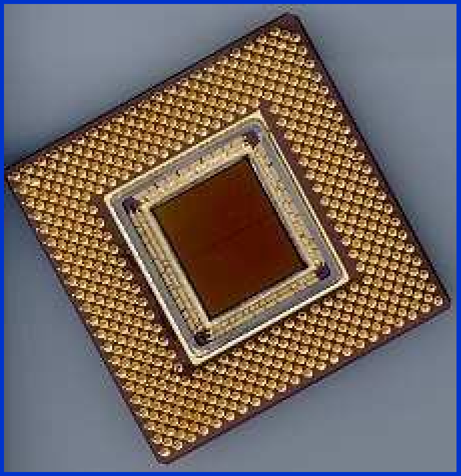
4 integrated circuits total

ABEL “Final Answer”…

```
MODULE gameshow

TITLE 'Who Wants to be a Digijock?'

DECLARATIONS
E, R, S, T pin 2..5; " input pins
A, B, C, D, F pin 14..18 istype 'com'; " output pins

TRUTH_TABLE ([E,R,S,T]->[A,B,C,D,F])
             [0,0,0,0]->[0,1,0,0,0];
             [0,0,0,1]->[0,0,0,1,0];
             [0,0,1,0]->[0,0,1,0,0];
             [0,0,1,1]->[0,0,0,1,0];
             [0,1,0,0]->[1,0,0,0,0];
             [0,1,0,1]->[1,0,0,0,0];
             [0,1,1,0]->[0,0,1,0,0];
             [0,1,1,1]->[1,0,0,0,0];
             [1,0,0,0]->[0,1,0,0,0];
             [1,0,0,1]->[0,1,0,0,0];
             [1,0,1,0]->[0,0,1,0,0];
             [1,0,1,1]->[0,0,0,0,1];
             [1,1,0,0]->[1,0,0,0,0];
             [1,1,0,1]->[1,0,0,0,0];
             [1,1,1,0]->[0,0,1,0,0];
             [1,1,1,1]->[1,0,0,0,0];
END
```



```
                     22V10
        +----------\       /----------+
        |           \     /           |
        |            -----            |
        |   1                    24   |  Vcc
   E    |   2                    23   |
   R    |   3                    22   |
   S    |   4                    21   |
   T    |   5                    20   |
        |   6                    19   |
        |   7                    18   |  F
        |   8                    17   |  D
        |   9                    16   |  C
        |  10                    15   |  B
        |  11                    14   |  A
  GND   |  12                    13   |
        |                             |
        `-----------------------------'
```



Introduction to Digital System Design

Module 2-G
Combinational Building Blocks: Decoders / Demultiplexers

Reading Assignment:

DDPP 4th Ed., pp. 384-398

Learning Objectives:

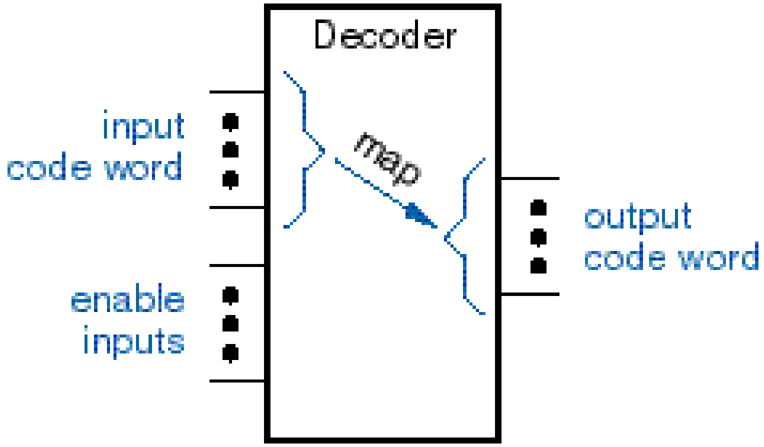
- **Define the function of a decoder (demultiplexer) and describe how it can be used as a combinational building block**
- **Illustrate how a decoder can be used to realize an arbitrary Boolean function**

Outline

- Overview
- Binary decoders
- Decoders in ABEL
- 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

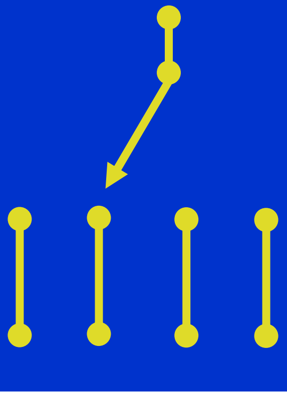


Overview

- **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*)**

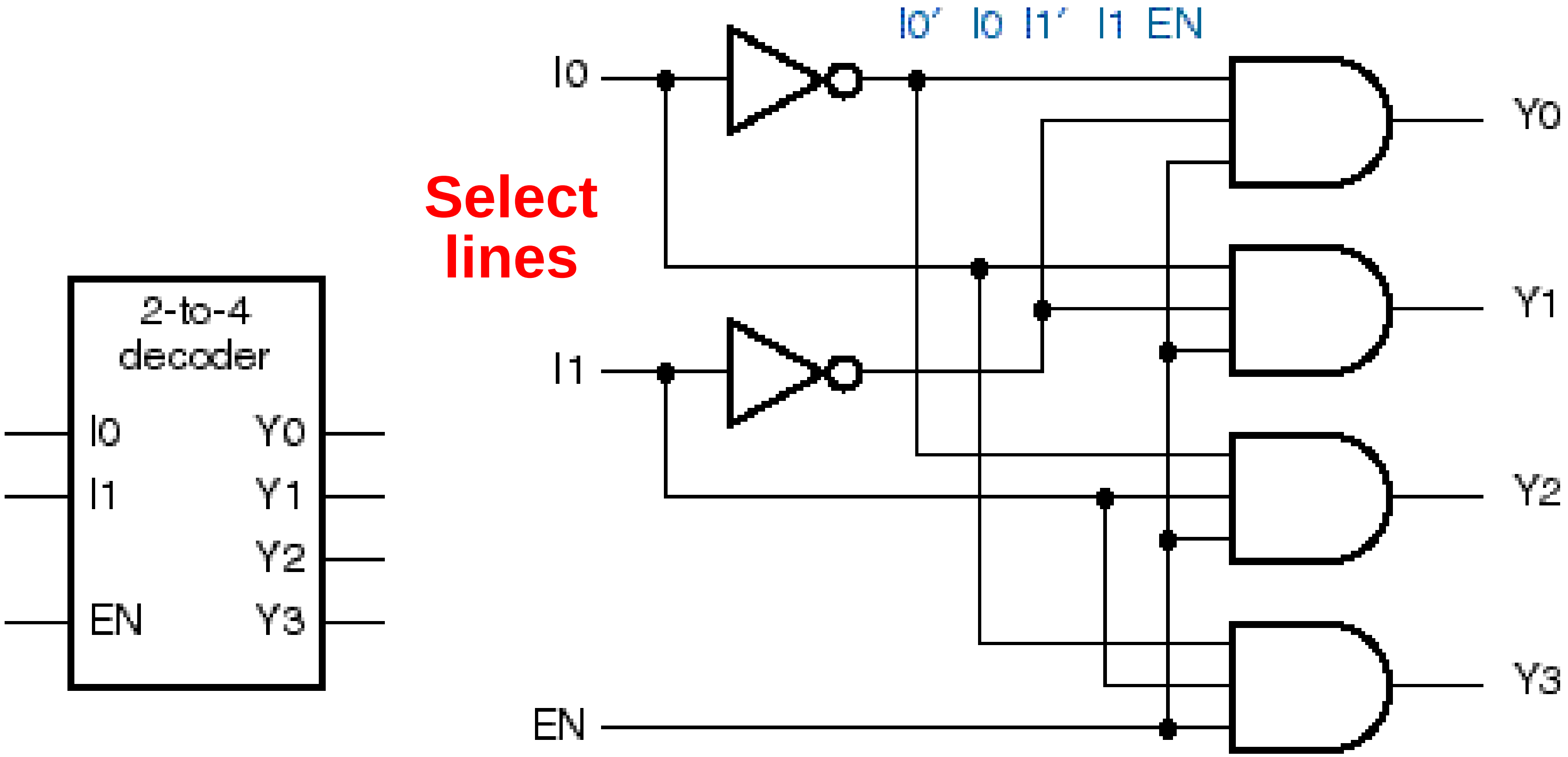
Binary Decoders

- 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



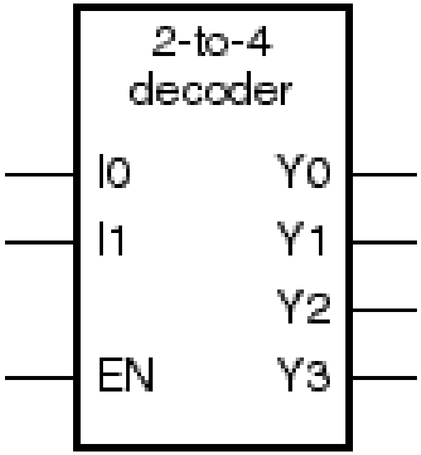
Select lines

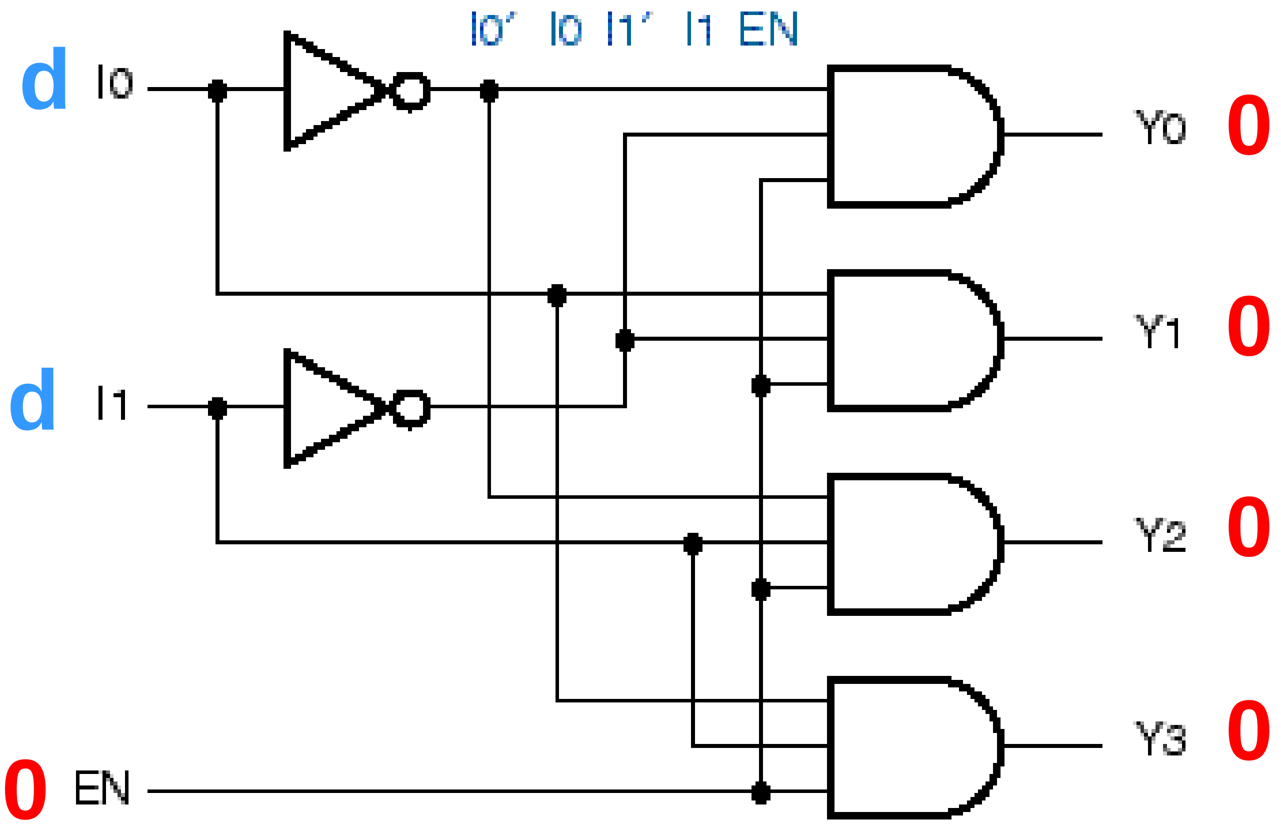
Enable

Decoded Outputs

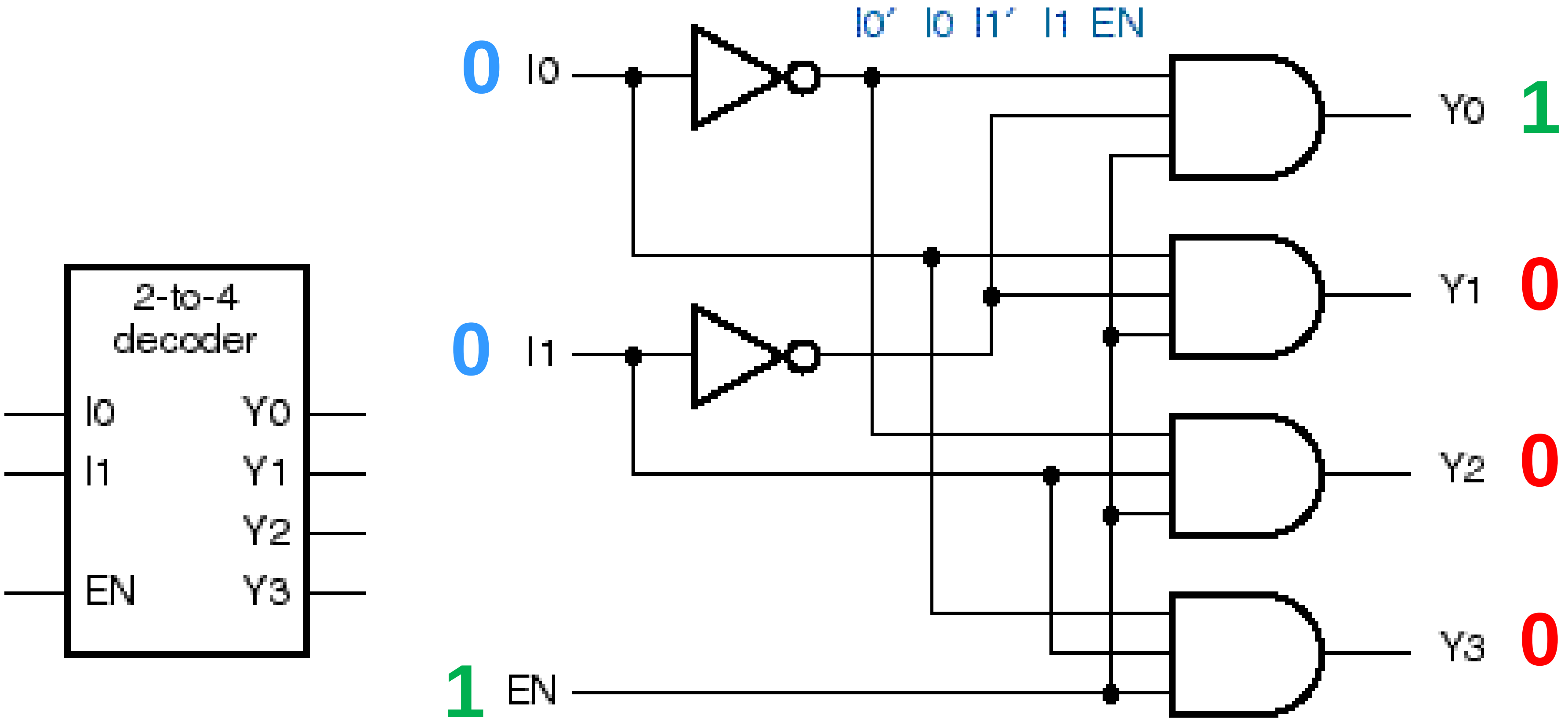
Note that **EN** can also be construed as a **digital input** that is routed to the **selected output**, in which case the circuit would be referred to as a (1:4) **demultiplexer**

