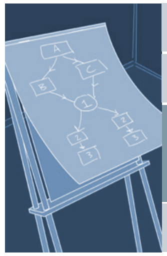
Chapter 2.
Combinational Logic Circuits

Apr., 2008



- **Logic Minimization**
  - Reduces complexity of the gate level implementation
    - Reduce number of literals (gate inputs)
    - Reduce number of gates
    - Reduce number of levels of gates

  - Two-Level Logic Minimization
    1. Apply the laws and theorems to simplify Boolean equations
    2. Karnaugh Map (K-Map) Method
    3. Quine-McCluskey Method
       - Tabular method to systematically find all prime implicants
       - 컴퓨터 논리회로(이상범 저), “**테이블 방법의 간소화**” 참조 (127페이지)
    4. CAD Tools for Simplification
       - Petrick’s Method
       - Espresso Method
       - …

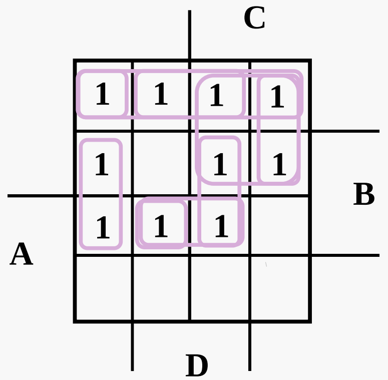
Practical Optimization

- Problem: Automated optimization algorithms:
  - require minterms as starting point,
  - require determination of all prime implicants, and/or
  - require a selection process with a potentially very large number of candidate solutions to be found.

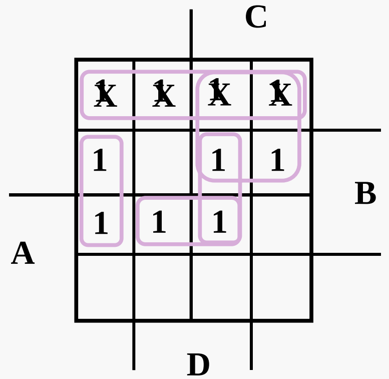
- Solution: Suboptimum algorithms not requiring any of the above in the general case

Example Algorithm: Espresso

- Illustration on a K-map:



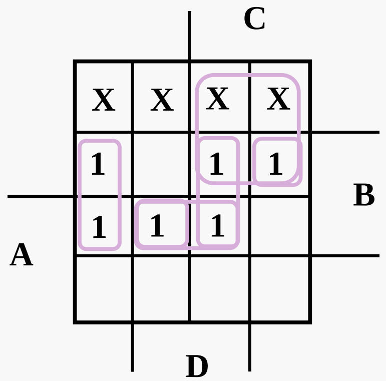
Original F & EXPAND



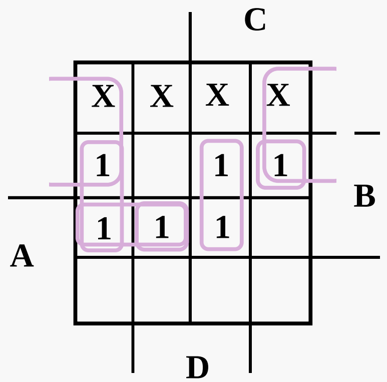
ESSENTIAL & IRREDUNDANT COVER

Example Algorithm: Espresso

- Continued:



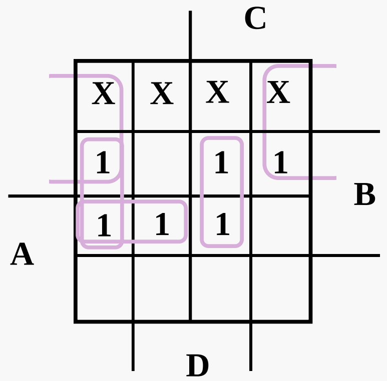
REDUCE



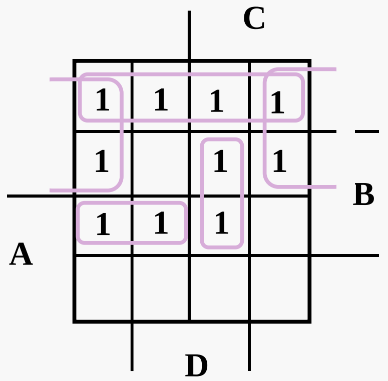
EXPAND

Example Algorithm: Espresso

- Continued:



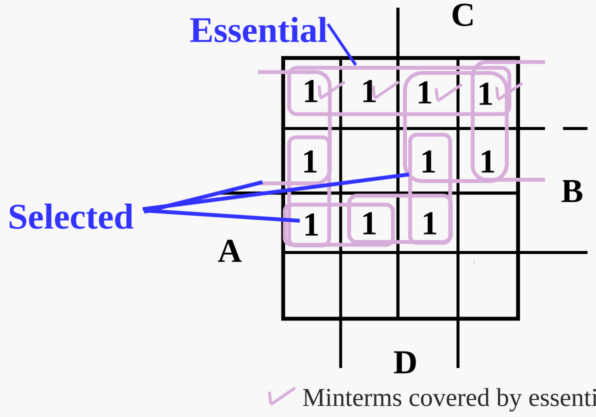
IRREDUNDANT COVER



After REDUCE, EXPAND, IRREDUNDANT COVER, LAST GASP, QUIT

Example Algorithm: Espresso

- This solution costs 2 + 2 + 3 + 3 + 4 = 14
- Finding the optimum solution and comparing:



- There are two optimum solutions one of which is the one obtained by Espresso.

Espresso;

- Two-Level Logic Minimization Tool made by the CAD group at UC Berkeley
  - Takes as input a two-level representation of a two-valued Boolean function
  - Produces a minimal equivalent representation
  - With options, user can specify exact optimization algorithm
- Input Format
  - Command line should start with dot(.)

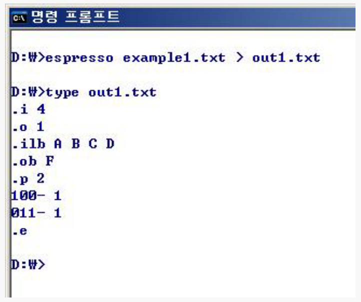
.i	# input
.o	# output
.ilb	input_node0, {input_node1, …, }
.ob	output_node0, {output_node1, … , }
.p	number of non-zero truth table entry
.e	end of input

- Input logic is given as follows : [input_value] [output_value]
- e.g. 0001 [1개 이상의 공백] 1

  → 4개의 입력 변수가 0001로 주어지면 출력은 1(logic-high)로 나타난다
- Example of Input,

.i	4	: number of input variables = 4
.o	1	: number of output variables = 1
.ilb	a b c d	: input variable names e.g. a b c d
.ob	f	: output function name e.g. f
.p	4	: number of non-zero truth table entry
0110	1	
0101	1	
1010	1	
1110	-	: Don't care condition
.e		: end of input

실행 예



```
D:₩>espresso example1.txt > out1.txt

D:₩>type out1.txt
.i 4
.o 1
.ilb A B C D
.ob F
.p 2
100- 1
011- 1
.e

D:₩>
```

Interpret the output

- **.p 2;** indicates that there are two terms in the output expression
- **100- 1;** this term is AB'C' (Note, B' is B inverse), so this is read as A and not B and not C.
- **011- 1;** this term is A'BC
- The logic expression is thus

  F = AB'C' + A'BC.
- In the output lines, 1 is the variable, o is the inverse and – means the variable is not involved.
- Tip; specifying the truth table entries only where the functions 1 is sufficient to define the entire truth table.

