Lecture Summary – Module 2
Combinational Logic Circuits

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

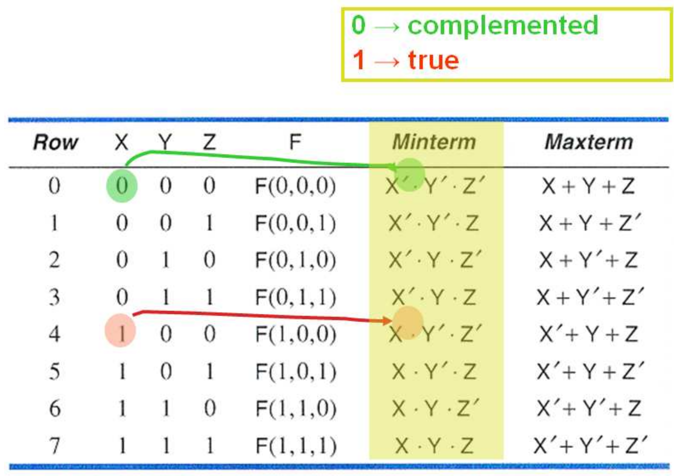
Learning Objectives:

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

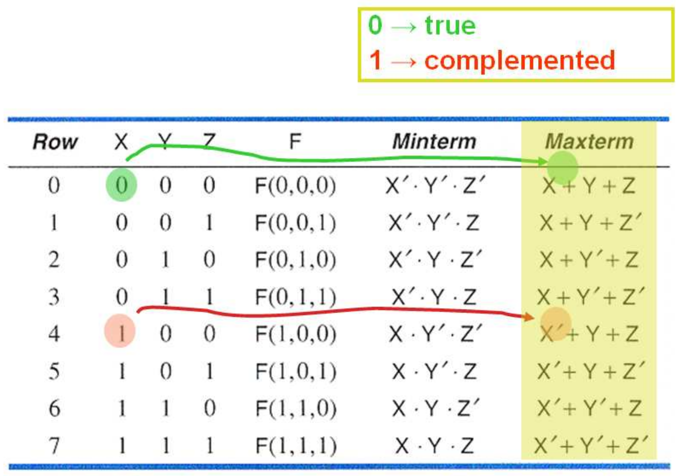
Lecture Summary – Module 2-A
Combinational Circuit Analysis and Synthesis

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 196-210

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


0 → complemented
1 → true

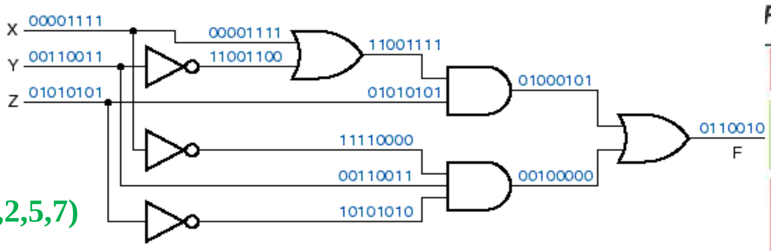
Row	X	Y	Z	F	*Minterm*	*Maxterm*
0	0	0	0	F(0,0,0)	$X' \cdot Y' \cdot Z'$	$X + Y + Z$
1	0	0	1	F(0,0,1)	$X' \cdot Y' \cdot Z$	$X + Y + Z'$
2	0	1	0	F(0,1,0)	$X' \cdot Y \cdot Z'$	$X + Y' + Z$
3	0	1	1	F(0,1,1)	$X' \cdot Y \cdot Z$	$X + Y' + Z'$
4	1	0	0	F(1,0,0)	$X \cdot Y' \cdot Z'$	$X' + Y + Z$
5	1	0	1	F(1,0,1)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

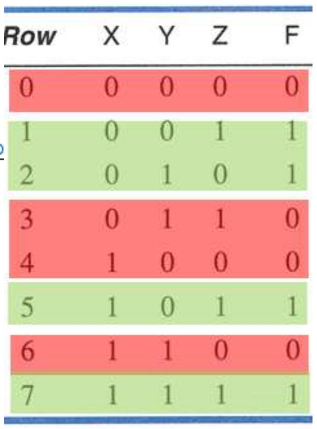

0 → true
1 → complemented

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

- **combinational analysis**

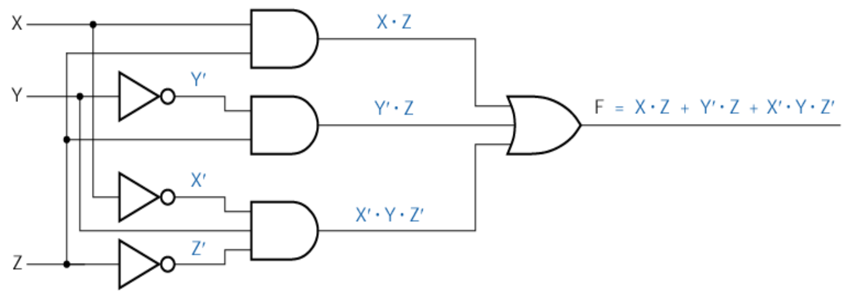




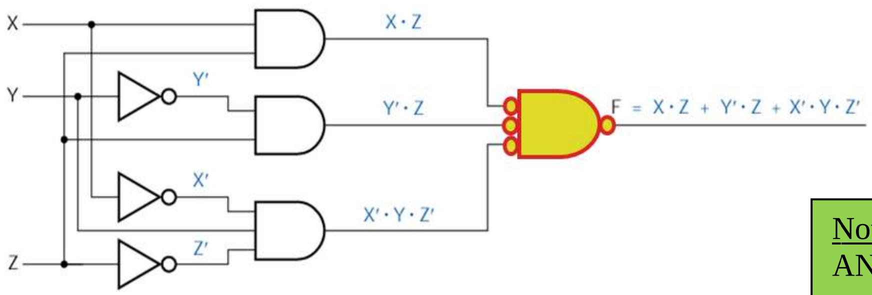
Row	X	Y	Z	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

  - **truth table**
  - **on set: $\Sigma_{X,Y,Z}$ (1,2,5,7)**
  - **canonical sum: f (X,Y,Z) = $X'\cdot Y'\cdot Z + X'\cdot Y\cdot Z' + X\cdot Y'\cdot Z + X\cdot Y\cdot Z$**
  - **off set: $\Pi_{X,Y,Z}$(0,3,4,6)**
  - **canonical product: f (X,Y,Z) = $(X+Y+Z)\cdot(X+Y'+Z')\cdot(X'+Y+Z)\cdot(X'+Y'+Z)$**