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

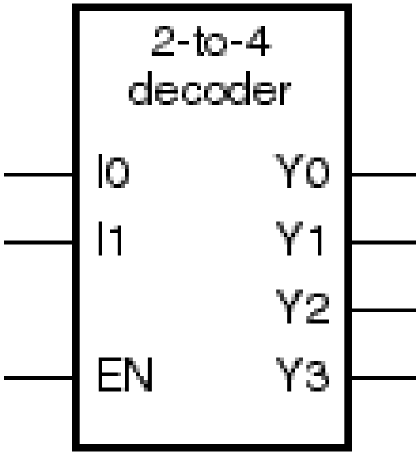


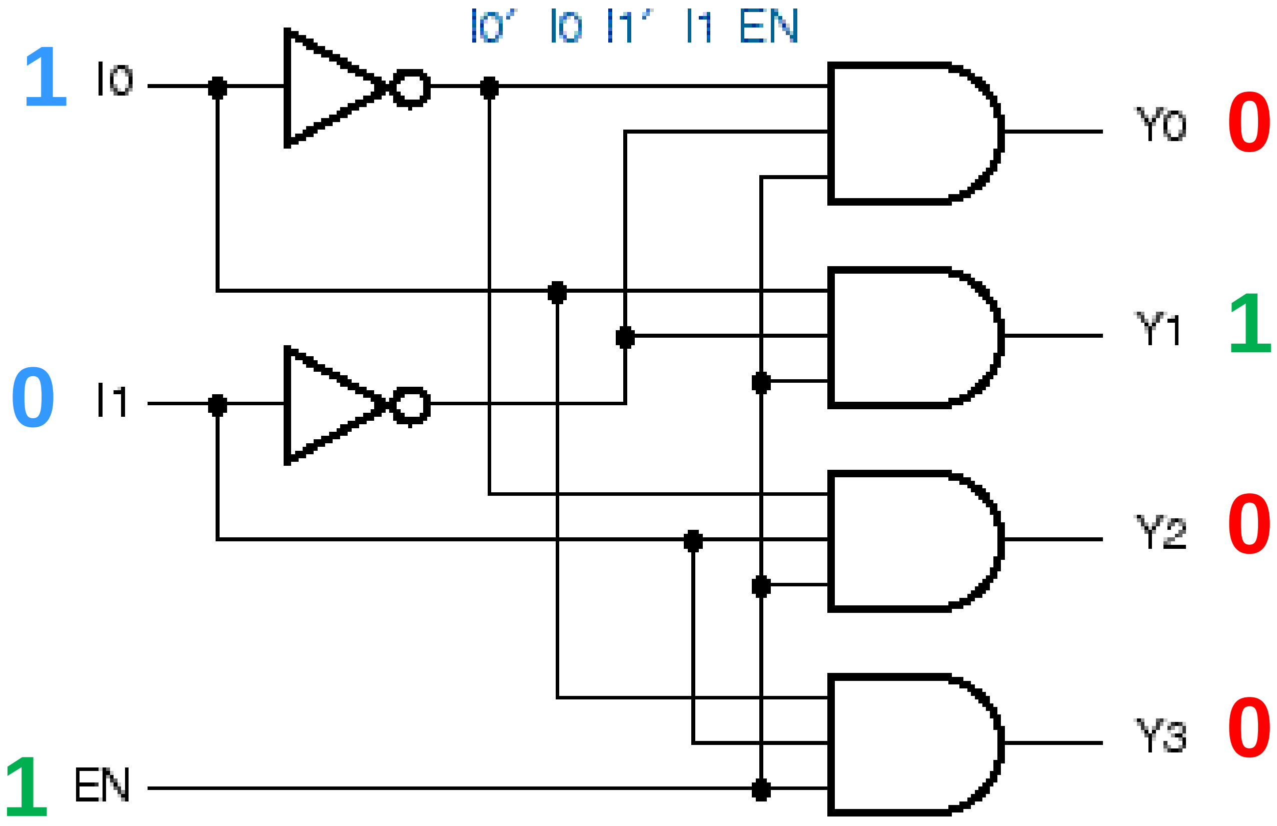


Example: 2-to-4 (2:4) Decoder

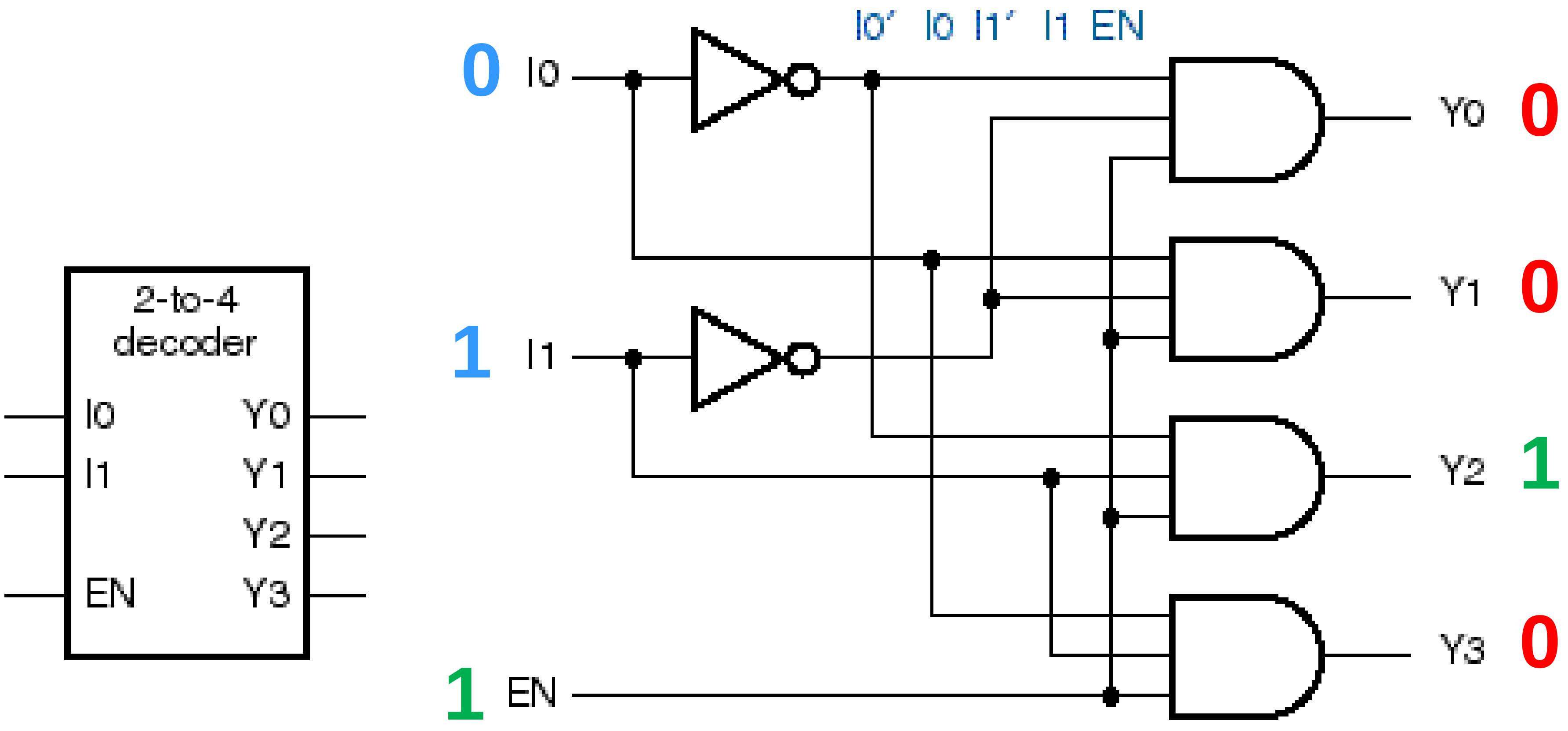


Example: 2-to-4 (2:4) Decoder

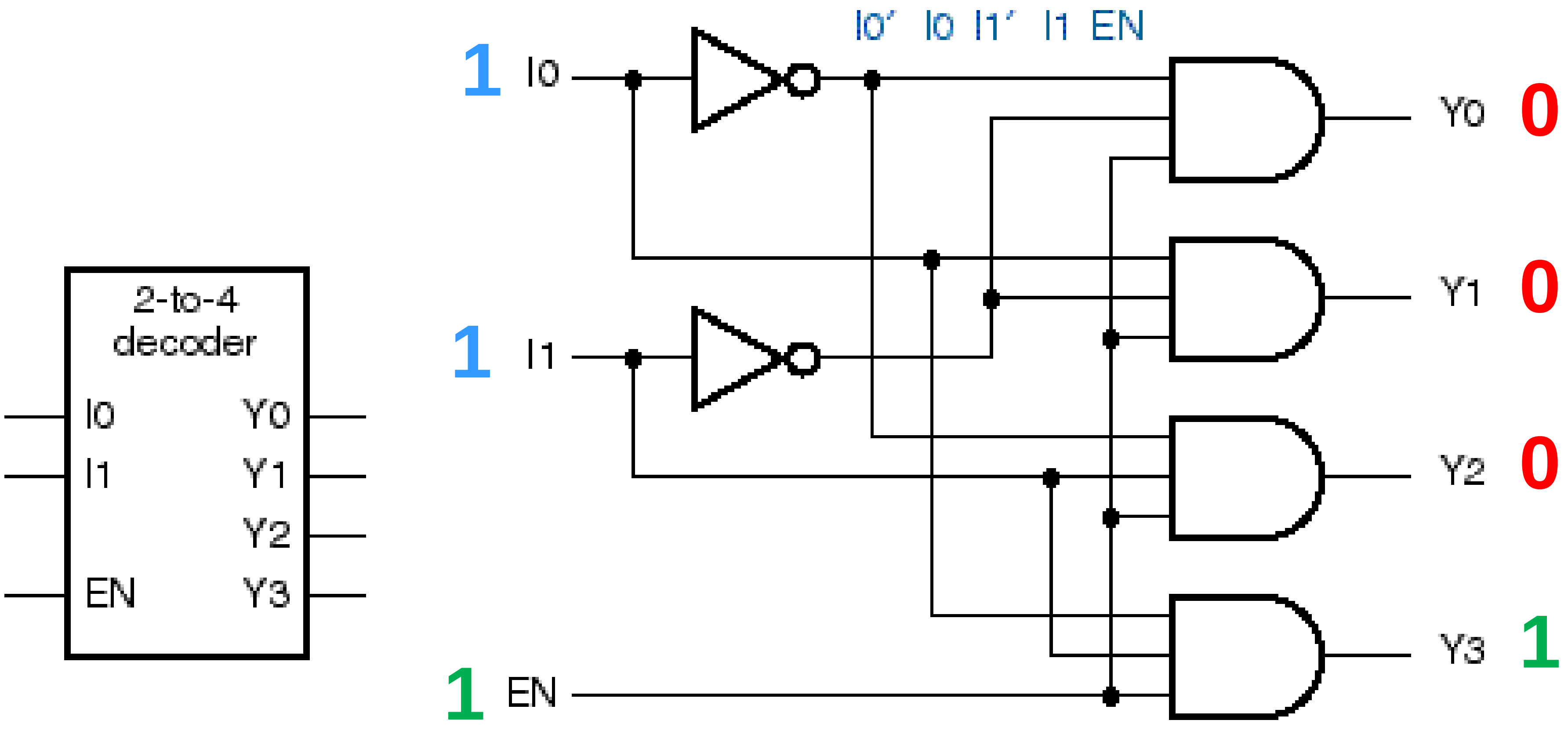




Example: 2-to-4 (2:4) Decoder



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



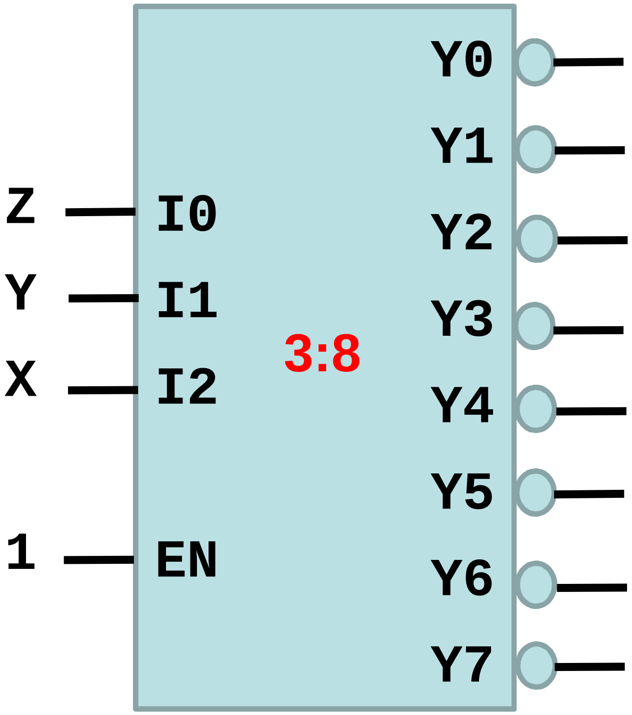
Key Observations

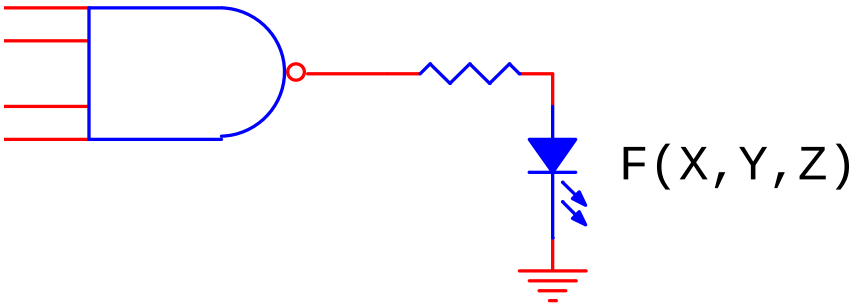
- Key Observation #1: each output of an n to 2^n binary decoder represents a minterm of an n-variable Boolean function; therefore, any arbitrary Boolean function of n-variables can be realized with an n-input binary decoder by simply "OR-ing" the needed outputs
- Key Observation #2: if the decoder outputs 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 ≤ 2^{n-1} members