Example of ESPRESSO Input/Output

ƒ(A,B,C,D) = ▯m(4,5,6,8,9,10,13) + d(0,7,15)

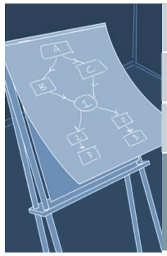
Espresso Input

```
.i 4              -- # inputs
.o 1              -- # outputs
.ilb a b c d      -- input names
.ob f             -- output name
.p 10             -- number of product terms
0100   1          -- A'BC'D'
0101   1          -- A'BC'D
0110   1          -- A'BCD'
1000   1          -- AB'C'D'
1001   1          -- AB'C'D
1010   1          -- AB'CD'
1101   1          -- ABC'D
0000   -          -- A'B'C'D' don't care
0111   -          -- A'BCD don't care
1111   -          -- ABCD don't care
.e                -- end of list
```

Espresso Output

```
.i 4
.o 1
.ilb a b c d
.ob f
.p 3
1-01   1
10-0   1
01--   1
.e
```

ƒ = AC'D + AB'D' + A'B



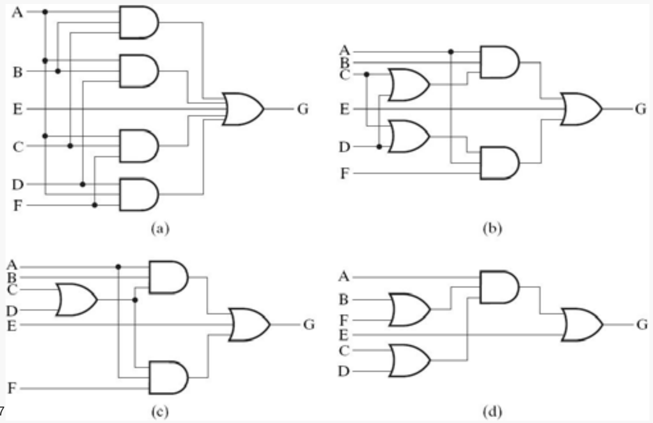
4. 2-level Circuit Optimization
5. Map Manipulation
6. Pragmatic Two-Level Optimization

7. Multiple-Level Circuit Optimization

Multiple-Level Circuit Optimization

- Although we have found that 2-level circuit optimization can reduce the cost of combinational logic circuits, often additional cost savings are available by using circuits with more than two levels.
  - Page 94~05, Figure 2-20 에서 (a), (b), (c) 와 (d) 의 gate input cost 는?

    G = ABC + ABD + E + ACF + ADF



Multiple-Level Circuit Optimization

- *Multiple-level circuits* are defined as circuits that are not two-level (with or without input and/or output inverters)
- Multiple-level circuits can have reduced gate input cost compared to two-level (SOP and POS) circuits
- Multiple-level optimization is performed by applying transformations to circuits represented by equations while evaluating cost

Transformations

1. Factoring(인수분해) - finding a factored form (인수분해된 형태) from SOP or POS expression
   - Algebraic - No use of axioms specific to Boolean algebra such as complements or idempotence
   - Boolean - Uses axioms unique to Boolean algebra
2. Decomposition(분해) - expression of a function as a set of new functions
3. Substitution(대체) of *G* into *F* - expression function *F* as a function of *G* and some or all of its original variables
4. Elimination(제거) - Inverse of substitution
5. Extraction(추출) – expression of multiple function as a set of new functions

   - decomposition applied to multiple functions simultaneously

Transformation Examples (1)

(1) Algebraic Factoring

$F = \overline{A}\,\overline{C}\,\overline{D} + \overline{A}B\overline{C} + ABC + AC\overline{D}$ **G=16**

- Factoring:

$F = \overline{A}\,(\overline{C}\,\overline{D} + B\overline{C}) + A\,(BC + C\overline{D})$ **G=16**

- Factoring again:

$F = \overline{A}\,\overline{C}\,(B + \overline{D}) + AC\,(B + \overline{D})$ **G=12**

- Factoring again:

$F = (\overline{A}\,\overline{C} + AC)(B + \overline{D})$ **G=10**

Transformation Examples (2)

(2) Decomposition

- The terms $(B + \overline{D})$ and $(\overline{A}\,\overline{C} + AC)$ can be defined as new functions E and H respectively, decomposing F:

F = E H, where **$E = B + \overline{D}$,** and **$H = \overline{A}\,\overline{C} + AC$** **G=10**

- This series of transformations has reduced G from 16 to 10, a substantial savings. The resulting circuit has three levels plus input inverters.