- **graphical application of DeMorgan's law**

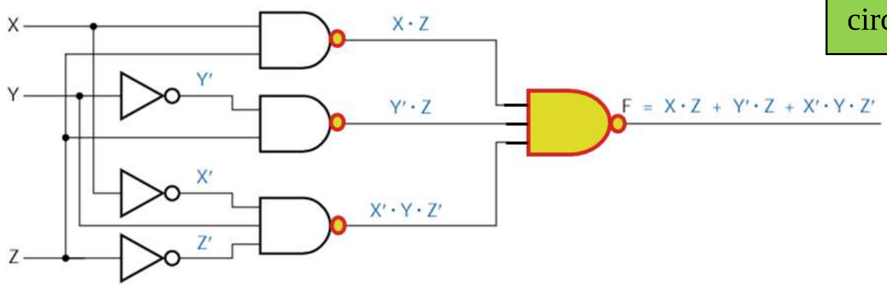


  - **step 1 – starting at the "output end", replace the original gate with its dual symbol and complement all its inputs and outputs**

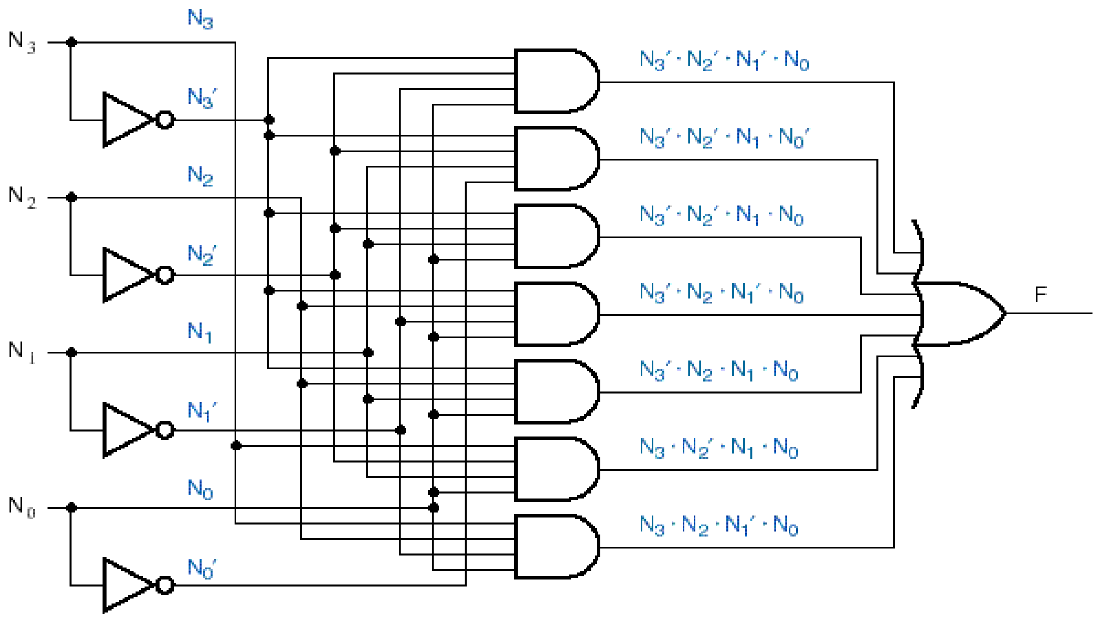


  - **step 2 – migrate the "inversion bubbles" by applying involution**

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



- **combinational synthesis**
  - **a circuit *realizes* ("makes real") an expression if its output function equals that expression (the circuit is called a *realization* of the function)**
  - **typically there are *many possible realizations* of the *same function***
    - **algebraic transformations**
    - **graphical transformations**
  - **starting point is often a "word description"**
    - **<u>example</u>: *Design a 4-bit prime number detector* (or, *Given a 4-bit input combination M = N3N2N1N0, 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* (N3,N2,N1,N0) = $\Sigma_{N3,N2,N1,N0}$(1,2,3,5,7,11,13)**

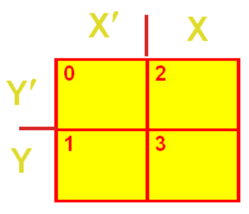


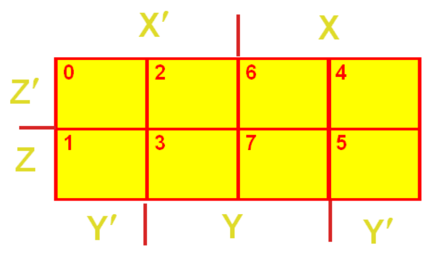
  - **but…how do we know if a given realization of a function is "best" in terms of:**
    - **speed (propagation delay)**
    - **cost**
      - **total number of gates**
      - **total number of gate inputs (fan-in)**
  - **Need two things:**
    - **a way to transform a logic function to its simplest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

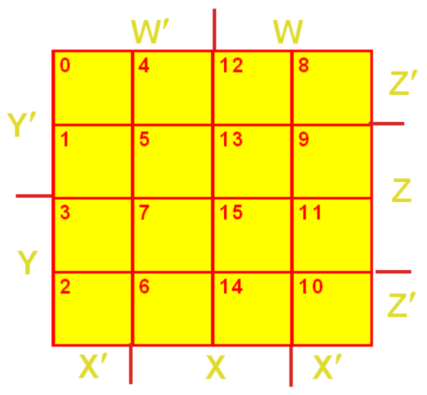
Lecture Summary – Module 2-B
Mapping and Minimization

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 210-222

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





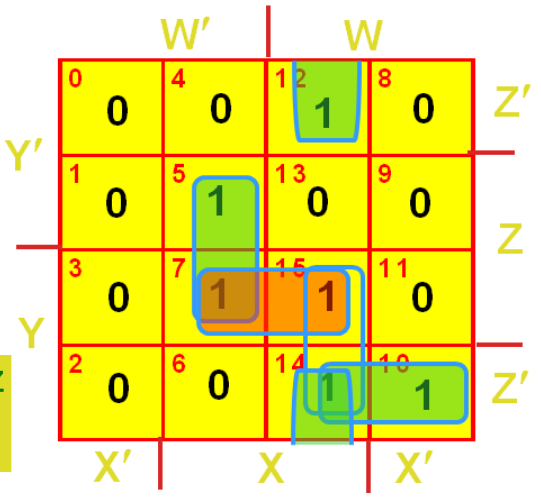


- **minimization**
  - **theory / terminology**
    - **the *minimal sum* has the fewest possible product terms (first-level gates / second-level gate inputs) and the fewest possible literals (first-level gate inputs)**
    - **prime implicants**
      - **a *prime implicant* is the *largest possible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

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

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

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

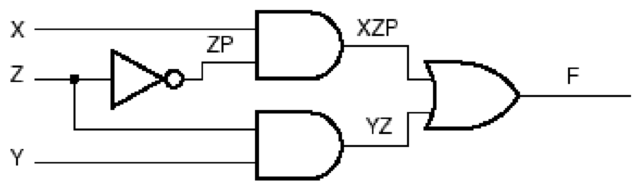


  - **procedure for finding minimal PoS expression: group 0's to find a minimal SoP expression for *f′*, then find an SoP expression for *f* by applying DeMorgan's Law → NOR-NOR realization**
  - **procedure for handling NAND-wired AND configuration: note that this configuration realizes the *complement* of a NAND-NAND circuit, so find a minimal SoP expression for *f′* (by grouping 0's) and implement these terms "directly" (as if it were NAND-NAND)**
  - **procedure for converting from one form of an expression to another: use a K-map!**
  - **procedure for handling *incompletely specified* functions (logic functions that do not assign a specific binary output value (0/1) to each of the 2^n input combinations)**
    - **unused combinations – called "*don't cares*" or "d-set"**
    - **rules for grouping**
      - **allow d's to be included when circling sets of 1's, to make the sets as large as possible**
      - **do *not* circle any sets that contain *only* d's**
      - **look for distinguished 1-cells only, *not* distinguished d-cells**