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

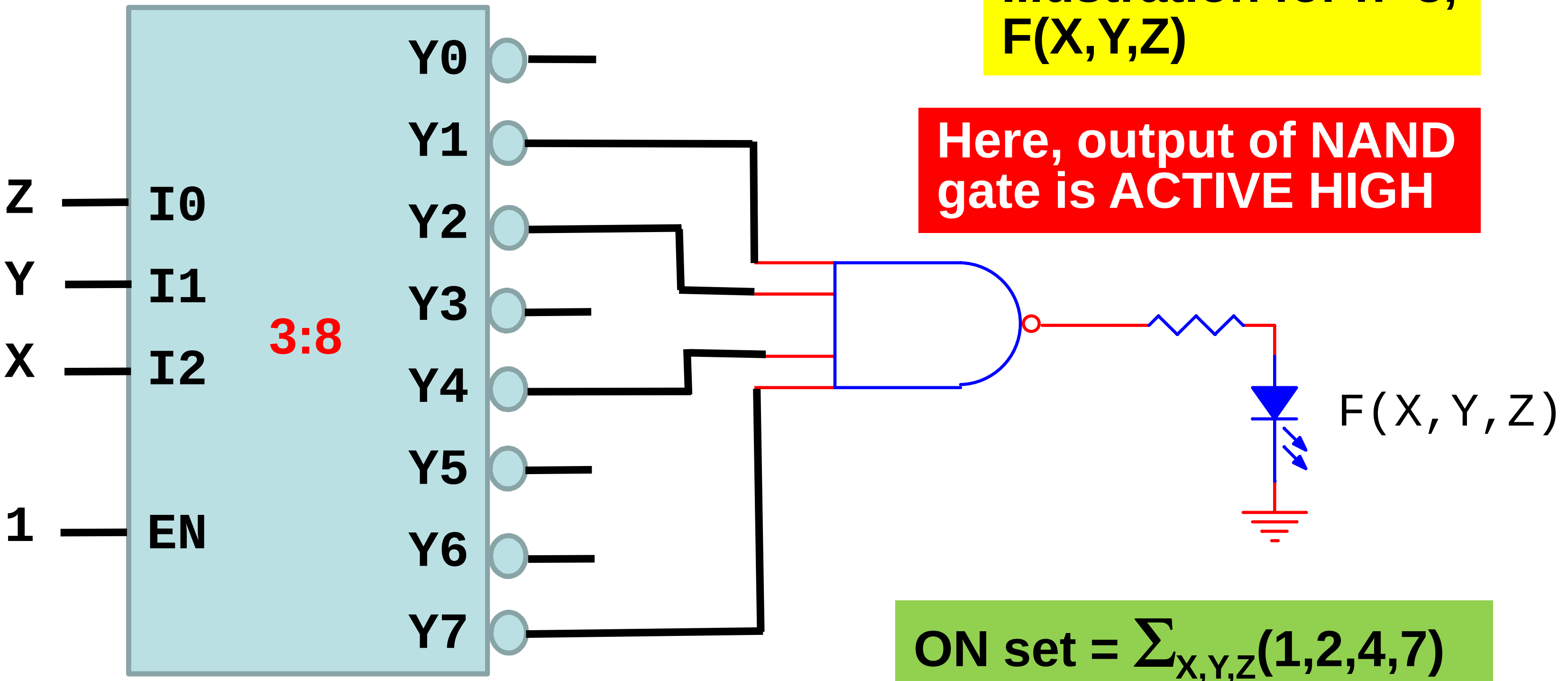




Example – Arbitrary Function Realization

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 $\leq 2^{n-1}$ members

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



Here, output of NAND gate is ACTIVE HIGH

ON set = $\Sigma_{X,Y,Z}(1,2,4,7)$

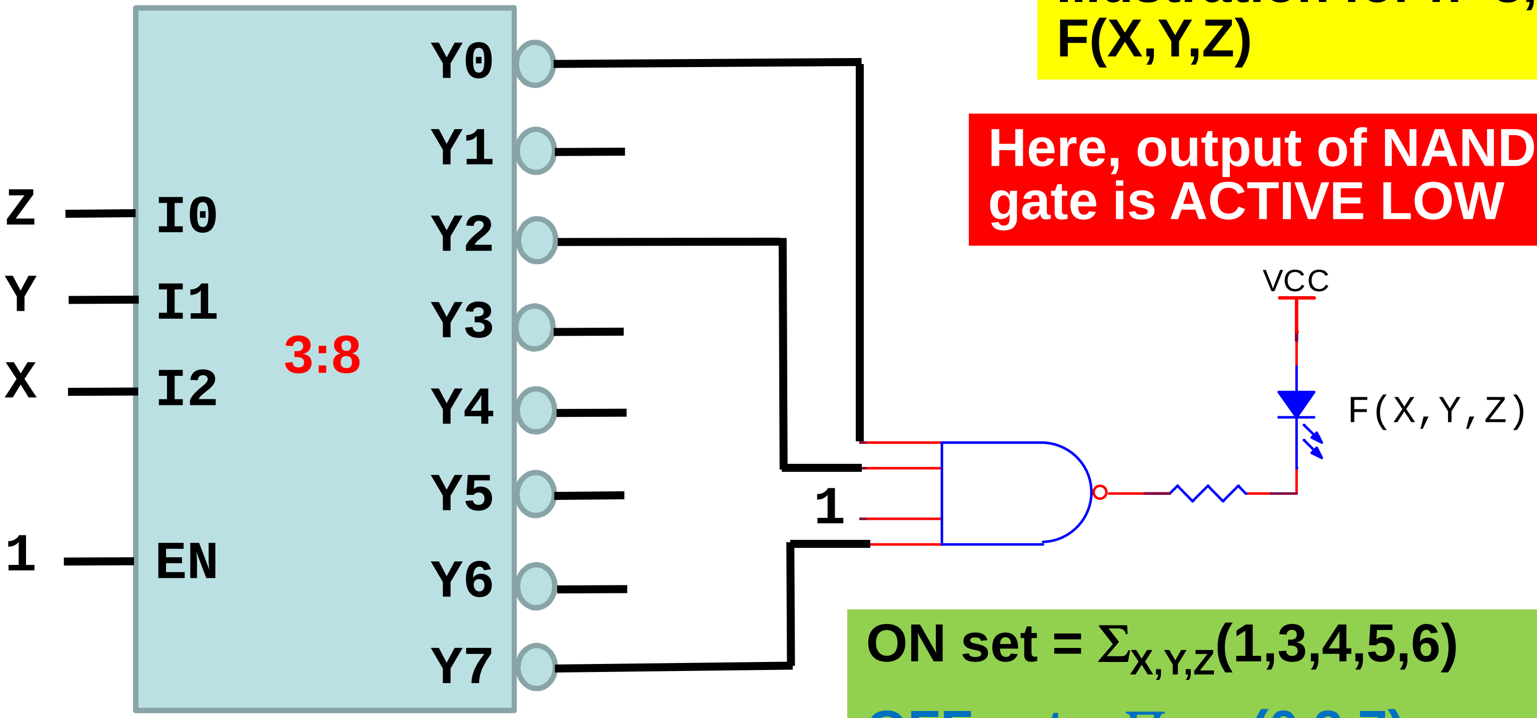
F(X,Y,Z)= $X \oplus Y \oplus Z$

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

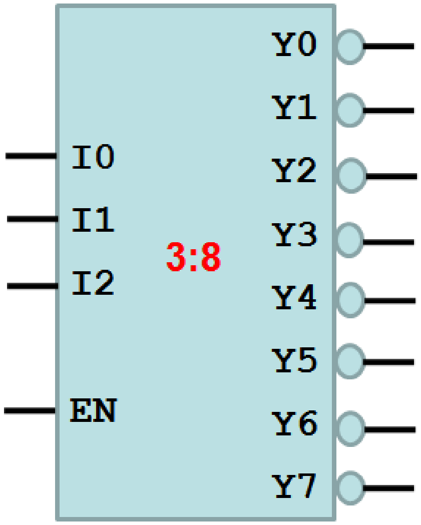


ON set = $\Sigma_{X,Y,Z}(1,3,4,5,6)$

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

F(X,Y,Z) = X′•Z + X•Y′ + X•Z′

Decoders in ABEL

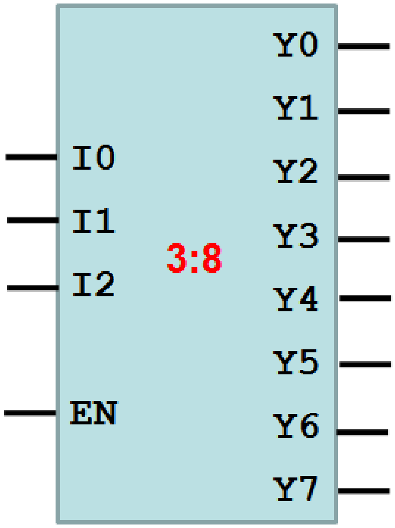


```
MODULE dec38L

TITLE '3:8 decoder implemented using 22V10'

DECLARATIONS
" Enable input pin
EN pin 2;

" Select input pins
I0..I2 pin 3, 4, 5;

" Output pins
!Y0, !Y1, !Y2, !Y3, !Y4, !Y5, !Y6, !Y7 pin 14..21 istype 'com';

EQUATIONS
Y0 = EN & !I2 & !I1 & !I0;
Y1 = EN & !I2 & !I1 &  I0;
Y2 = EN & !I2 &  I1 & !I0;
Y3 = EN & !I2 &  I1 &  I0;
Y4 = EN &  I2 & !I1 & !I0;
Y5 = EN &  I2 & !I1 &  I0;
Y6 = EN &  I2 &  I1 & !I0;
Y7 = EN &  I2 &  I1 &  I0;

END
```

Note active low pin declarations

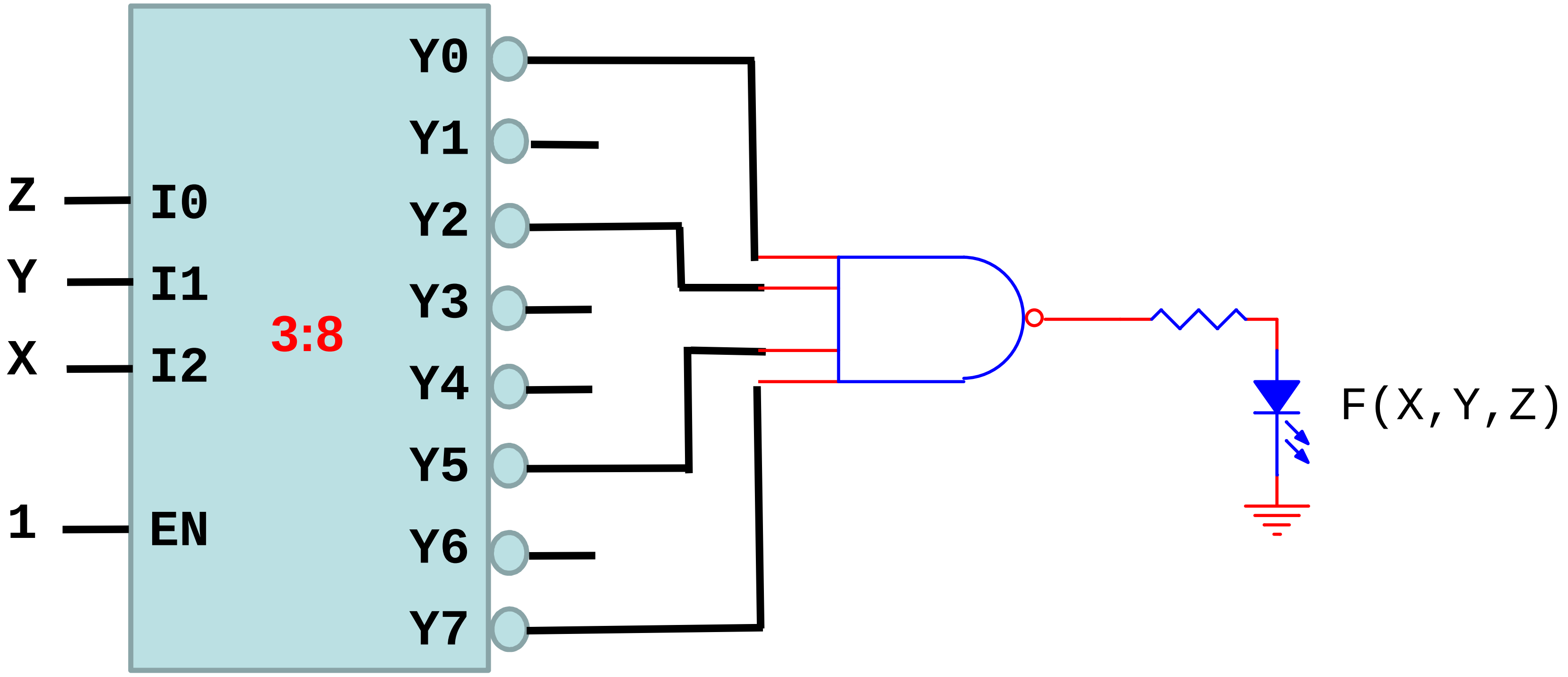
Decoders/Demultiplexers in ABEL

```
MODULE dec38H

TITLE '3:8 decoder/1:8 demultiplexer using 22V10'

DECLARATIONS
" Enable input pin
EN pin 2;

" Select input pins
I0..I2 pin 3, 4, 5;

" Output pins
Y0, Y1, Y2, Y3, Y4, Y5, Y6, Y7 pin 14..21 istype 'com';

EQUATIONS
Y0 = EN & !I2 & !I1 & !I0;
Y1 = EN & !I2 & !I1 &  I0;
Y2 = EN & !I2 &  I1 & !I0;
Y3 = EN & !I2 &  I1 &  I0;
Y4 = EN &  I2 & !I1 & !I0;
Y5 = EN &  I2 & !I1 &  I0;
Y6 = EN &  I2 &  I1 & !I0;
Y7 = EN &  I2 &  I1 &  I0;

END
```