Transformation Examples (3)

(3) Substitution of E into F

- Returning to F just before the final factoring step:

$F = \overline{A}\,\overline{C}(B + \overline{D}) + AC(B + \overline{D})$ **G=12**

- Defining **$E = B + \overline{D}$**, and substituting in F :

$F = \overline{A}\,\overline{C}E + ACE$ **G=10**

- This substitution has resulted in the same cost as the decomposition

Transformation Examples (4)

(4) Elimination

- Beginning with a new set of functions:

X = B + C

Y = A + B

$Z = \overline{A}X + CY$ **G=10**

- Eliminating X and Y from Z:

$Z = \overline{A}(B + C) + C(A + B)$ **G=10**

- "*Flattening* (평탄화)" (Converting to SOP expression):

$Z = \overline{A}B + \overline{A}C + AC + BC$ **G=12**

- This has increased the cost, but has provided an new SOP expression for two-level optimization.

Transformation Examples (4)

- Two-level Optimization
  - The result of 2-level optimization (using K-Map) is:

$Z = \overline{A}B + C$ G=4

  - This example illustrates that:
    - Optimization can begin with any set of equations, not just with minterms or a truth table
    - Increasing gate input count G temporarily during a series of transformations can result in a final solution with a smaller G

Transformation Examples (5)

(5) Extraction

- Beginning with two functions:

$E = \overline{A}\overline{B}\overline{D} + \overline{A}BD$

$H = \overline{B}C\overline{D} + BCD$ G=16

- Finding a common factor and defining it as a function:

$F = \overline{B}\overline{D} + BD$

- We perform extraction by expressing E and H as the three functions:

$F = \overline{B}\overline{D} + BD$, $E = \overline{A}F$, $H = CF$ G=10

- The reduced cost G results from the sharing of logic between the two output functions

- Transformation example on text book
  - Page 96~97, Example 2-16

$G = A\overline{C}D + A\overline{C}F + A\overline{D}E + A\overline{D}F + BCD\overline{E}\overline{F}$

$H = \overline{A}BCD + ABE + ABF + BCD + BCF$

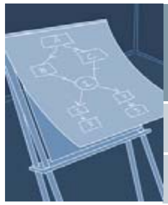
1. Algebraic Factoring (P.96)
2. Decomposition (P.96)
3. Substitution (P.96~97)
4. Extraction (P.97)

Terms of Use

8. Other Gate Types

9. Exclusive-OR Operator and Gates

10. High-Impedance Outputs



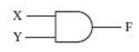
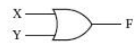
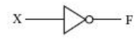
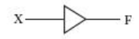
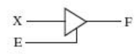
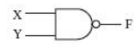
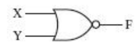
Overview

- Part 1 – Gate Circuits and Boolean Equations
  - Binary Logic and Gates
  - Boolean Algebra
  - Standard Forms
- Part 2 – Circuit Optimization
  - Two-Level Optimization
  - Map Manipulation
  - Practical Optimization (Espresso)
  - Multi-Level Circuit Optimization
- Part 3 – Additional Gates and Circuits
  - Other Gate Types
  - Exclusive-OR Operator and Gates
  - High-Impedance Outputs

Other Gate Types

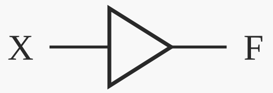
- Why?
  - Implementation feasibility and low cost
  - Power in implementing Boolean functions
  - Convenient conceptual representation
- Gate classifications
  - Primitive (단순) gate - a gate that can be described using a single primitive operation type (AND or OR) plus an optional inversion(s).
  - Complex (복합) gate - a gate that requires more than one primitive operation type for its description
- Primitive gates will be covered first

- Primitive Gate (Fig 2-22, Page 101)

Name	Distinctive-Shape Graphics Symbol	Algebraic Equation	Truth Table
AND		$F = XY$	X Y | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = X + Y$	X Y | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT (inverter)		$F = \overline{X}$	X | F 0 | 1 1 | 0
Buffer		$F = X$	X | F 0 | 0 1 | 1
3-State Buffer			E X | F 0 0 | Hi-Z 0 1 | Hi-Z 1 0 | 0 1 1 | 1
NAND		$F = \overline{X \cdot Y}$	X Y | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{X + Y}$	X Y | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0

Buffer

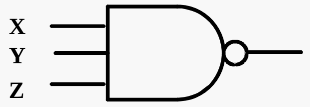
- A buffer is a gate with the function F = X:



- In terms of Boolean function, a buffer is the same as a connection!
- So why use it?
  - A buffer is an electronic amplifier used to (1) improve circuit voltage levels and (2) increase the speed of circuit operation.