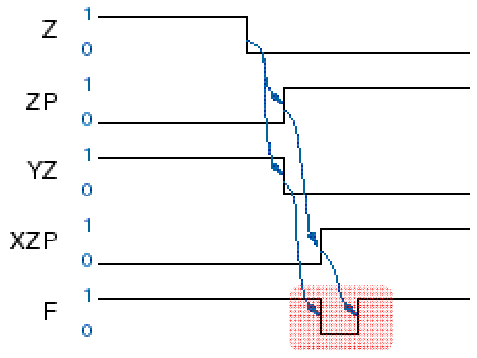
Lecture Summary – Module 2-C
Timing Hazards

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 224-229

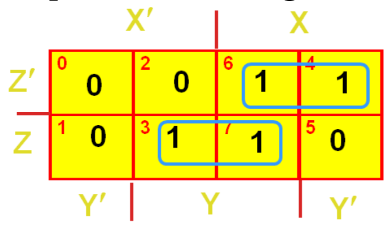
- **introduction**
  - **gate propagation delay may cause the transient behavior of logic circuit to *differ* from that predicted by steady state analysis**
  - **short pulse – glitch/hazard**
- **definitions**
  - **a static-1 hazard is a *pair of input combinations* that: (a) differ in only one input variable and (b) both produce a "1" output, such that it is possible for a momentary "0" output to occur during a transition in the differing input variable**



2:1 multiplexer circuit

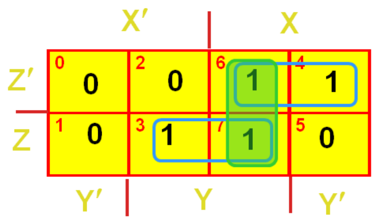


  - **a static-0 hazard is a *pair of input combinations* that: (a) differ in only one input variable and (b) both produce a "0" output, such that it is possible for a momentary "1" output to occur during a transition in the differing input variable (the *dual* of a static-1 hazard)**
- **prediction of static hazards from a K-map**
  - **<u>important</u>: the existence or nonexistence of static hazards depends on the *circuit design* (i.e., *realization*) of a logic function**
  - **cause of hazards: it is possible for the output to momentarily glitch to "0" if the AND gate that covers one of the combinations goes to "0" before the AND gate covering the other input combination goes to "1" (referred to as *break before make*)**



f (X,Y,Z) = X•Z′+Y•Z

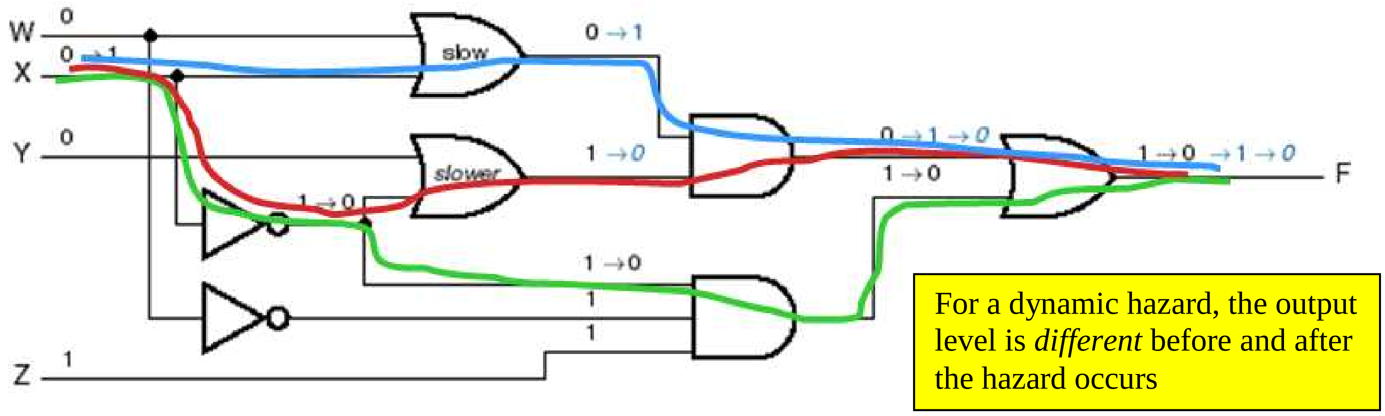
  - **use of consensus term(s) to eliminate hazards**



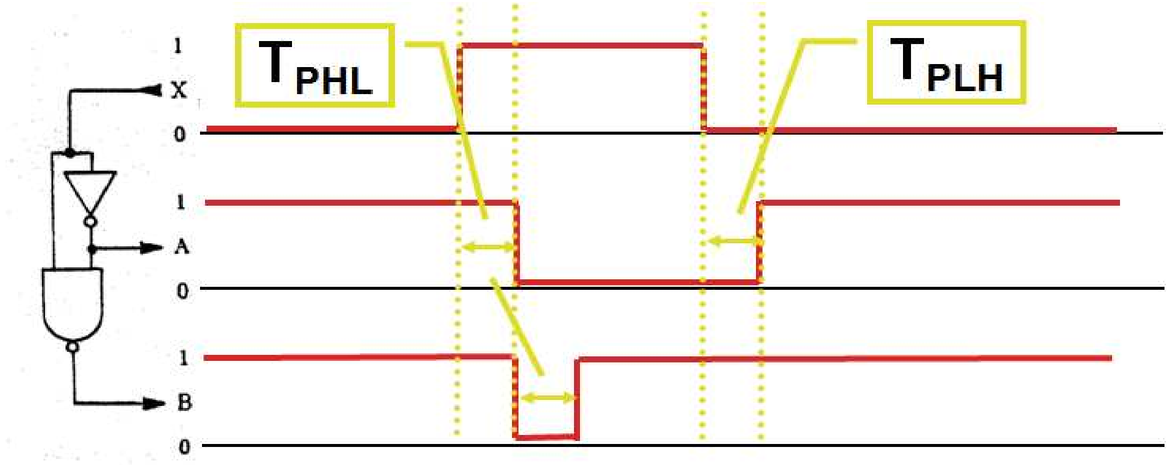
f (X,Y,Z) = X•Z′+Y•Z
+ X•Y

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

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


W
X
Y
Z
0
0 → 1
0
1
slow
slower
0 → 1
1 → 0
1 → 0
1 → 0
1
1
0 → 1 → 0
1 → 0
1 → 0 → 1 → 0
F
For a dynamic hazard, the output level is *different* before and after the hazard occurs