Note active high pin declarations

Clicker Quiz



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

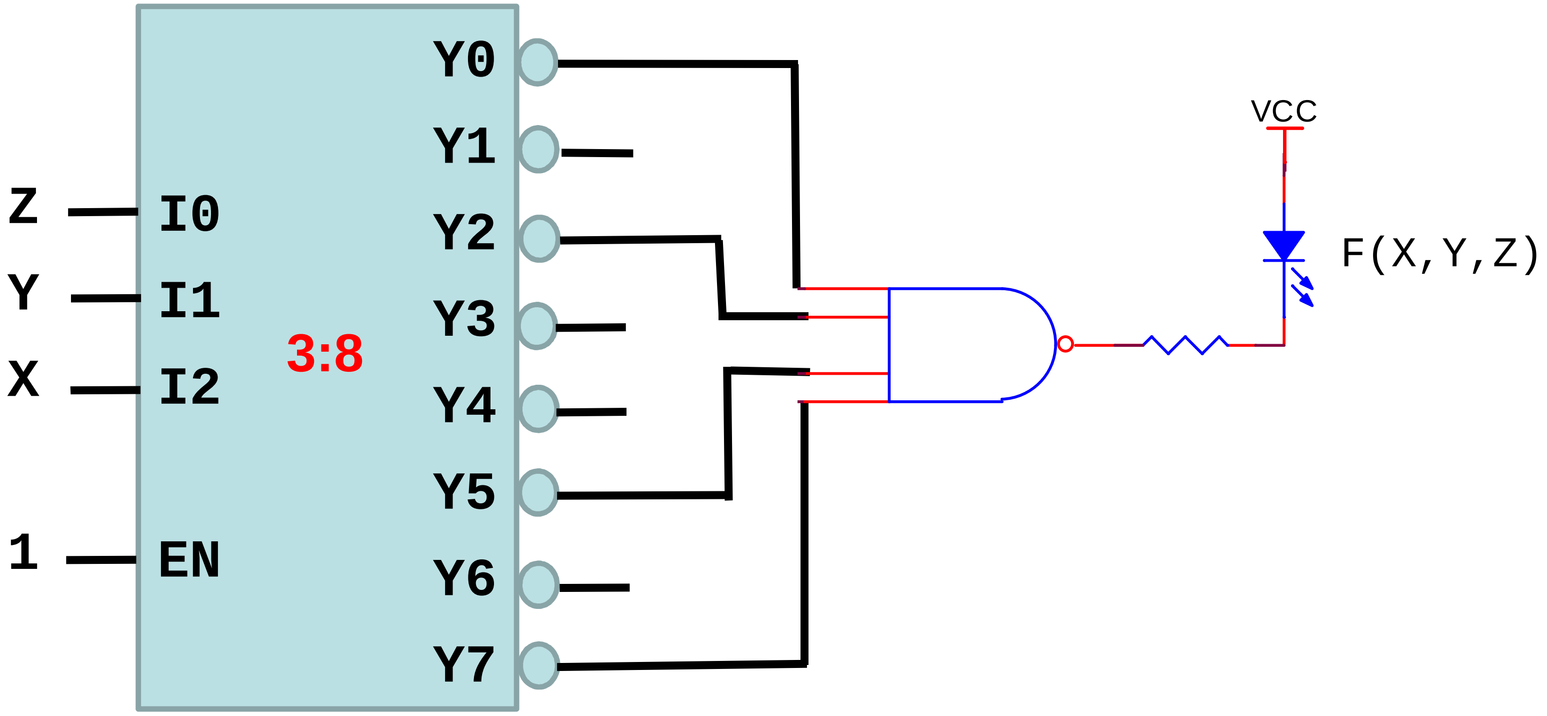
A. $\Pi_{X,Y,Z}$**(0,2,5,7)**

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

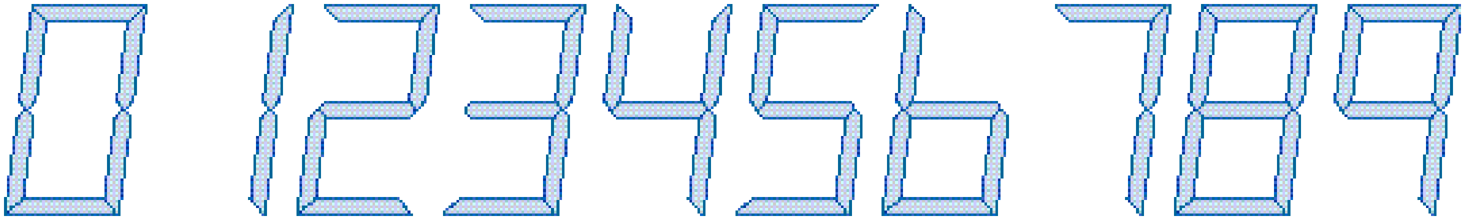
C. **$\Sigma_{X,Y,Z}(1,2,4,5)$**

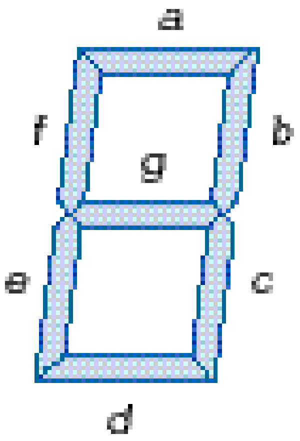
D. **$\Sigma_{X,Y,Z}(0,3,4,6)$**

E. none of the above

Special Purpose Decoders

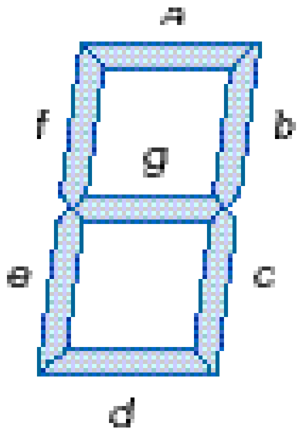
- **A seven-segment decoder has 4-bit BCD or hexadecimal data as its input code and “seven-segment code” as its output code**





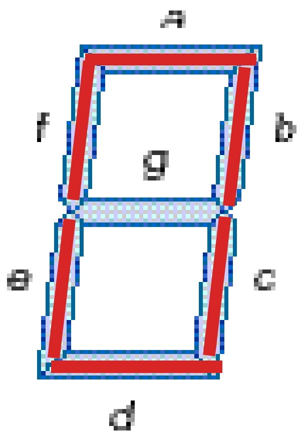
Example: Hexadecimal 7-Segment Decoder

```
MODULE hexadec

TITLE 'Hexadecimal 7-Segment Decoder for 22V10'

DECLARATIONS

i0..i3 pin 2..5;

a,b,c,d,e,f,g pin 14..20 istype 'com';

truth_table([i3,i2,i1,i0]->[a,b,c,d,e,f,g])
            [ 0, 0, 0, 0]->[1,1,1,1,1,1,0];
            [ 0, 0, 0, 1]->[0,1,1,0,0,0,0];
            [ 0, 0, 1, 0]->[1,1,0,1,1,0,1];
            [ 0, 0, 1, 1]->[1,1,1,1,0,0,1];
            [ 0, 1, 0, 0]->[0,1,1,0,0,1,1];
            [ 0, 1, 0, 1]->[1,0,1,1,0,1,1];
            [ 0, 1, 1, 0]->[1,0,1,1,1,1,1];
            [ 0, 1, 1, 1]->[1,1,1,0,0,0,0];
            [ 1, 0, 0, 0]->[1,1,1,1,1,1,1];
            [ 1, 0, 0, 1]->[1,1,1,1,0,1,1];
            [ 1, 0, 1, 0]->[1,1,1,0,1,1,1];
            [ 1, 0, 1, 1]->[0,0,1,1,1,1,1];
            [ 1, 1, 0, 0]->[1,0,0,1,1,1,0];
            [ 1, 1, 0, 1]->[0,1,1,1,1,0,1];
            [ 1, 1, 1, 0]->[1,0,0,1,1,1,1];
            [ 1, 1, 1, 1]->[1,0,0,0,1,1,1];

END
```



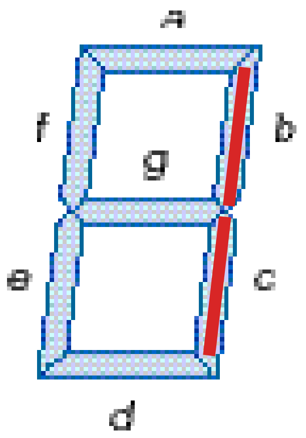
Example: Hexadecimal 7-Segment Decoder

```
MODULE hexadec

TITLE 'Hexadecimal 7-Segment Decoder for 22V10'

DECLARATIONS

i0..i3 pin 2..5;

a,b,c,d,e,f,g pin 14..20 istype 'com';

truth_table([i3,i2,i1,i0]->[a,b,c,d,e,f,g])
            [ 0, 0, 0, 0]->[1,1,1,1,1,1,0];
            [ 0, 0, 0, 1]->[0,1,1,0,0,0,0];
            [ 0, 0, 1, 0]->[1,1,0,1,1,0,1];
            [ 0, 0, 1, 1]->[1,1,1,1,0,0,1];
            [ 0, 1, 0, 0]->[0,1,1,0,0,1,1];
            [ 0, 1, 0, 1]->[1,0,1,1,0,1,1];
            [ 0, 1, 1, 0]->[1,0,1,1,1,1,1];
            [ 0, 1, 1, 1]->[1,1,1,0,0,0,0];
            [ 1, 0, 0, 0]->[1,1,1,1,1,1,1];
            [ 1, 0, 0, 1]->[1,1,1,1,0,1,1];
            [ 1, 0, 1, 0]->[1,1,1,0,1,1,1];
            [ 1, 0, 1, 1]->[0,0,1,1,1,1,1];
            [ 1, 1, 0, 0]->[1,0,0,1,1,1,0];
            [ 1, 1, 0, 1]->[0,1,1,1,1,0,1];
            [ 1, 1, 1, 0]->[1,0,0,1,1,1,1];
            [ 1, 1, 1, 1]->[1,0,0,0,1,1,1];

END
```



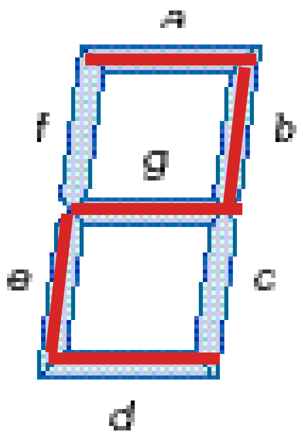
Example: Hexadecimal 7-Segment Decoder

```
MODULE hexadec

TITLE 'Hexadecimal 7-Segment Decoder for 22V10'

DECLARATIONS

i0..i3 pin 2..5;

a,b,c,d,e,f,g pin 14..20 istype 'com';

truth_table([i3,i2,i1,i0]->[a,b,c,d,e,f,g])
            [ 0, 0, 0, 0]->[1,1,1,1,1,1,0];
            [ 0, 0, 0, 1]->[0,1,1,0,0,0,0];
            [ 0, 0, 1, 0]->[1,1,0,1,1,0,1];
            [ 0, 0, 1, 1]->[1,1,1,1,0,0,1];
            [ 0, 1, 0, 0]->[0,1,1,0,0,1,1];
            [ 0, 1, 0, 1]->[1,0,1,1,0,1,1];
            [ 0, 1, 1, 0]->[1,0,1,1,1,1,1];
            [ 0, 1, 1, 1]->[1,1,1,0,0,0,0];
            [ 1, 0, 0, 0]->[1,1,1,1,1,1,1];
            [ 1, 0, 0, 1]->[1,1,1,1,0,1,1];
            [ 1, 0, 1, 0]->[1,1,1,0,1,1,1];
            [ 1, 0, 1, 1]->[0,0,1,1,1,1,1];
            [ 1, 1, 0, 0]->[1,0,0,1,1,1,0];
            [ 1, 1, 0, 1]->[0,1,1,1,1,0,1];
            [ 1, 1, 1, 0]->[1,0,0,1,1,1,1];
            [ 1, 1, 1, 1]->[1,0,0,0,1,1,1];

END
```



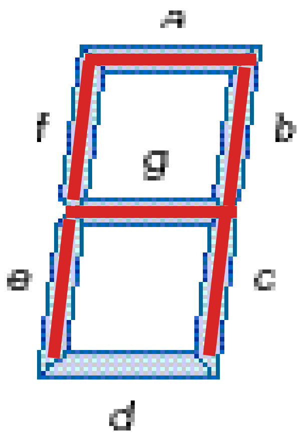
Example: Hexadecimal 7-Segment Decoder

```
MODULE hexadec

TITLE 'Hexadecimal 7-Segment Decoder for 22V10'

DECLARATIONS

i0..i3 pin 2..5;

a,b,c,d,e,f,g pin 14..20 istype 'com';

truth_table([i3,i2,i1,i0]->[a,b,c,d,e,f,g])
            [ 0, 0, 0, 0]->[1,1,1,1,1,1,0];
            [ 0, 0, 0, 1]->[0,1,1,0,0,0,0];
            [ 0, 0, 1, 0]->[1,1,0,1,1,0,1];
            [ 0, 0, 1, 1]->[1,1,1,1,0,0,1];
            [ 0, 1, 0, 0]->[0,1,1,0,0,1,1];
            [ 0, 1, 0, 1]->[1,0,1,1,0,1,1];
            [ 0, 1, 1, 0]->[1,0,1,1,1,1,1];
            [ 0, 1, 1, 1]->[1,1,1,0,0,0,0];
            [ 1, 0, 0, 0]->[1,1,1,1,1,1,1];
            [ 1, 0, 0, 1]->[1,1,1,1,0,1,1];
            [ 1, 0, 1, 0]->[1,1,1,0,1,1,1];
            [ 1, 0, 1, 1]->[0,0,1,1,1,1,1];
            [ 1, 1, 0, 0]->[1,0,0,1,1,1,0];
            [ 1, 1, 0, 1]->[0,1,1,1,1,0,1];
            [ 1, 1, 1, 0]->[1,0,0,1,1,1,1];
            [ 1, 1, 1, 1]->[1,0,0,0,1,1,1];

END
```



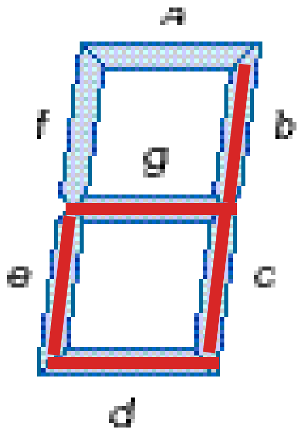
Example: Hexadecimal 7-Segment Decoder

```
MODULE hexadec

TITLE 'Hexadecimal 7-Segment Decoder for 22V10'

DECLARATIONS

i0..i3 pin 2..5;

a,b,c,d,e,f,g pin 14..20 istype 'com';

truth_table([i3,i2,i1,i0]->[a,b,c,d,e,f,g])
            [ 0, 0, 0, 0]->[1,1,1,1,1,1,0];
            [ 0, 0, 0, 1]->[0,1,1,0,0,0,0];
            [ 0, 0, 1, 0]->[1,1,0,1,1,0,1];
            [ 0, 0, 1, 1]->[1,1,1,1,0,0,1];
            [ 0, 1, 0, 0]->[0,1,1,0,0,1,1];
            [ 0, 1, 0, 1]->[1,0,1,1,0,1,1];
            [ 0, 1, 1, 0]->[1,0,1,1,1,1,1];
            [ 0, 1, 1, 1]->[1,1,1,0,0,0,0];
            [ 1, 0, 0, 0]->[1,1,1,1,1,1,1];
            [ 1, 0, 0, 1]->[1,1,1,1,0,1,1];
            [ 1, 0, 1, 0]->[1,1,1,0,1,1,1];
            [ 1, 0, 1, 1]->[0,0,1,1,1,1,1];
            [ 1, 1, 0, 0]->[1,0,0,1,1,1,0];
            [ 1, 1, 0, 1]->[0,1,1,1,1,0,1];
            [ 1, 1, 1, 0]->[1,0,0,1,1,1,1];
            [ 1, 1, 1, 1]->[1,0,0,0,1,1,1];

END
```



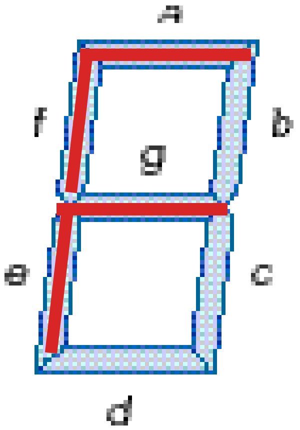
Example: Hexadecimal 7-Segment Decoder

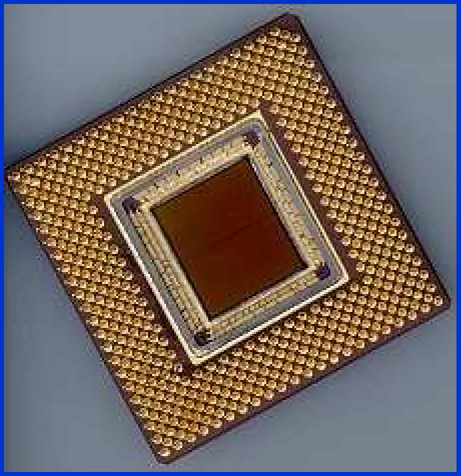
```
MODULE hexadec

TITLE 'Hexadecimal 7-Segment Decoder for 22V10'

DECLARATIONS

i0..i3 pin 2..5;

a,b,c,d,e,f,g pin 14..20 istype 'com';

truth_table([i3,i2,i1,i0]->[a,b,c,d,e,f,g])
            [ 0, 0, 0, 0]->[1,1,1,1,1,1,0];
            [ 0, 0, 0, 1]->[0,1,1,0,0,0,0];
            [ 0, 0, 1, 0]->[1,1,0,1,1,0,1];
            [ 0, 0, 1, 1]->[1,1,1,1,0,0,1];
            [ 0, 1, 0, 0]->[0,1,1,0,0,1,1];
            [ 0, 1, 0, 1]->[1,0,1,1,0,1,1];
            [ 0, 1, 1, 0]->[1,0,1,1,1,1,1];
            [ 0, 1, 1, 1]->[1,1,1,0,0,0,0];
            [ 1, 0, 0, 0]->[1,1,1,1,1,1,1];
            [ 1, 0, 0, 1]->[1,1,1,1,0,1,1];
            [ 1, 0, 1, 0]->[1,1,1,0,1,1,1];
            [ 1, 0, 1, 1]->[0,0,1,1,1,1,1];
            [ 1, 1, 0, 0]->[1,0,0,1,1,1,0];
            [ 1, 1, 0, 1]->[0,1,1,1,1,0,1];
            [ 1, 1, 1, 0]->[1,0,0,1,1,1,1];
            [ 1, 1, 1, 1]->[1,0,0,0,1,1,1];

END
```