NAND Gate

- The basic NAND gate has the following symbol, illustrated for three inputs:
  - AND-Invert (NAND)

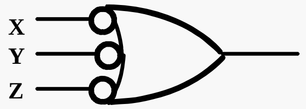


$F(X,Y,Z) = \overline{X \times Y \times Z}$

- NAND represents NOT AND, i. e., the AND function with a NOT applied. The symbol shown is an AND-Invert. The small circle ("*bubble*") represents the invert function.

NAND Gates (continued)

- Applying DeMorgan's Law gives Invert-OR (NAND)

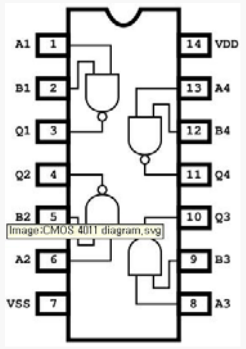


$F(X,Y,Z) = \overline{X} + \overline{Y} + \overline{Z}$

- This NAND symbol is called Invert-OR, since inputs are inverted and then ORed together.
- AND-Invert and Invert-OR both represent the NAND gate. Having both makes visualization of circuit function easier.
- A NAND gate with one input degenerates to an inverter.

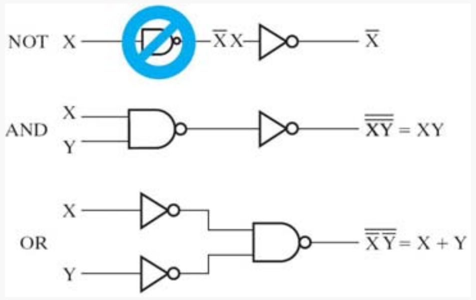
NAND Gates (continued)

- The NAND gate is the natural implementation for CMOS technology in terms of chip area and speed.
  - NAND gates are basic logic gates, and as such they are recognized in TTL and CMOS ICs. The standard, 4000 series, CMOS IC is the 4011, which includes four independent, two-input, NAND gates.
  - The schematic diagram shows the arrangement of NAND gates within a standard 4011 CMOS integrated circuit



NAND Gates (continued)

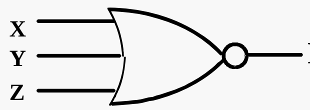
- *Universal gate* - a gate type that can implement any Boolean function.
- The NAND gate is a universal gate as shown in Figure 2-24 of the text.


NOT
X
$\overline{X}$ X
$\overline{X}$
AND
X
Y
$\overline{\overline{XY}} = XY$
OR
X
Y
$\overline{\overline{X}\,\overline{Y}} = X + Y$

- NAND usually does not have a operation symbol defined since
  - the NAND operation is not associative, and
  - we have difficulty dealing with non-associative mathematics!

NOR Gate

- The basic NOR gate has the following symbol, illustrated for three inputs:
  - OR-Invert (NOR)

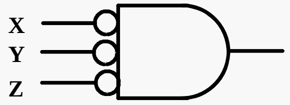


$$F(X, Y, Z) = \overline{X + Y + Z}$$

- NOR represents <u>NOT</u> <u>OR</u>, i. e., the OR function with a NOT applied. The symbol shown is an OR-Invert. The small circle ("*bubble*") represents the invert function.

NOR Gate (continued)

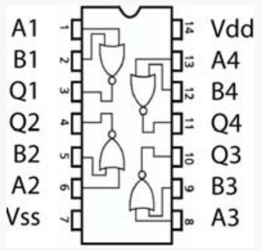
- Applying DeMorgan's Law gives Invert-AND (NOR)



- This NOR symbol is called <u>Invert-AND</u>, since inputs are inverted and then ANDed together.
- OR-Invert and Invert-AND both represent the NOR gate. Having both makes visualization of circuit function easier.
- A NOR gate with one input degenerates to an inverter.