- **"useful" hazards – example: "leading edge" detector**



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

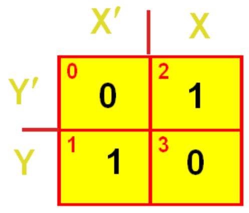
Lecture Summary – Module 2-D
XOR/XNOR Functions

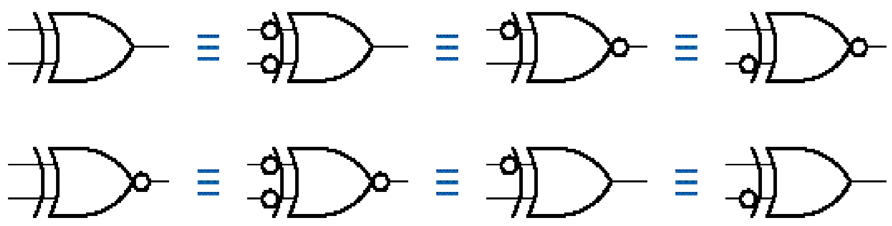
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 447-448

- **introduction**
  - **an *Exclusive-OR (XOR) gate* is a 2-input gate whose output is "1" if exactly one of its inputs is "1" (or, an XOR gate produces an output of "1" if its inputs are *different*)**
  - **an *Exclusive-NOR (XNOR) gate* is the *complement* of an XOR gate – it produces an output of "1" if its inputs are the *same***
  - **an XNOR gate is also referred to as an *Equivalence* (or *XAND*) gate**
  - **although XOR is not one of the basic functions of switching algebra, discrete XOR gates are commonly used in practice**
  - **the *"ring sum" operator* ⊕ is 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 XOR**

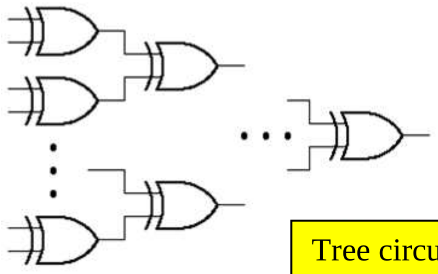
X	Y	X⊕Y	(X⊕Y)′
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

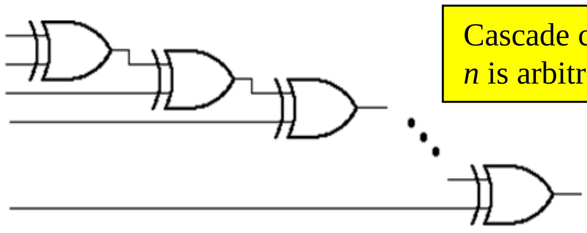
- **XOR properties**
  - **$X \oplus X = X' \bullet X + X \bullet X' = 0 + 0 = 0$**
  - **$X' \oplus X' = X \bullet X' + X' \bullet X = 0 + 0 = 0$**
  - **$X \oplus 1 = X' \bullet 1 + X \bullet 0 = X'$**
  - **$X' \oplus 1 = X \bullet 1 + X' \bullet 0 = X$**
  - **$(X \oplus Y)' = X \oplus Y \oplus 1$**
  - **$X \oplus Y = Y \oplus X$**
  - **$X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$**
  - **$X \bullet (Y \oplus Z) = (X \bullet Y) \oplus (X \bullet Z)$**
  - **"checkerboard" K-map → *non-reducible function***
  - **common/equivalent symbols**



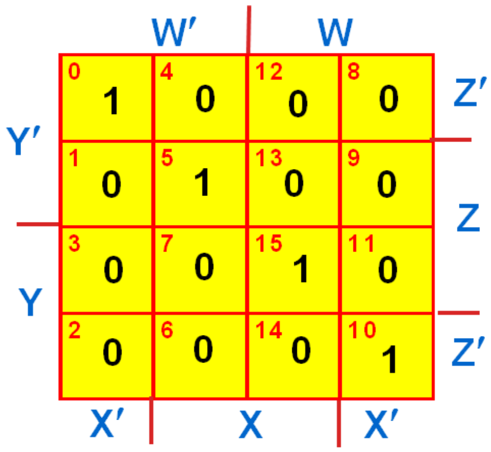


- **N-variable XOR/XNOR functions – tree and cascade circuits**
  - **the output of an n-variable XOR function is 1 if an <u>odd</u> number of inputs are 1**
  - **the output of an n-variable XNOR function is 1 if an <u>even</u> number of inputs are 1**
  - **realization of an n-variable XOR/XNOR function will require 2^{n-1} P-terms**


Tree circuit: *n* is a power of 2


Cascade circuit: *n* is arbitrary

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

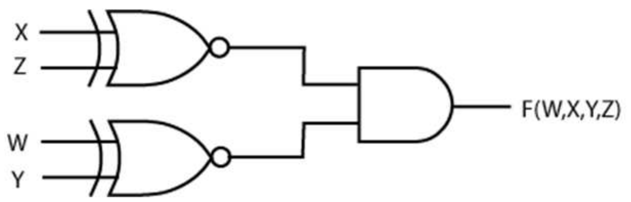


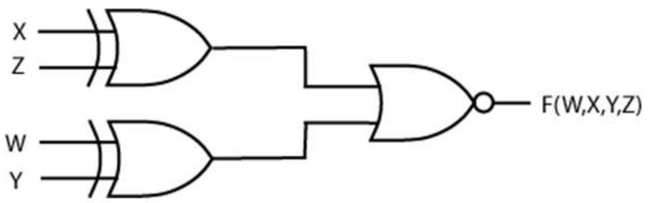
F(W,X,Y,Z) =
W′•X′•Y′•Z′ + W′•X•Y′•Z
+ W•X•Y•Z + W•X′•Y•Z′

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

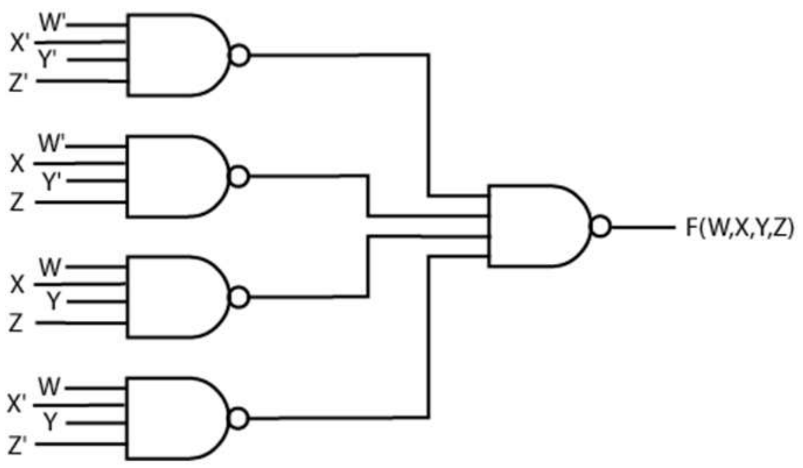
= (X′•Z′ +X•Z)•(W′•Y′+W•Y)