Example: Hexadecimal 7-Segment Decoder

```
MODULE hexadec

TITLE 'Hexadecimal 7-Segment Decoder for 22V10'

DECLARATIONS

i0..i3 pin 2..5;

a,b,c,d,e,f,g pin 14..20 istype 'com';

truth_table([i3,i2,i1,i0]->[a,b,c,d,e,f,g])
            [ 0, 0, 0, 0]->[1,1,1,1,1,1,0];
            [ 0, 0, 0, 1]->[0,1,1,0,0,0,0];
            [ 0, 0, 1, 0]->[1,1,0,1,1,0,1];
            [ 0, 0, 1, 1]->[1,1,1,1,0,0,1];
            [ 0, 1, 0, 0]->[0,1,1,0,0,1,1];
            [ 0, 1, 0, 1]->[1,0,1,1,0,1,1];
            [ 0, 1, 1, 0]->[1,0,1,1,1,1,1];
            [ 0, 1, 1, 1]->[1,1,1,0,0,0,0];
            [ 1, 0, 0, 0]->[1,1,1,1,1,1,1];
            [ 1, 0, 0, 1]->[1,1,1,1,0,1,1];
            [ 1, 0, 1, 0]->[1,1,1,0,1,1,1];
            [ 1, 0, 1, 1]->[0,0,1,1,1,1,1];
            [ 1, 1, 0, 0]->[1,0,0,1,1,1,0];
            [ 1, 1, 0, 1]->[0,1,1,1,1,0,1];
            [ 1, 1, 1, 0]->[1,0,0,1,1,1,1];
            [ 1, 1, 1, 1]->[1,0,0,0,1,1,1];

END
```





Introduction to Digital System Design

Module 2-H
Combinational Building Blocks: Encoders and Tri-State Outputs

Reading Assignment:

DDPP 4th Ed., pp. 408-415

Learning Objectives:

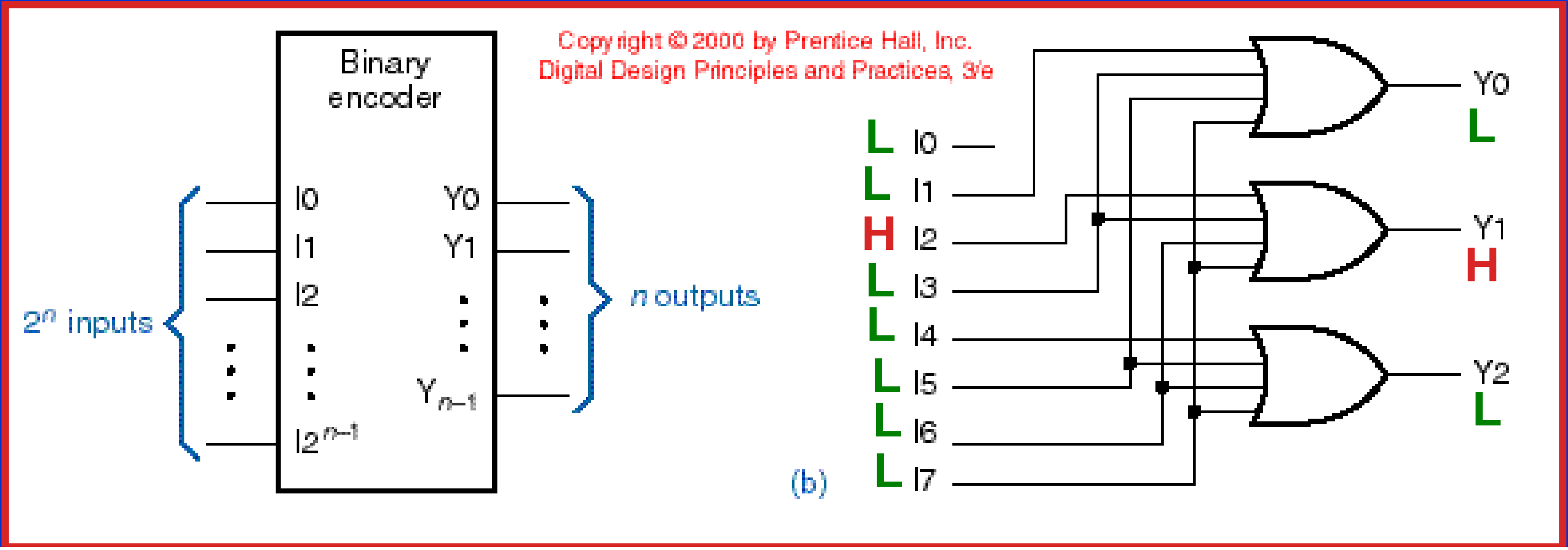
- **Define the function of an encoder and describe how it can be used as a combinational building block**
- **Discuss why the inputs of an encoder typically need to be prioritized**

Outline

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

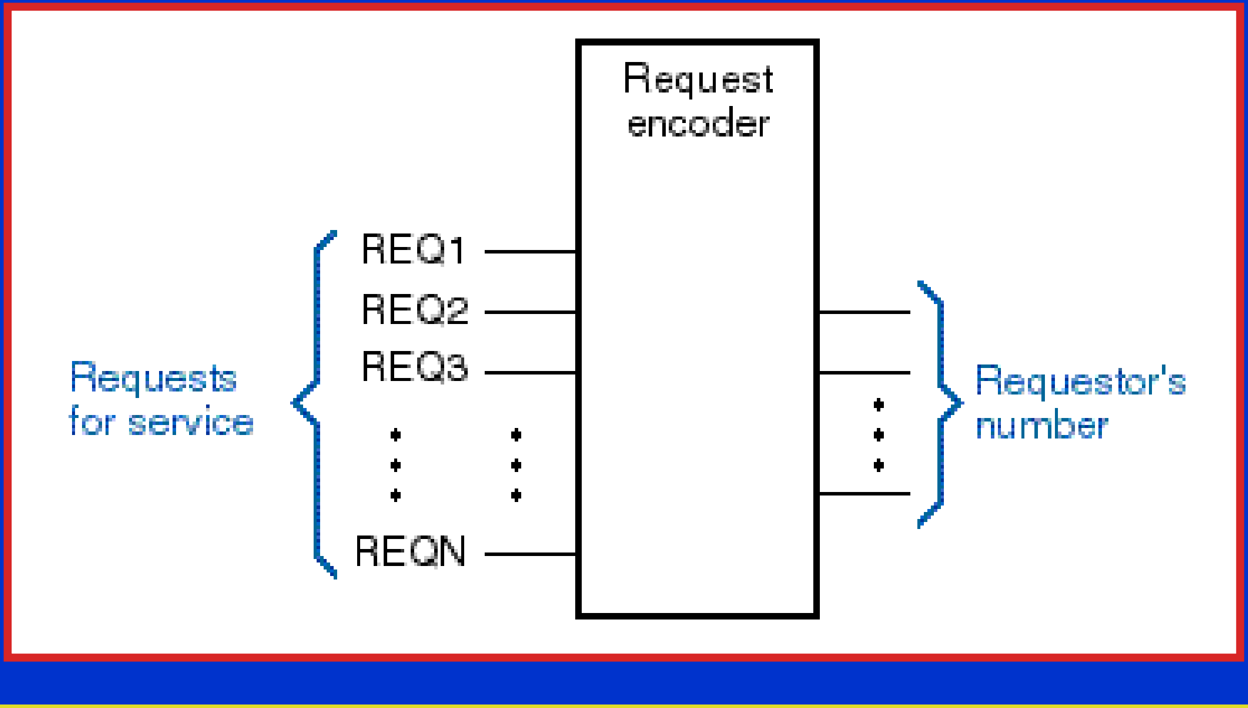
Overview

- 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



<u>Problem</u>: 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 ABEL is to use a ***truth table***
- 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

```
MODULE  pri_enc

TITLE  '8-to-3 Priority Encoder Using a GAL22V10'

DECLARATIONS
" Input pins
I0..I7  pin;  "Input 0 - lowest priority, Input 7 - highest

" Output pins
E0..E2   pin istype 'com';  "E2 E1 E0 - encoded output

GS pin istype 'com'; "strobe output (asserted if any input asserted)

" Short-hand for don't care
X = .X.;


TRUTH_TABLE ([I7,I6,I5,I4,I3,I2,I1,I0]->[E2,E1,E0,GS])
             [ 0, 0, 0, 0, 0, 0, 0, 0]->[ 0, 0, 0, 0]; " none active
             [ 0, 0, 0, 0, 0, 0, 0, 1]->[ 0, 0, 0, 1]; " input 0 wins
             [ 0, 0, 0, 0, 0, 0, 1, X]->[ 0, 0, 1, 1]; " input 1 wins
             [ 0, 0, 0, 0, 0, 1, X, X]->[ 0, 1, 0, 1]; " input 2 wins
             [ 0, 0, 0, 0, 1, X, X, X]->[ 0, 1, 1, 1]; " input 3 wins
             [ 0, 0, 0, 1, X, X, X, X]->[ 1, 0, 0, 1]; " input 4 wins
             [ 0, 0, 1, X, X, X, X, X]->[ 1, 0, 1, 1]; " input 5 wins
             [ 0, 1, X, X, X, X, X, X]->[ 1, 1, 0, 1]; " input 6 wins
             [ 1, X, X, X, X, X, X, X]->[ 1, 1, 1, 1]; " input 7 wins
END
```

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

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

Positive-Polarity (SoP) Equations:

E0 = (!I6 & !I4 & !I2 & I1 # !I6 & !I4 & I3 # !I6 & I5 # I7);

E1 = (!I5 & !I4 & I2 # !I5 & !I4 & I3 # I6 # I7);

E2 = (I4 # I5 # I6 # I7);

GS = (I1 # I0 # I2 # I3 # I4 # I5 # I6 # I7);

Reverse-Polarity (!SoP) Equations:

!E0 = (!I7 & !I5 & !I3 & !I1 # !I7 & !I5 & !I3 & I2 # !I7 & !I5 & I4 # !I7 & I6);

!E1 = (!I7 & !I6 & !I3 & !I2 # !I7 & !I6 & I4 # !I7 & !I6 & I5);

!E2 = (!I7 & !I6 & !I5 & !I4);

!GS = (!I7 & !I6 & !I5 & !I4 & !I3 & !I2 & !I1 & !I0);
```

Tri-State Outputs

- In ABEL, an ***attribute suffix*** **“.OE”** is attached to a signal name on the left-hand side of an equation to indicate that the equation applies to the ***output enable*** for that signal
- Example: Create an ABEL file 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.
- ***The tri-state buffer will enable the encoder to connect directly to a microprocessor data bus, and only “talk” on the bus when enabled.***

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

```
MODULE  prienc42
TITLE  '4-to-2 Priority Encoder with Tri-State Enable'
DECLARATIONS
" Input pins
I0..I3  pin;  " Input 0 - lowest priority, Input 3 - highest
" Output pins
E0..E1   pin istype 'com';  " E1 E0 - encoded output
GS pin istype 'com';  " strobe output (asserted if any input asserted)
EN pin;  " tri-state enable control input
" Short-hand for don't care
X = .X.;
TRUTH_TABLE ([I3,I2,I1,I0]->[E1,E0,GS])
             [ 0, 0, 0, 0]->[ 0, 0, 0];  " no inputs active
             [ 0, 0, 0, 1]->[ 0, 0, 1];  " input 0 wins
             [ 0, 0, 1, X]->[ 0, 1, 1];  " input 1 wins
             [ 0, 1, X, X]->[ 1, 0, 1];  " input 2 wins
             [ 1, X, X, X]->[ 1, 1, 1];  " input 3 wins

EQUATIONS
[E0..E1].OE = EN;
END
```

set

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 ABEL file described previously (include tri-state output capability)

```
MODULE  bcd_enc
TITLE  '10-to-4 BCD Priority Keypad Encoder Using a GAL22V10'
DECLARATIONS
" Input pins
K0..K9   pin;  " key inputs (0 - lowest priority, 9 - highest)
EN pin; " tri-state output enable
" Output pins
E0..E3   pin istype 'com';  " E3 E2 E1 E0 - encoded BCD output
KS  pin  istype 'com';  " key strobe (high when any key pressed)
X = .X.;

TRUTH_TABLE ([K9,K8,K7,K6,K5,K4,K3,K2,K1,K0]->[E3,E2,E1,E0,KS])
             [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]->[ 0, 0, 0, 0, 0];
             [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]->[ 0, 0, 0, 0, 1];
             [ 0, 0, 0, 0, 0, 0, 0, 0, 1, X]->[ 0, 0, 0, 1, 1];
             [ 0, 0, 0, 0, 0, 0, 0, 1, X, X]->[ 0, 0, 1, 0, 1];
             [ 0, 0, 0, 0, 0, 0, 1, X, X, X]->[ 0, 0, 1, 1, 1];
             [ 0, 0, 0, 0, 0, 1, X, X, X, X]->[ 0, 1, 0, 0, 1];
             [ 0, 0, 0, 0, 1, X, X, X, X, X]->[ 0, 1, 0, 1, 1];
             [ 0, 0, 0, 1, X, X, X, X, X, X]->[ 0, 1, 1, 0, 1];
             [ 0, 0, 1, X, X, X, X, X, X, X]->[ 0, 1, 1, 1, 1];
             [ 0, 1, X, X, X, X, X, X, X, X]->[ 1, 0, 0, 0, 1];
             [ 1, X, X, X, X, X, X, X, X, X]->[ 1, 0, 0, 1, 1];
EQUATIONS
[E3..E0].OE = EN;
END
```

Clicker Quiz

```
MODULE  diff_pri

TITLE  'Different Priority Encoder'

DECLARATIONS
A,B,C,D  pin;
E0..E1   pin istype 'com';
GS pin istype 'com';
X = .X.;

TRUTH_TABLE ([ A, B, C, D]->[E1,E0,GS])
             [ 0, 0, 0, 0]->[ 0, 0, 0];
             [ 0, 0, 0, 1]->[ 1, 1, 1];
             [ 0, 0, 1, X]->[ 1, 0, 1];
             [ 0, 1, X, X]->[ 0, 1, 1];
             [ 1, X, X, X]->[ 0, 0, 1];