NOR Gate (continued)

- The NOR gate is a natural implementation for some technologies other than CMOS in terms of chip area and speed.
  - NOR Gates are basic logic gates, and as such they are recognized in <u>TTL</u> and <u>CMOS</u> <u>ICs</u>. The standard, 4000 series, CMOS IC is the 4001, which includes four independent, two-input, NOR gates.
  - Diagram of a 4001 Quad NOT DIL (Dual-In-Line) format IC



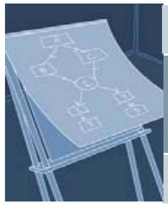
- The NOR gate is a universal gate
- NOR usually does not have a defined operation symbol since
  - the NOR operation is not associative, and
  - we have difficulty dealing with non-associative mathematics!

More Complex Gates

- The remaining complex gates are SOP or POS structures with and without an output inverter.
- The names are derived using:
  - A - AND
  - O - OR
  - I - Inverter
  - Numbers of inputs on first-level "gates" or directly to second-level "gates"

More Complex Gates (continued)

- Example: AOI - AND-OR-Invert consists of a single gate with AND functions driving an OR function which is inverted.
- Example: 2-2-1 AO has two 2-input ANDS driving an OR with one additional OR input.
- These gate types are used because:
  - the number of transistors needed is fewer than required by connecting together primitive gates
  - potentially, the circuit delay is smaller, increasing the circuit operating speed



Exclusive OR/ Exclusive NOR

- The *eXclusive OR* (*XOR*) function is an important Boolean function used extensively in logic circuits.
- The XOR function may be;
  - implemented directly as an electronic circuit (truly a gate) or
  - implemented by interconnecting other gate types (used as a convenient representation)
- The *eXclusive NOR* function is the complement of the XOR function
- By our definition, XOR and XNOR gates are complex gates.

Exclusive OR/ Exclusive NOR

- Uses for the XOR and XNORs gate include:
  - Adders / subtractors / multipliers
  - Counters / incrementers / decrementers
  - Parity generators/checkers
- Definitions
  - The ***XOR*** function is: $\mathbf{X \oplus Y = X\overline{Y} + \overline{X}Y}$
  - The ***eXclusive NOR*** (***XNOR***) function, otherwise known as *equivalence* is: $\mathbf{\overline{X \oplus Y} = X\,Y + \overline{X}\,\overline{Y}}$
- Strictly speaking, XOR and XNOR gates do no exist for more that two inputs. Instead, they are replaced by odd and even functions.

Truth Tables for XOR/XNOR

- Operator Rules: XOR XNOR

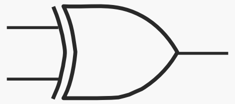
X	Y	X⊕Y
0	0	0
0	1	1
1	0	1
1	1	0

X	Y	$\overline{(X\oplus Y)}$ or $X \equiv Y$
0	0	1
0	1	0
1	0	0
1	1	1

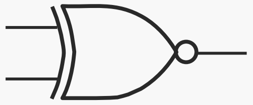
- The XOR function means:
  X OR Y, but NOT BOTH
- Why is the XNOR function also known as the *equivalence* function, denoted by the operator ≡?

Symbols For XOR and XNOR

- XOR symbol:



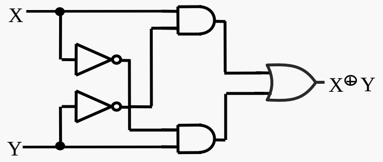
- XNOR symbol:



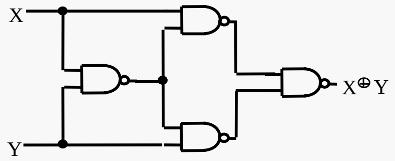
- Shaped symbols exist only for two inputs

XOR Implementations

- Two-input XOR function may be constructed with conventional gate. Two NOT gates, two AND gates and an OR gate are used
- The simple SOP implementation uses the following structure:



- A NAND only implementation is:



XOR/XNOR (Continued)