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





COST = 6 inputs + 3 outputs = 9

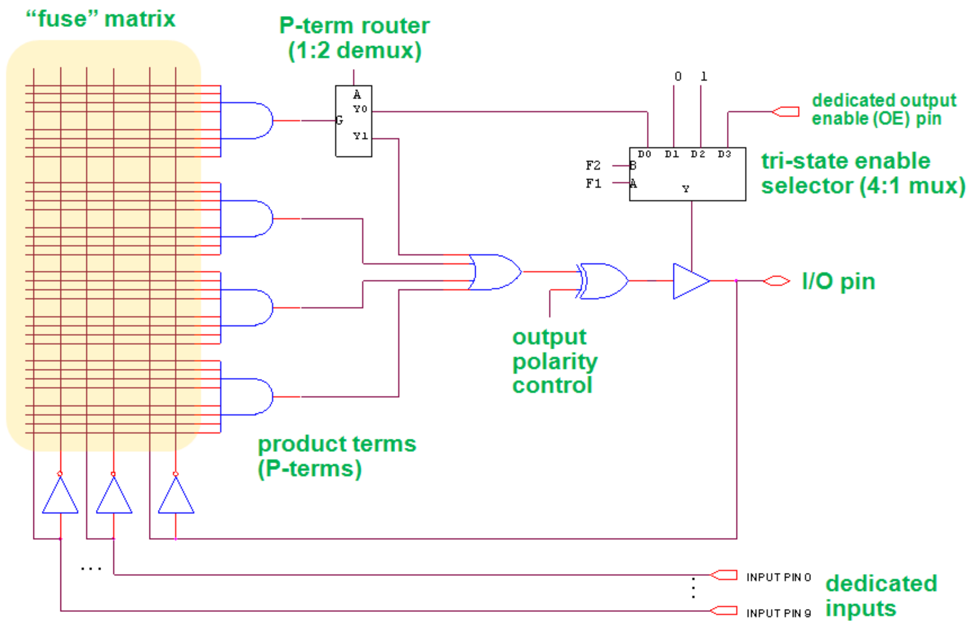


COST = 20 inputs + 5 outputs = 25

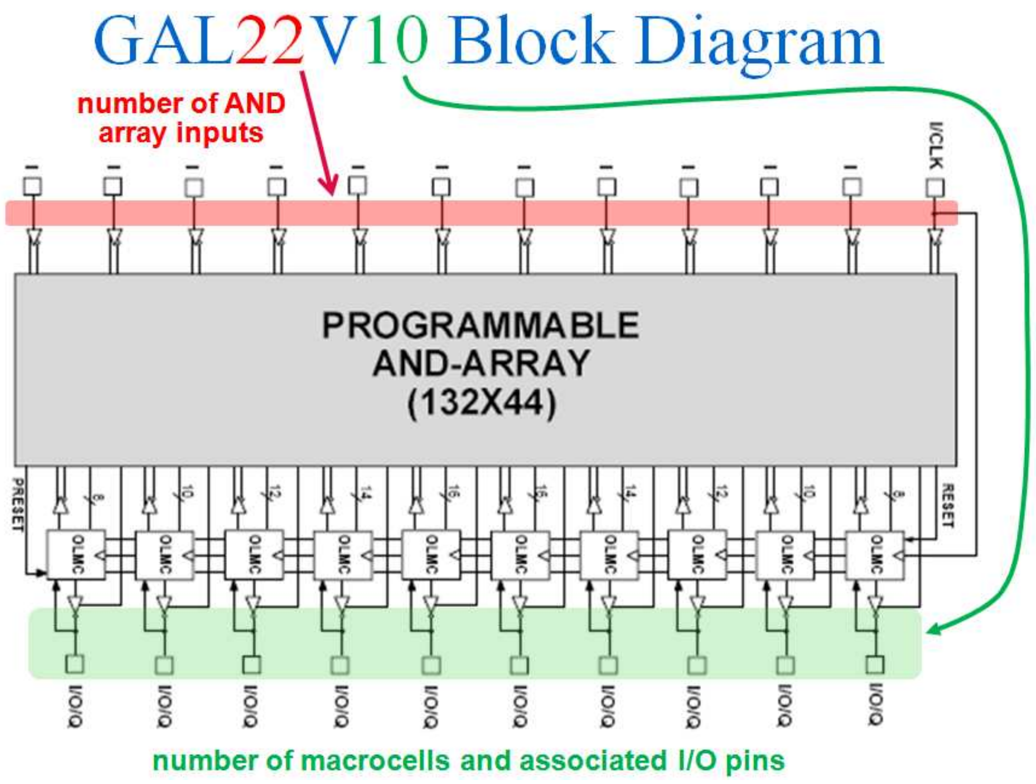
Lecture Summary – Module 2-E
Programmable Logic Devices

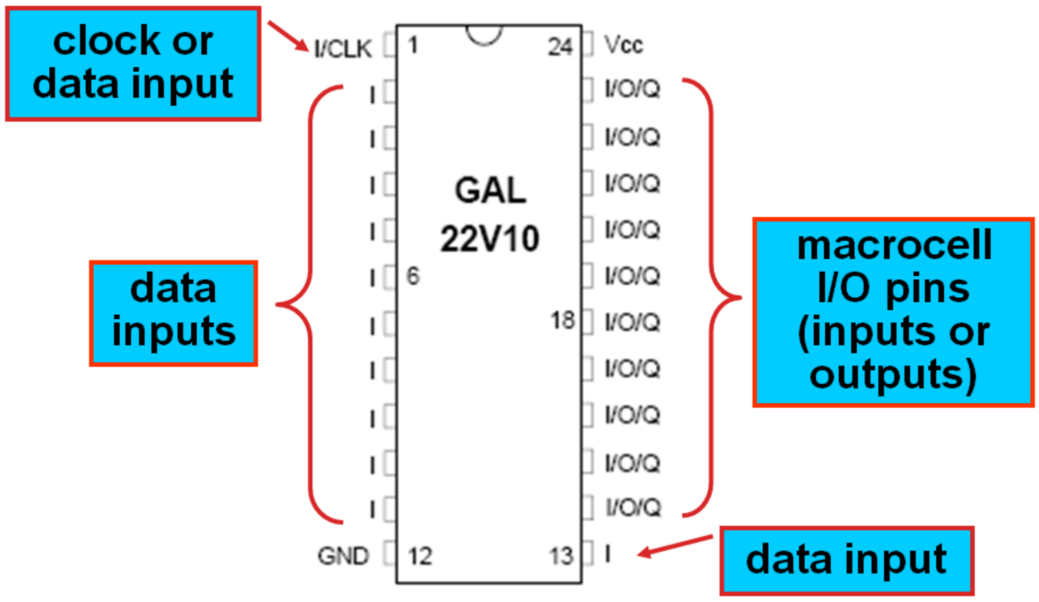
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 370-383, 840-859