END
```

1. The highest priority input is:

A. A

B. B

C. C

D. D

E. none of the above

2. The lowest priority input is:

A. A

B. B

C. C

D. D

E. none of the above

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

A. E1=0, E0=0, 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

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

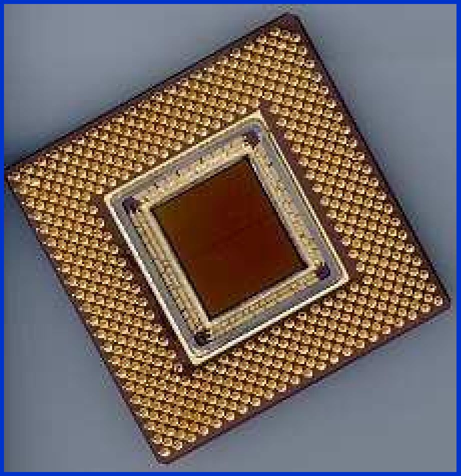
A. E1=0, E0=0, GS=1

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



Introduction to Digital System Design

Module 2-I Combinational Building Blocks: Multiplexers

Reading Assignment:

DDPP 4th Ed., pp. 432-443

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 ABEL

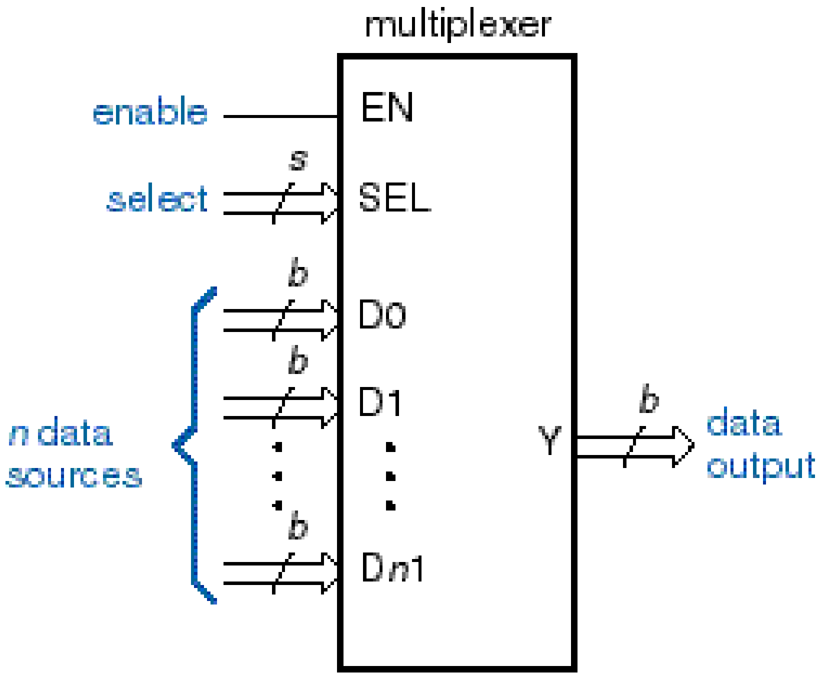
Overview

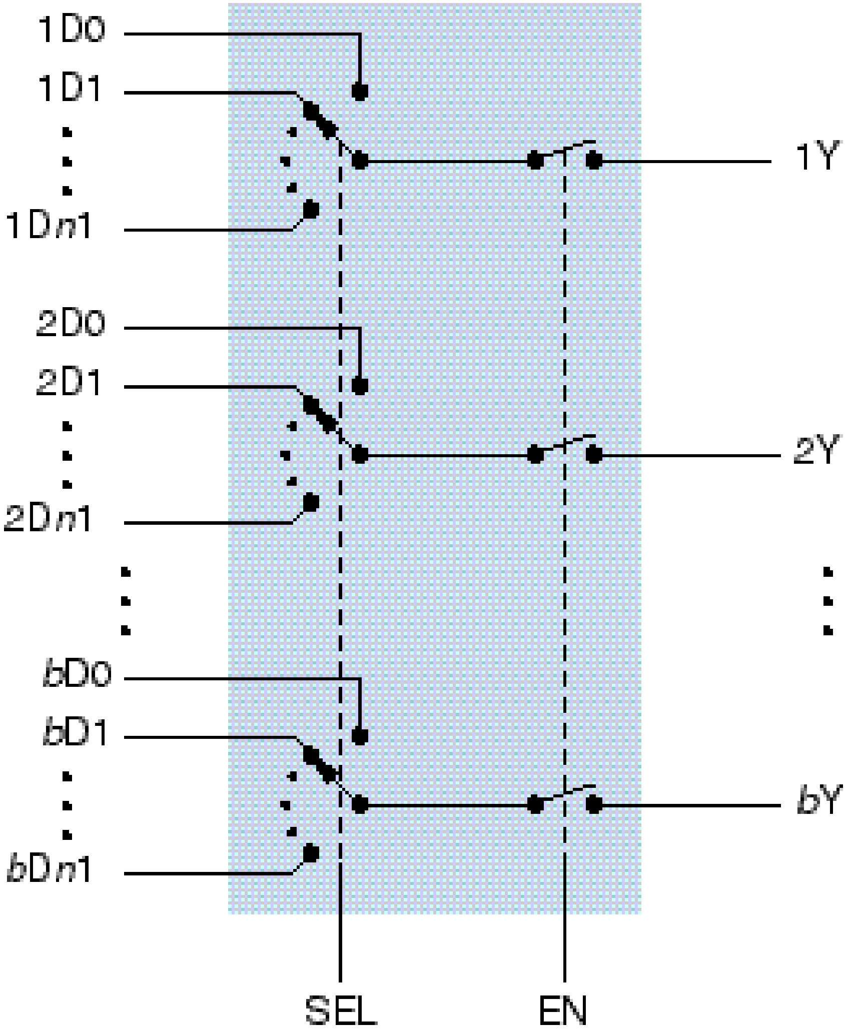
- Definition: A ***multiplexer*** is a digital switch that uses ***s*** select lines to determine which of $n = 2^s$ inputs is connected to its output
- It is often called a ***mux*** for short
- Each of the input paths may be ***b*** bits wide
- An overall enable signal **(EN)** is usually provided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

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

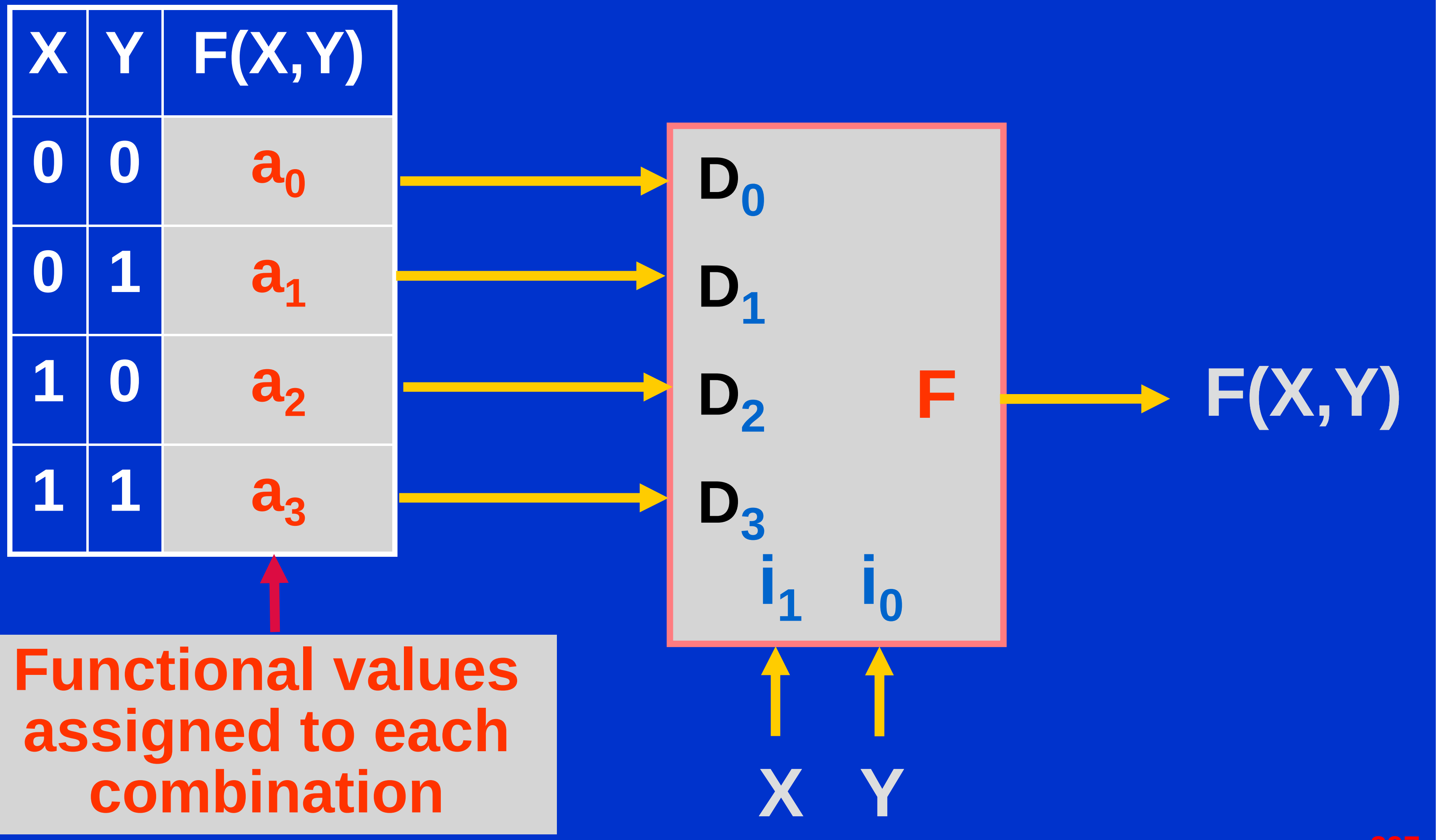
General Multiplexer Structure

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





Multiplexer Truth Table Analogy



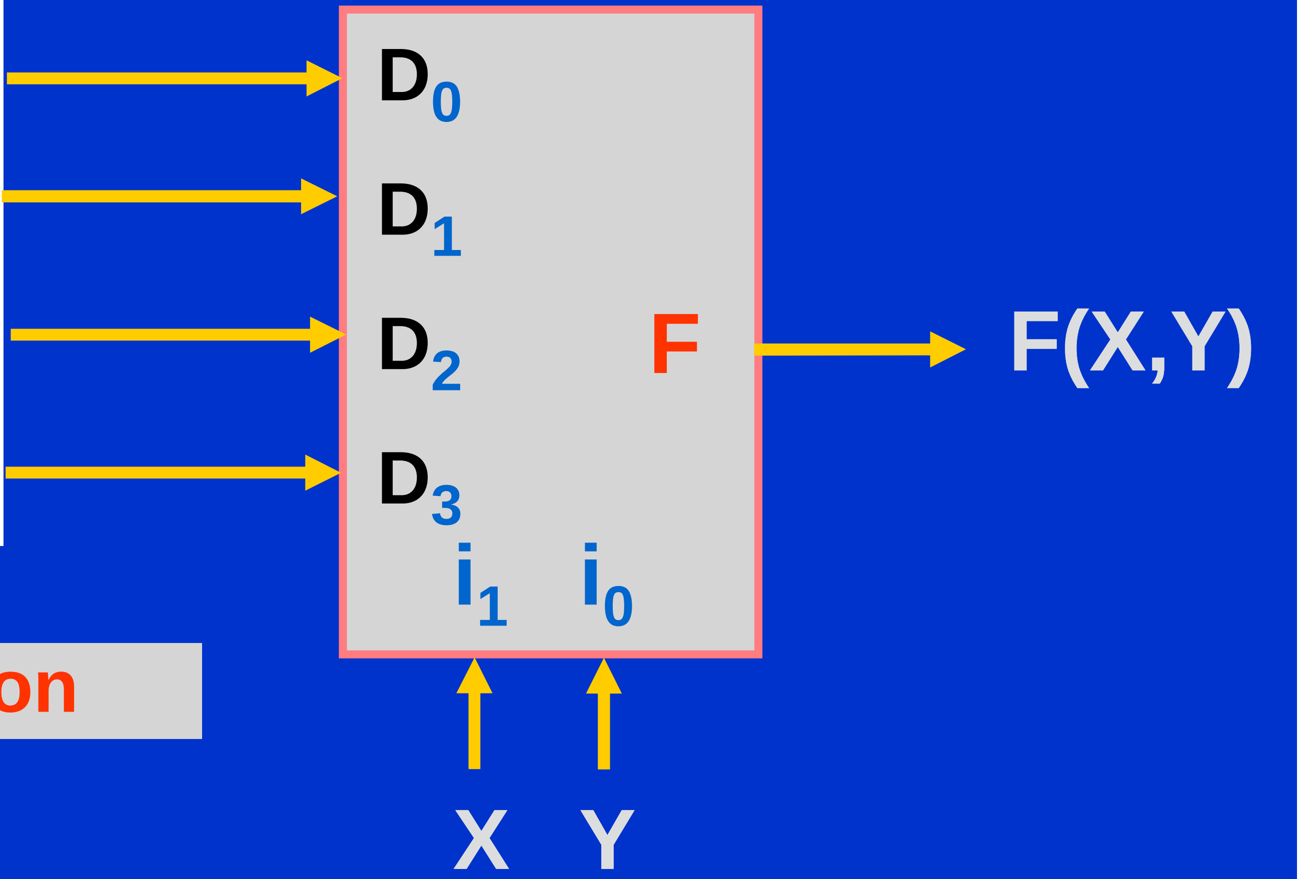
X	Y	F(X,Y)
0	0	a_0
0	1	a_1
1	0	a_2
1	1	a_3

Functional values assigned to each combination

Multiplexer Truth Table Analogy

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

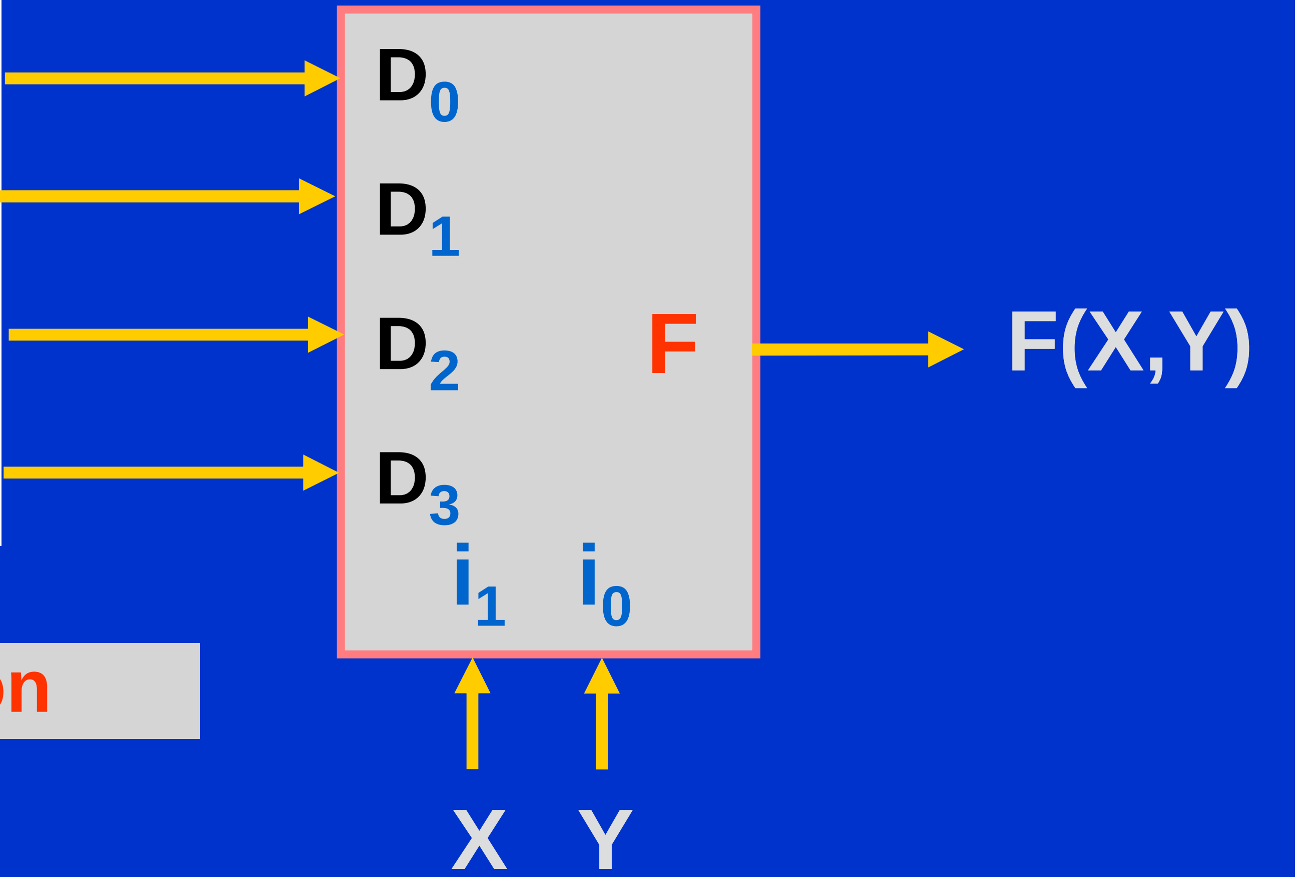
AND function



Multiplexer Truth Table Analogy

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

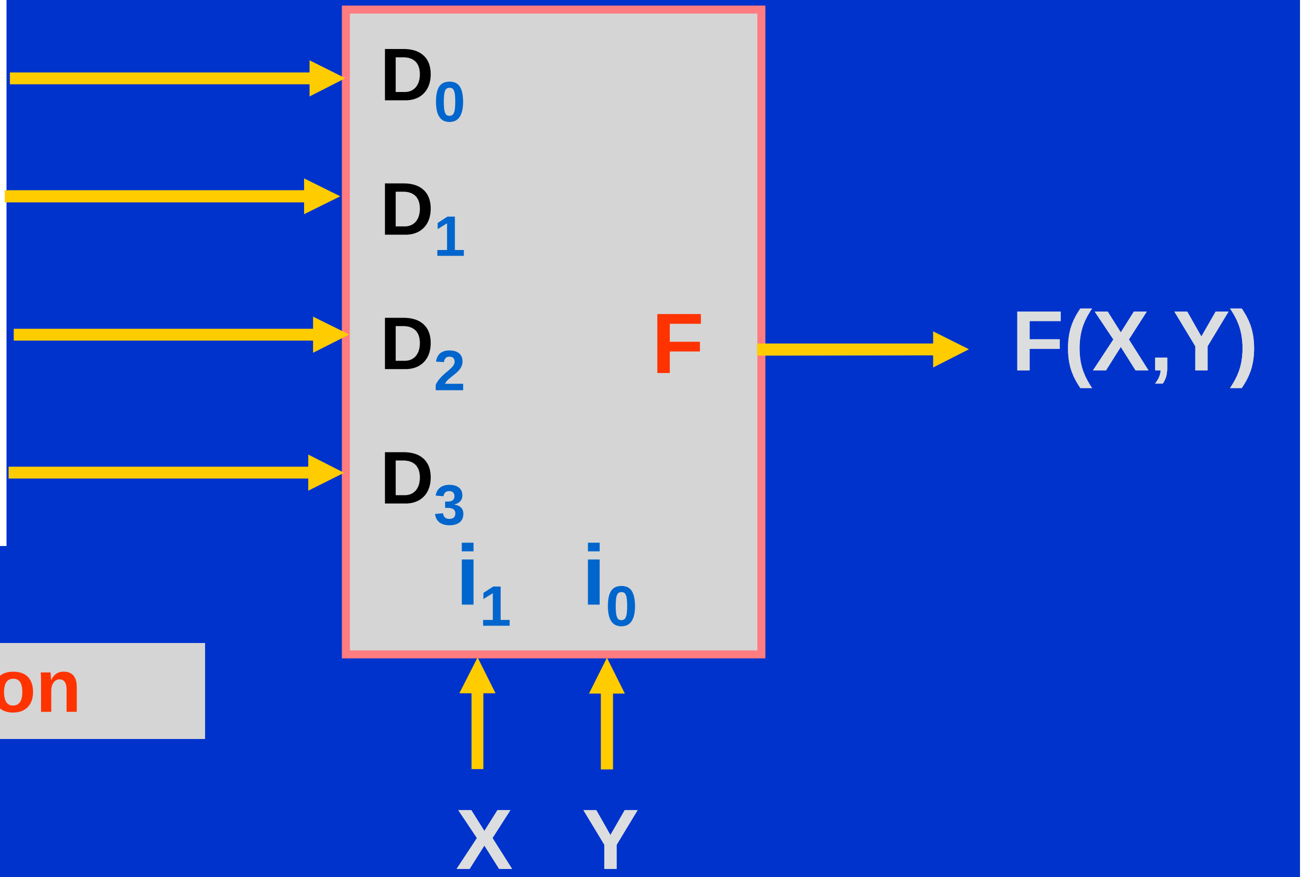
OR function



Multiplexer Truth Table Analogy

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