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

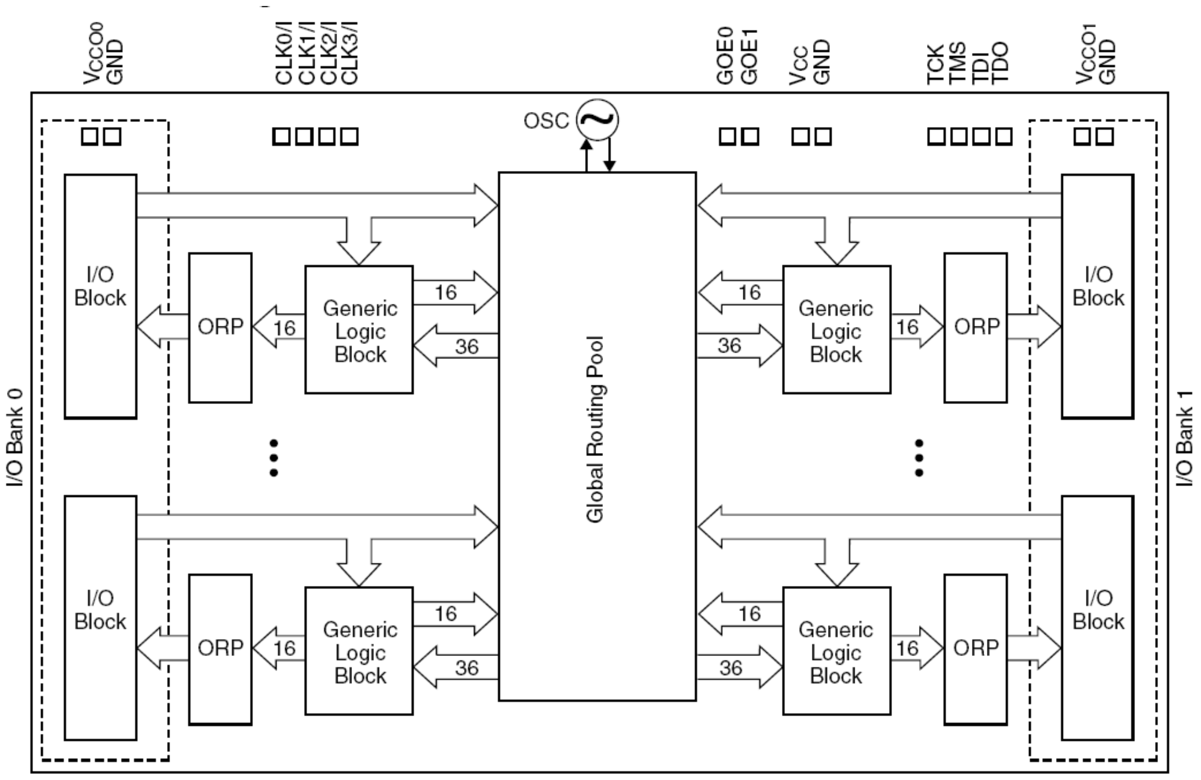


- **example – GAL22V10**


GAL22V10 Block Diagram
number of AND array inputs
I/CLK
PROGRAMMABLE AND-ARRAY (132X44)
PRESET
RESET
OLMC
I/O/Q
number of macrocells and associated I/O pins



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



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

Lecture Summary – Module 2-F

Hardware Description Languages

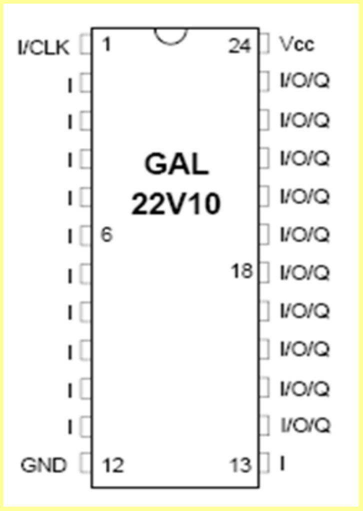
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 237-255

- **overview**
  - **hardware description languages (HDLs) are the most common way to describe the programming configuration of a CPLD or an FPGA**
  - **the first HDL to enjoy widespread use was PALASM ("PAL Assembler") from Monolithic Memories, Inc. (inventors of the PAL device)**
  - **early HDLs only supported equation entry**
  - **next generation languages such as CUPL (Compiler Universal for Programmable Logic) and ABEL (Advanced Boolean Expression Language) added more advanced capabilities:**
    - **truth tables and clocked operator tables**
    - **logic minimization**
    - **high-level constructs such as *when-else-then* and *state diagram***
    - **test vectors**
    - **timing analysis**
  - **VHDL and Verilog**
    - **started out as *simulation languages* (later developments in these languages allowed actual hardware design)**
    - **support modular, hierarchical coding and support a wide variety of high-level programming constructs – represents a *higher level of abstraction***
      - **arrays**
      - **procedures**
      - **function calls**
      - **conditional and iterative statements**
    - **potential pitfall – because VHDL and Verilog have their genesis as simulation languages, it is possible to create *non-synthesizable HDL code* using them (i.e., code that can *simulate* a digital system, but *not actually realize* it)**

- **concentration on use of 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 – free copy of software package available at latticesemi.com**
  - **ABEL program contents**
    - **documentation (program name, comments)**
    - **declarations that identify the inputs and outputs of the logic functions to be performed**
    - **statements that specify the logic functions to be performed**
    - **[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 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 modul**
- **basic symbols used in formulating ABEL equations**
  - **& AND**
  - **# OR**
  - **! NOT**
  - **$ XOR**
  - **!$ XNOR**
  - **= assignment**
- **example ABEL program with equations**

```
MODULE  abel_ex

TITLE  'ABEL Combinational Example for GAL22V10'

DECLARATIONS
" Input pins
A   pin   2;
B   pin   3;
C   pin   4;
D   pin   5;

" 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
```



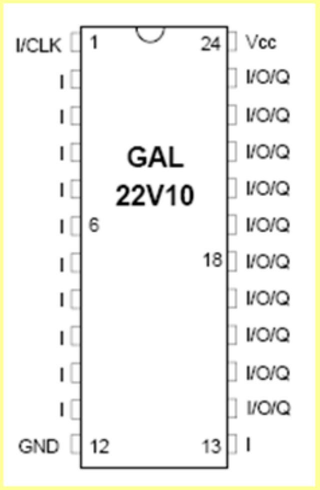
Note: Explicit pin declarations can be omitted and automatically assigned

- **example ABEL program with truth table**

```
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
```

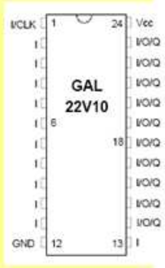
range

truth table declaration

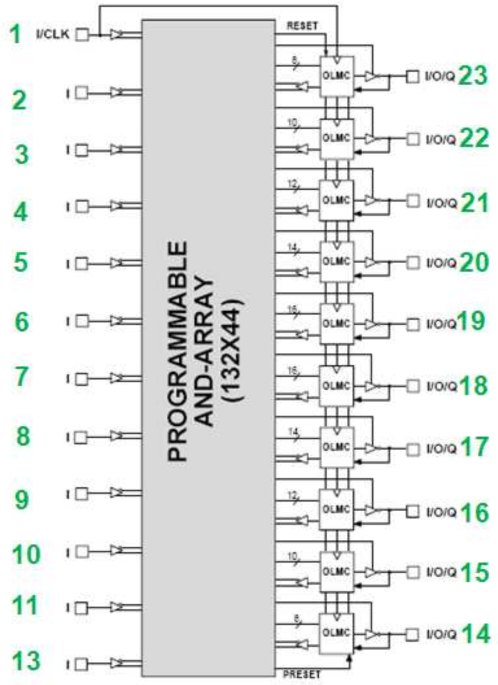


- **example ABEL program with multi-input XOR**

```
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
```