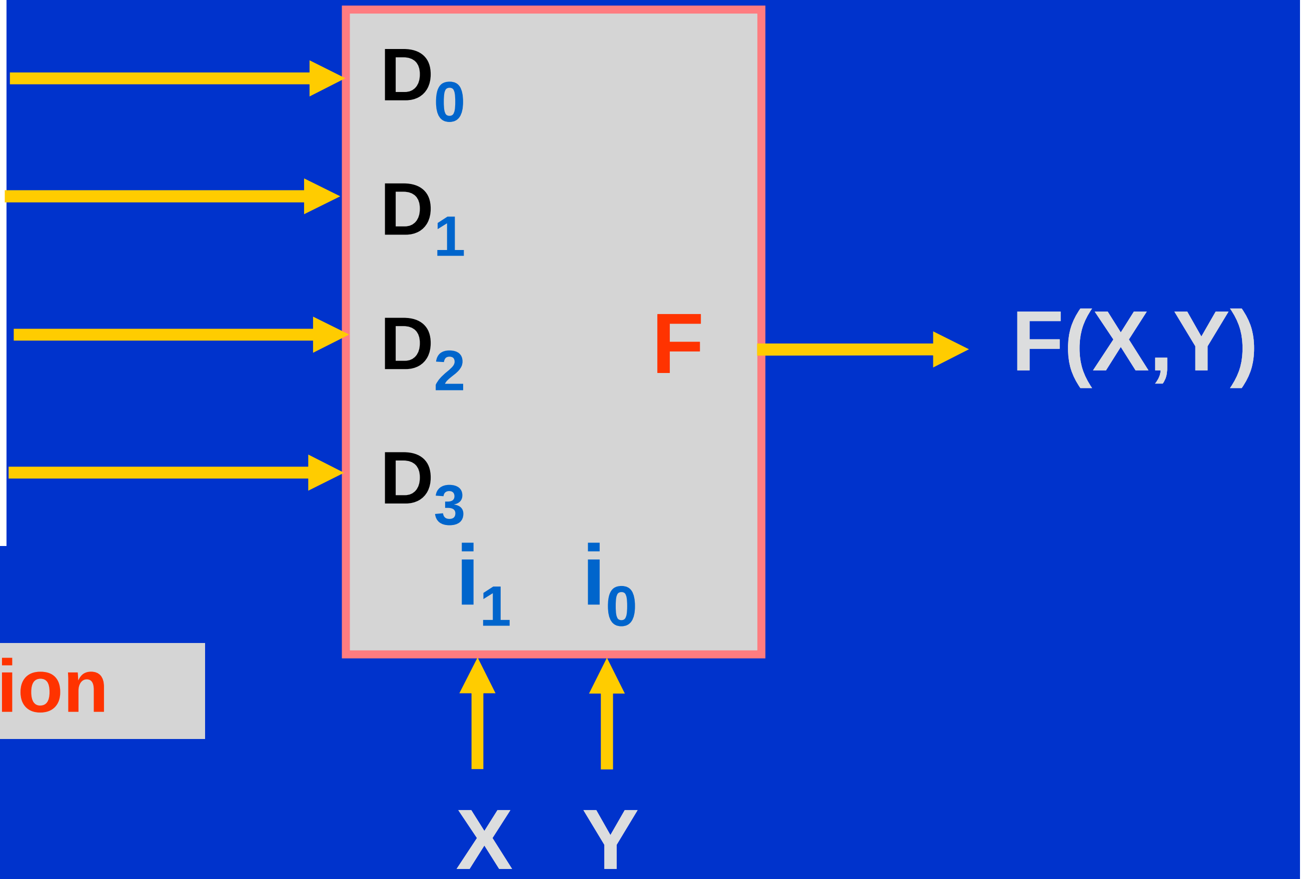
XOR function



Multiplexer Truth Table Analogy

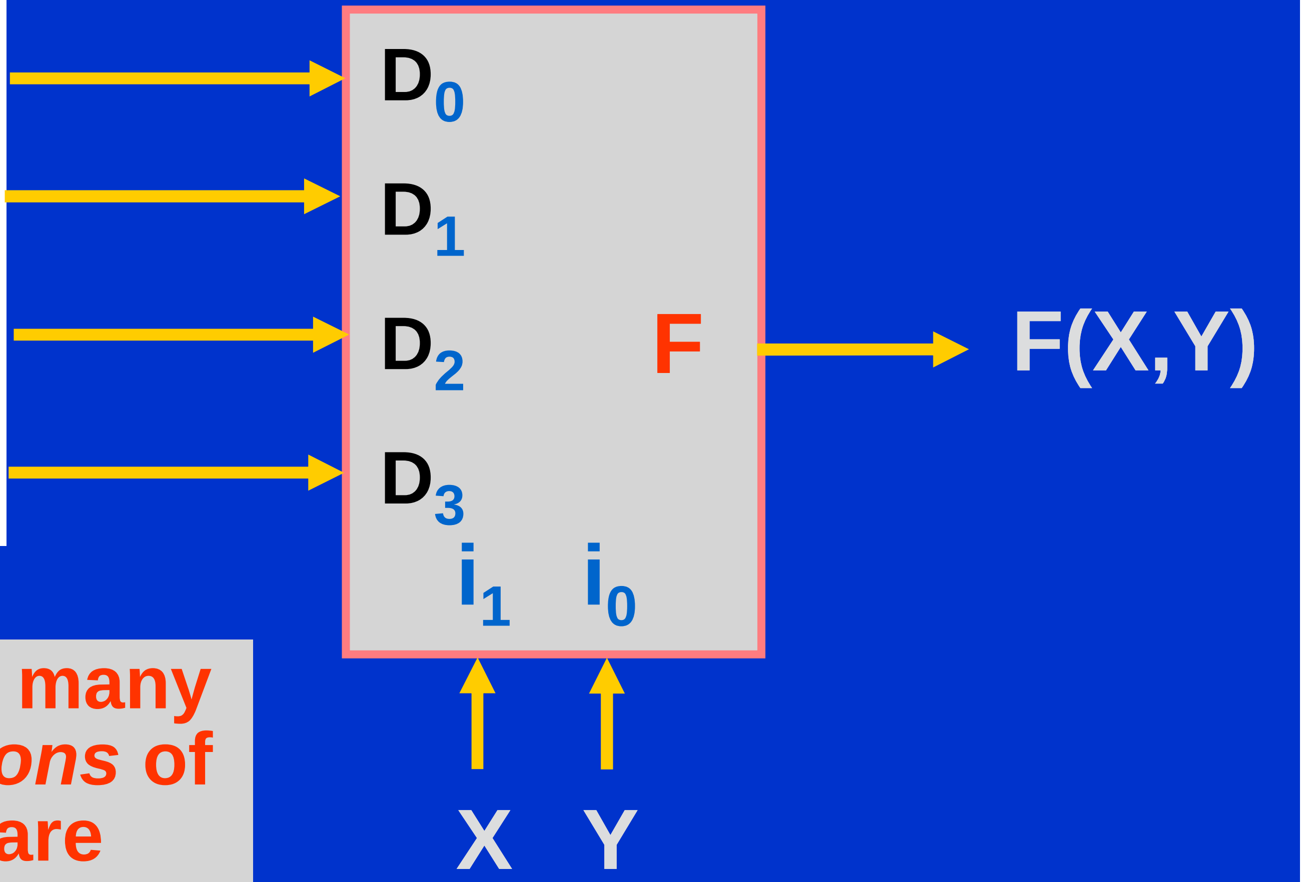
X	Y	F(X,Y)
0	0	1
0	1	0
1	0	0
1	1	1

XNOR function



Multiplexer Truth Table Analogy

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

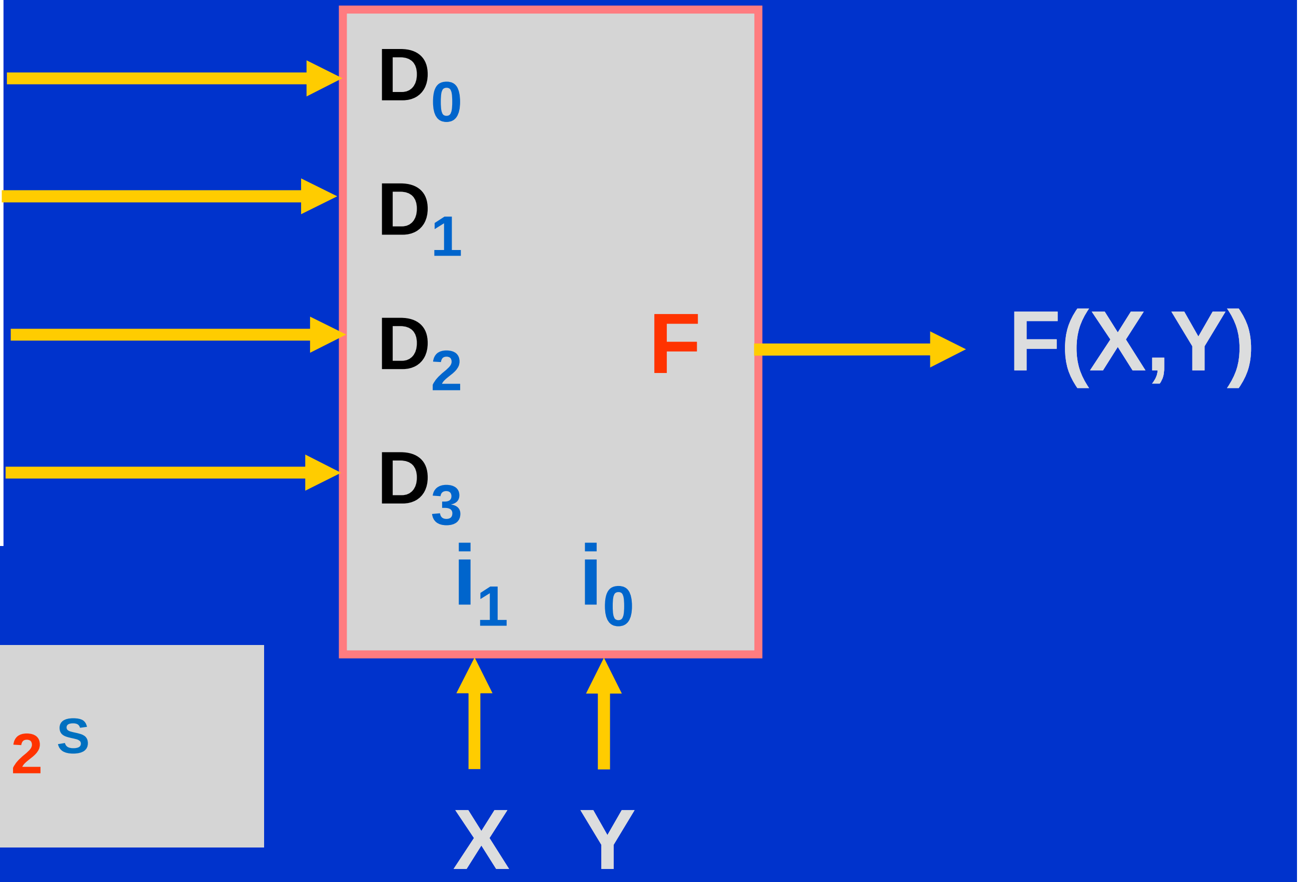


Question: How many *different functions* of S variables are possible?

Multiplexer Truth Table Analogy

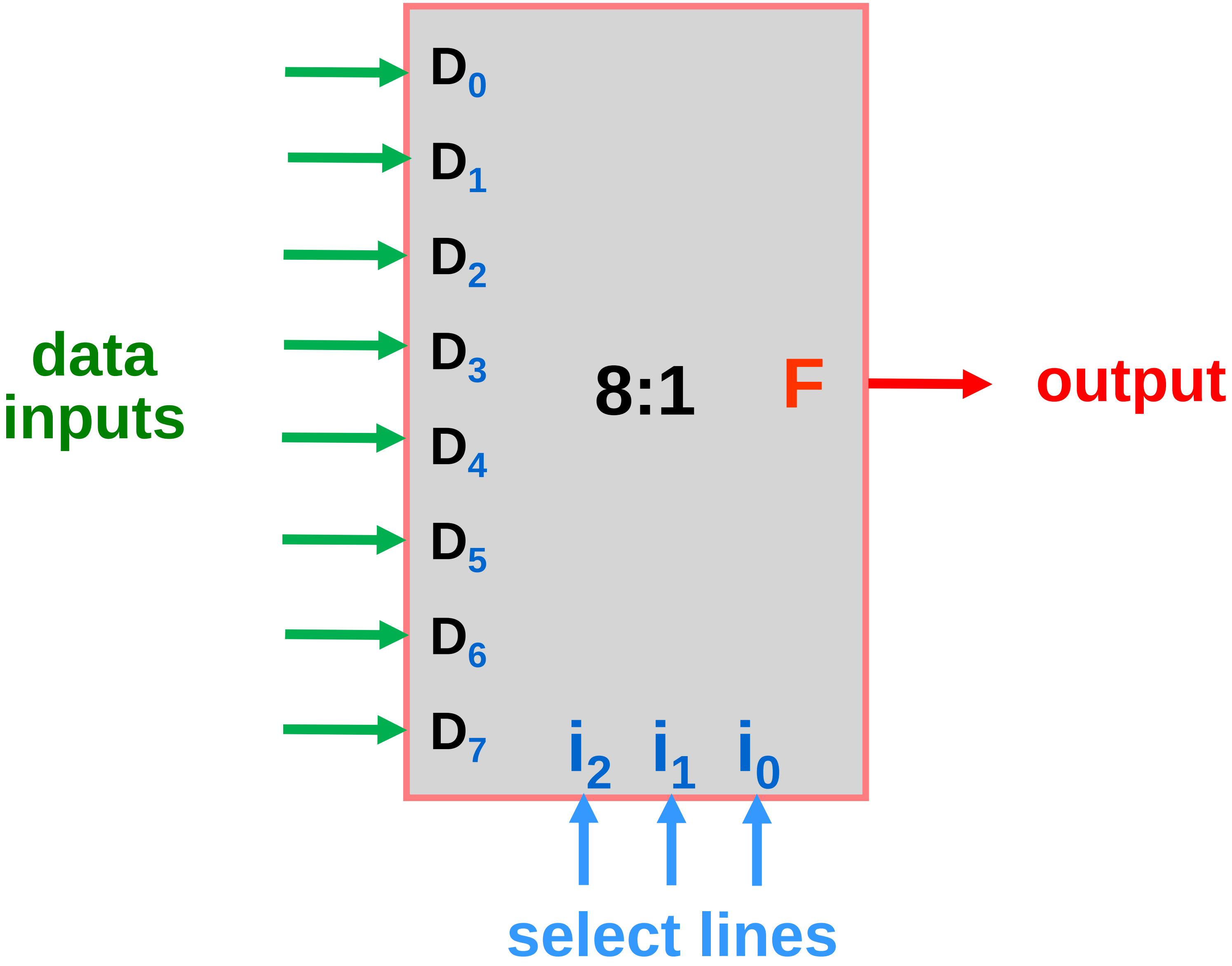
This is very similar to the look-up tables (LUTs) used in FPGAs

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



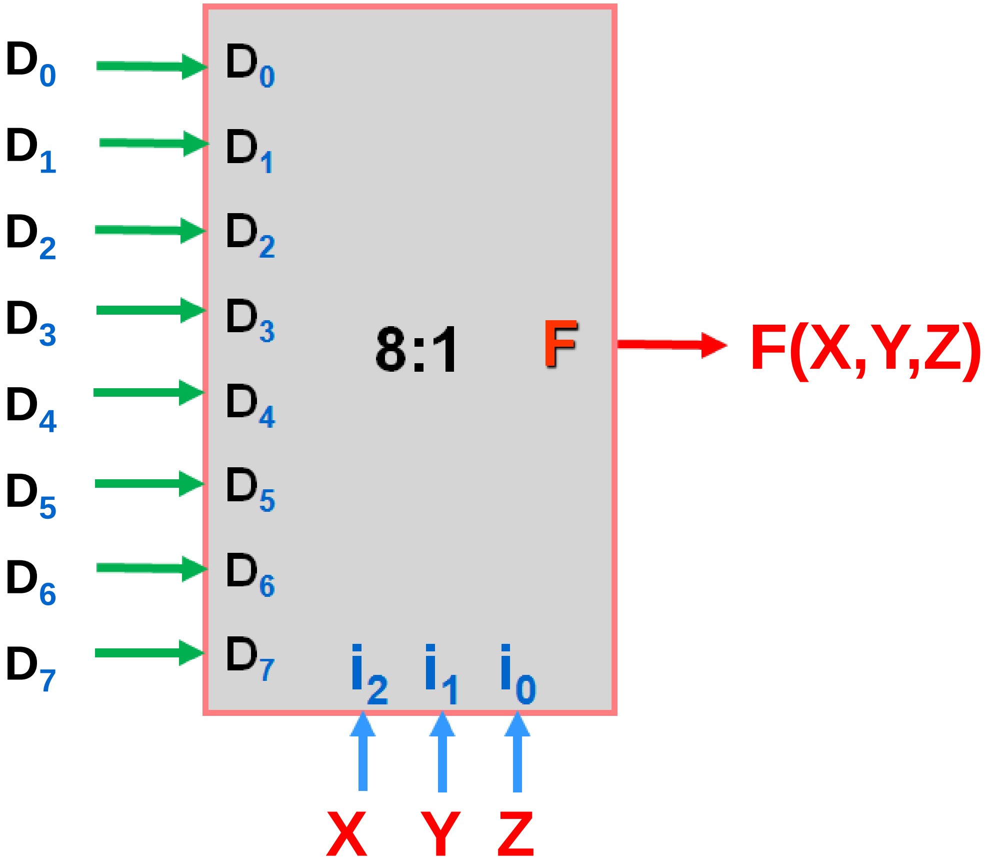
Answer: 2^{2^S}

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



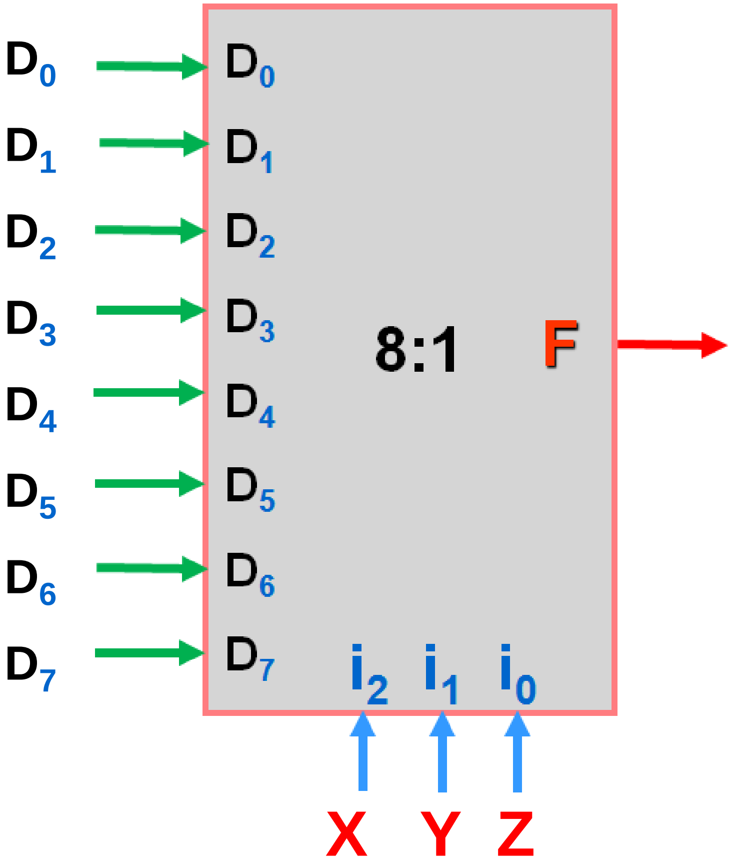
Example: Multiplexer Function Realization

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



Example: Multiplexer Function Realization

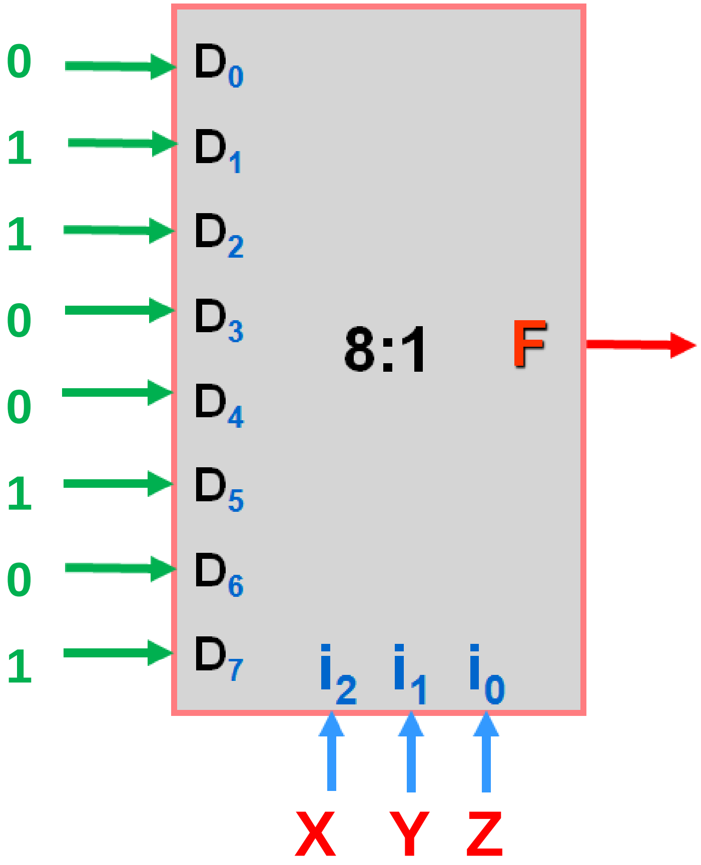
Determine the multiplexer data input values for realizing the function $F(X,Y,Z) = X \bullet Z + X' \bullet (Y \oplus Z)$



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

Example: Multiplexer Function Realization

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



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

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

$$= X \bullet Z + X' \bullet Y' \bullet Z + X' \bullet Y \bullet Z'$$

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

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

Multiplexers in ABEL

- **Multiplexer functionality can be expressed in ABEL several different ways:**
  - **using conventional sum-of-products expressions**
  - **using sets and relations**
  - **using when-else-then constructs**
- **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 × 1-bit Multiplexer