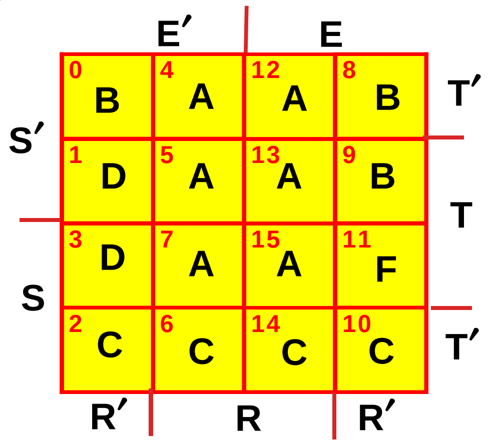
Equation requires 8 P-terms → can be realized on any 22V10 macrocell (any I/O pin)



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

```
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';

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
```

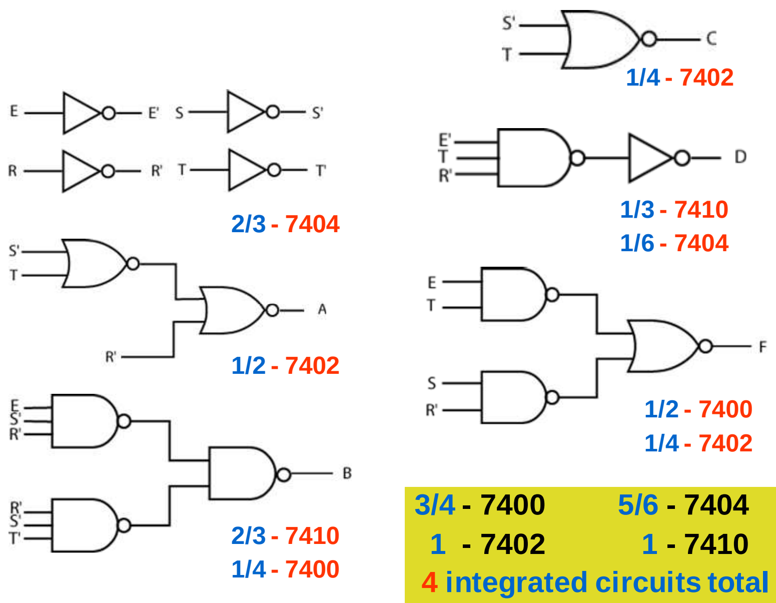
VS.



×5

```
                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   |
    |                           |
    +---------------------------+
```

VS.



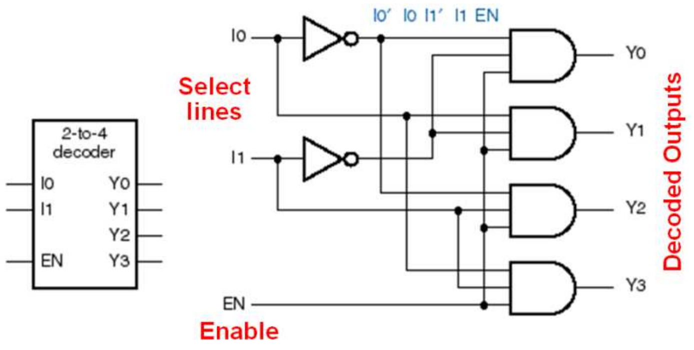
3/4 - 7400 5/6 - 7404
1 - 7402 1 - 7410
4 integrated circuits total

Lecture Summary – Module 2-G

Combinational Building Blocks: Decoders/Demultiplexers

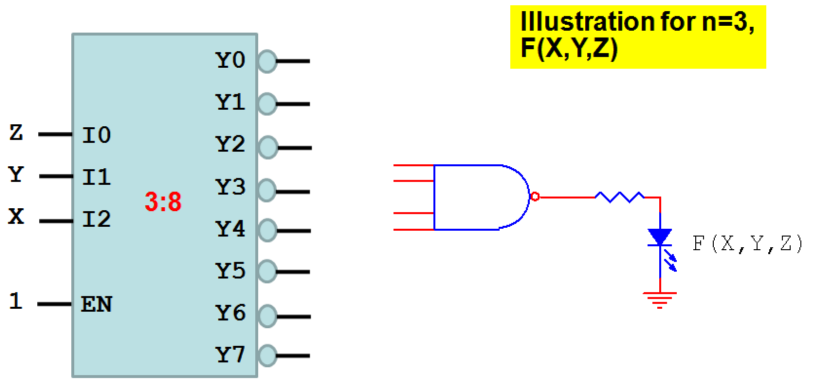
Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 384-398

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



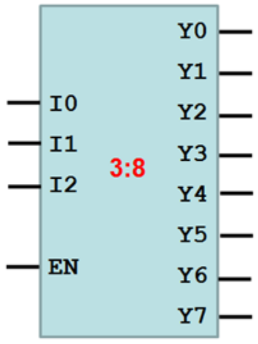
Note that **EN** can also be construed as a **digital input** that is routed to the **selected output**, in which case the circuit would be referred to as a **demultiplexer**

- **key observations**
  - **each output of an n to 2^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**
  - **if the decoder outputs are <u>active low</u>, a NAND gate can be used to "OR" the minterms of the function (representing its ON set)**
  - **if the decoder outputs are <u>active low</u>, an AND gate can be used to "OR" the minterms of the complement function (representing its OFF set)**
  - **a NAND gate (or AND gate) with at most 2^{n-1} inputs is needed to implement an arbitrary n-variable function using an n to 2^n binary decoder (that has <u>active low</u> outputs)**
- **general circuit for implementing an arbitrary n-variable function using a decoder, for case where ON set has ≤ 2^{n-1} members**



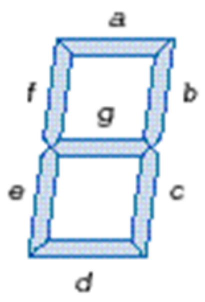
- **decoders/demultiplexers in ABEL**

```
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
```



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

```
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
```



Lecture Summary – Module 2-H

Combinational Building Blocks: Encoders and Tri-State Outputs

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 408-415

- **overview**
  - **an encoder is an "inverse decoder" – the role of inputs and outputs is reversed, and there are more input code bits than output code bits**
  - **common application: encode "device number" associated with service request**
  - **problem: more than one device may be requesting service at a given instant – motivation for "priority encoder"**
- **priority encoders**
  - **inputs are numbered, priority is assigned based on number (usually lowest number → lowest priority, etc., but *not always*)**
  - **easiest way to do this in ABEL is with a truth table**
  - **example – 8:3 priority encoder with "strobe output" to indicate if any of the encoder inputs have 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
```

ABEL-isms to note in this example: the "don't care" symbol (.X.), and how to define a "substitute symbol" (here, X); also note use of *ranges* to define the input and output pins.

- **examine the "reduced equation report" produced by ispLever**

```
Title: 8-to-3 Priority Encoder Using GAL22V10

 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);
```

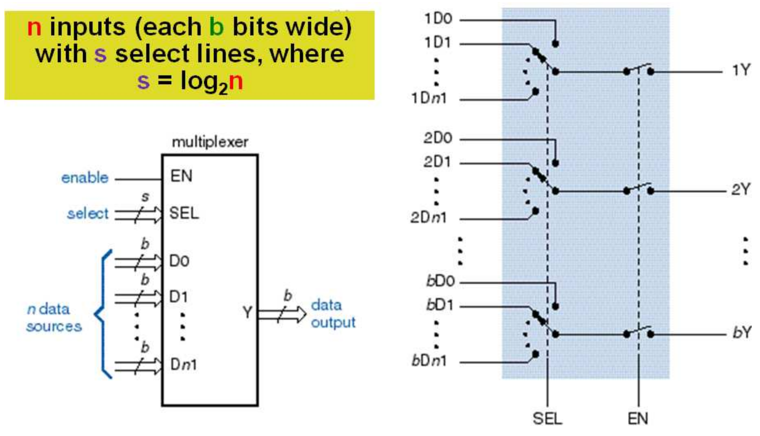
- **tri-state (three-state) outputs**
  - **in ABEL, an *attribute suffix* ".OE" is attached to a signal name on the left-hand side of an equation to indicate that the equation applies to the *output enable* for that signal**
  - **example: Create an ABEL file that implements a 4:2 priority encoder with tri-state encoded outputs (E1, E0). This design should include an active high output strobe (GS) that is asserted when any input is asserted.**

```
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
```

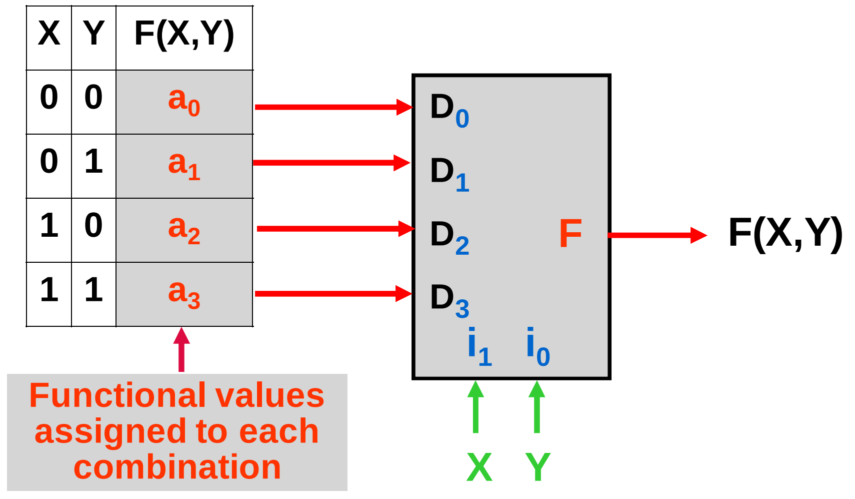
Lecture Summary – Module 2-I
Combinational Building Blocks: Multiplexers

Reference: *Digital Design Principles and Practices* (4th Ed.), pp. 432-443

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



- **truth table analogy**

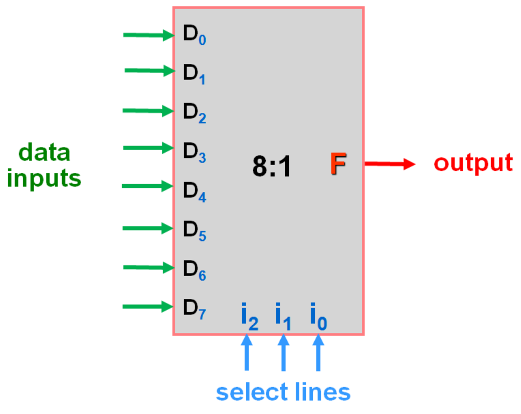


X	Y	F(X,Y)
0	0	a_0
0	1	a_1
1	0	a_2
1	1	a_3

Number of different functions of ***s*** variables possible:

$$2^{2^S}$$

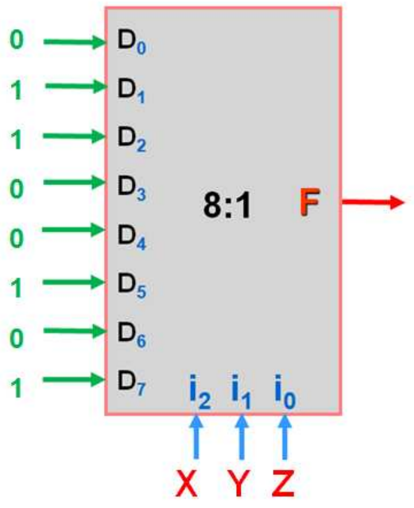
- **example: 8-to-1 (8:1) multiplexer**

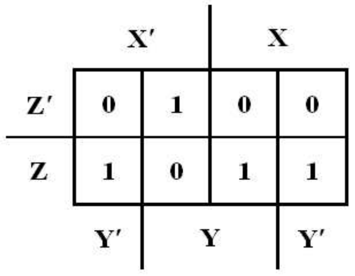


```
MODULE mux811
TITLE '8-to-1 X 1-bit Mux Using 22V10'
DECLARATIONS
D0..D7 pin 2..9; " data inputs
EN pin 9; " function enable
S0..S2 pin 14..16; " select lines
Y pin 23 istype 'com'; " output
EQUATIONS
Y = EN & (!S2&!S1&!S0&D0 #
            !S2&!S1& S0&D1 #
            !S2& S1&!S0&D2 #
            !S2& S1& S0&D3 #
             S2&!S1&!S0&D4 #
             S2&!S1& S0&D5 #
             S2& S1&!S0&D6 #
             S2& S1& S0&D7);
END
```

- **example – multiplexer function realization**

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





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

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

```
MODULE mux418a

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

DECLARATIONS
EN pin; " output enable control line
S0..S1 pin; " select inputs

" 8-bit input buses
A0..A7, B0..B7, C0..C7, D0..D7 pin;

" 8-bit output bus
Y0..Y7 pin istype ‘com’;

" Sets
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
```

```
MODULE mux418b

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

DECLARATIONS
EN pin; " output enable control line
S0..S1 pin; " select inputs

" 8-bit input buses
A0..A7, B0..B7, C0..C7, D0..D7 pin;

" 8-bit output bus
Y0..Y7 pin istype ‘com’;

" Sets
SEL = {S1..S0];
A = [A0..A7];
B = [B0..B7];
C = [C0..C7];
D = [D0..D7];
Y = [Y0..Y7];

EQUATIONS
Y.OE = EN;
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
```