```
MODULE mux811

TITLE '8-to-1 X 1-bit Multiplexer Using 22V10’

DECLARATIONS

D0..D7 pin 2..9; " data inputs
EN pin 9; " function enable
S0..S2 pin 14..16; " select lines
Y pin 23 istype ‘com’; " output

EQUATIONS

Y = EN & (!S2&!S1&!S0&D0 #
          !S2&!S1& S0&D1 #
          !S2& S1&!S0&D2 #
          !S2& S1& S0&D3 #
           S2&!S1&!S0&D4 #
           S2&!S1& S0&D5 #
           S2& S1&!S0&D6 #
           S2& S1& S0&D7);
END
```

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

```
MODULE mux418a

TITLE '4-to-1 X 8-bit Multiplexer Using CPLD'

DECLARATIONS
EN pin; " tri-state output enable control line
S0..S1 pin; " select inputs
A0..A7, B0..B7, C0..C7, D0..D7 pin; " 8-bit input buses
Y0..Y7 pin istype 'com'; " 8-bit output bus

" Sets
A = [A0..A7];
B = [B0..B7];
C = [C0..C7];
D = [D0..D7];
Y = [Y0..Y7];

EQUATIONS
Y.OE = EN; " tri-state enable
Y = !S1&!S0&A # !S1&S0&B # S1&!S0&C # S1&S0&D;

END
```

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

```
MODULE mux418b
TITLE '4-to-1 X 8-bit Multiplexer Using CPLD'

DECLARATIONS
EN pin; " tri-state output enable control line
S0..S1 pin; " select inputs
A0..A7, B0..B7, C0..C7, D0..D7 pin; " 8-bit input buses
Y0..Y7 pin istype 'com'; " 8-bit output bus
" Sets
SEL = [S1..S0];
A = [A0..A7];
B = [B0..B7];
C = [C0..C7];
D = [D0..D7];
Y = [Y0..Y7];

EQUATIONS
Y.OE = EN; " tri-state enable
WHEN (SEL == 0) THEN Y = A;
ELSE WHEN (SEL == 1) THEN Y = B;
ELSE WHEN (SEL == 2) THEN Y = C;
ELSE WHEN (SEL == 3) THEN Y = D;

END
```

Clicker Quiz

```
MODULE bigmux

TITLE 'Big Multiplexer'

DECLARATIONS
EN pin;
S0..S1 pin;
A0..A7, B0..B7, C0..C7, D0..D7 pin;
Y0..Y7 pin istype 'com';

A = [A0..A7];
B = [B0..B7];
C = [C0..C7];
D = [D0..D7];
Y = [Y0..Y7];

EQUATIONS
Y.OE = EN;
Y = !S1&!S0&A # !S1&S0&B # S1&!S0&C # S1&S0&D;

END
```

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 outputs [Y0..Y7]:

A. will all be Hi-Z

B. will all be zero

C. will all be one

D. will be equal to the inputs [D0..D7]

E. none of the above

3. When EN=1, S1=1, and S0=1, the outputs [Y0..Y7]:

A. will all be Hi-Z

B. will all be zero

C. will all be one

D. will be equal to the inputs [D0..D7]

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