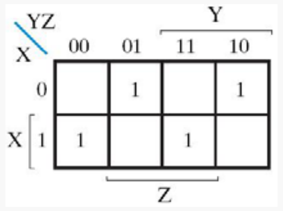
- The XOR function can be extended to 3 or more variables. For more than 2 variables, it is called an *odd function* or *modulo 2 sum* (*Mod 2 sum*), not an XOR:

$$\begin{aligned} X \oplus Y \oplus Z &= (X \oplus Y)\,\overline{Z} + (\overline{X \oplus Y})\,Z \\ &= (X\overline{Y} + \overline{X}Y)\,\overline{Z} + (XY + \overline{X}\,\overline{Y})\,Z \\ &= X\overline{Y}\,\overline{Z} + \overline{X}Y\overline{Z} + XYZ + \overline{X}\,\overline{Y}Z \\ &= \overline{X}\,\overline{Y}Z + \overline{X}Y\overline{Z} + X\overline{Y}\,\overline{Z} + XYZ \end{aligned}$$

- The complement of the odd function is the even function.
- The XOR identities:

$X \oplus 0 = X$	$X \oplus 1 = \overline{X}$
$X \oplus X = 0$	$X \oplus \overline{X} = 1$
$X \oplus \overline{Y} = \overline{X \oplus Y}$	$\overline{X} \oplus Y = \overline{X \oplus Y}$

$$X \oplus Y \oplus Z = X\overline{Y}\,\overline{Z} + \overline{X}Y\overline{Z} + \overline{X}\,\overline{Y}Z + XYZ$$



- Three exclusive OR is equal to 1, if only one variable is equal to 1 or if all three variables are equal to 1

  → The multiple-variable exclusive OR operation is defined as the *odd function*.

Odd and Even Functions

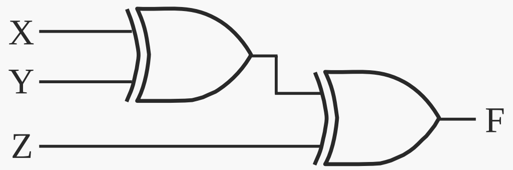
- The odd and even functions on a K-map form "*checkerboard*" patterns.
- The 1s of an odd function correspond to minterms having an index with an odd number of 1s.
- The 1s of an even function correspond to minterms having an index with an even number of 1s.
- Implementation of odd and even functions for greater than four variables as a two-level circuit is difficult, so we use "trees" made up of :
  - 2-input XOR or XNORs
  - 3- or 4-input odd or even functions

- In mathematics, ***even functions*** and ***odd functions*** are functions which satisfy particular symmetry relations, with respect to taking additive inverses. They are important in many areas of mathematical analysis, especially the theory of power series and Fourier series
  - Let $f(x)$ be a real-valued function of a real variable. Then f is even if the following equation holds for all x in the domain of f: $f(x) = f(-x)$
  - Geometrically, an even function is symmetric with respect to the y-axis, meaning that its graph remains unchanged after reflection about the y-axis.
  - Examples of even functions are $|x|$, x^2, x^4, cos(x), and cosh(x).

- A *checkerboard* (or chequerboard) is a board on which American checkers is played. It is an 8×8 board and the 64 squares are of alternating dark and light color, often red and black.

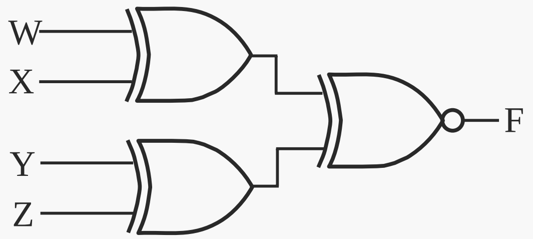
Example: Odd Function Implementation

- **Design a 3-input odd function $F = X \oplus Y \oplus Z$ with 2-input XOR gates**
- **Factoring, $F = (X \oplus Y) \oplus Z$**
- The circuit:



Example: Even Function Implementation

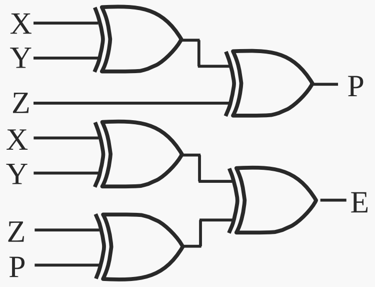
- **Design a 4-input odd function $F = W \oplus X \oplus Y \oplus Z$ with 2-input XOR and XNOR gates**
- **Factoring, $F = (W \oplus X) \oplus (Y \oplus Z)$**
- The circuit:

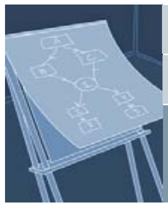


Parity Generators and Checkers

- In Chapter 1, a parity bit added to n-bit code to produce an n + 1 bit code:
  - Add odd parity bit to generate code words with even parity
  - Add even parity bit to generate code words with odd parity
  - Use odd parity circuit to check code words with even parity
  - Use even parity circuit to check code words with odd parity
- Example: n = 3. Generate even parity code words of length four with odd parity generator:
- Check even parity code words of length four with odd parity checker:
- Operation: (X,Y,Z) = (0,0,1) gives (X,Y,Z,P) = (0,0,1,1) and E = 0. If Y changes from 0 to 1 between generator and checker,

  then E = 1 indicates an error.

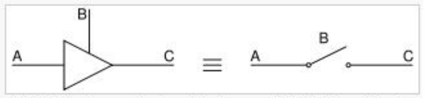




10. High-Impedance Outputs

- In electronics, *high impedance* (also known as *hi-Z*, *tri-stated*, or *floating*) is the state of an output terminal which is not currently driven by the circuit.
- In digital circuits, it means that the signal is neither driven to a logical high nor to a logical low level - hence "tri-stated". Such a signal can be seen as an open circuit (or "floating" wire) because connecting it to a (low impedance) circuit will not affect that circuit; it will instead itself be pulled to the same voltage as the actively driven output. The combined input/output pins found on many ICs are actually tri-state capable outputs which have been internally connected to inputs. This is the basis for bus-systems in computers, among many other uses.

- In digital electronics, *three-state*, tri-state, or 3-state logic allows output ports to have a value of 0, 1, or Z. A *Z* output stands for the output port being disconnected from the rest of the circuit, putting the output in a high impedance state. The intent of this state is to allow multiple circuits to share the same output line or bus without affecting each other.
- Three-state outputs are implemented in various families of digital integrated circuits such as the 7400 series of TTL gates, and often in the data and address bus lines of microprocessors.



A tristate buffer can be thought of as a switch. If *B* is on, the switch is closed. If B is off, the switch is open.

- Uses of three-state logic
  - Three-state buffers can be used to implement efficient multiplexers, especially those with large numbers of inputs.

High-Impedance Outputs

- Logic gates introduced thus far
  - have 1 and 0 output values,
  - cannot have their outputs connected together, and
  - transmit signals on connections in only one direction.
- Three-state logic adds a third logic value, *Hi-Impedance*(*Hi-Z*), giving three states: 0, 1, and Hi-Z on the outputs.
- The presence of a Hi-Z state makes a gate output as described above behave quite differently:
  - “1 and 0” become “1, 0, and Hi-Z”
  - “cannot” becomes “can,”
  - and “only one” becomes “two”

- What is a Hi-Z value?
  - The Hi-Z value behaves as an open circuit
  - This means that, looking back into the circuit, the output appears to be disconnected.
  - It is as if a switch between the internal circuitry and the output has been opened.
- Hi-Z may appear on the output of any gate, but we restrict gates to:
  - a 3-state buffer, or
  - Optional: a transmission gate (See Reading Supplement: More on CMOS Circuit-Level Design),

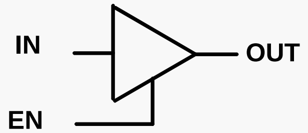
each of which has one data input and one control input.

The 3-State Buffer

- For the symbol and truth table, IN is the data input, and EN, the control input.
- For EN = 0, regardless of the value on IN (denoted by X), the output value is Hi-Z.
- For EN = 1, the output value follows the input value.
- Variations:
  - Data input, IN, can be inverted
  - Control input, EN, can be inverted

  by addition of "bubbles" to signals.

Symbol



Truth Table

EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

Resolving 3-State Values on a Connection

- Connection of two 3-state buffer outputs, B1 and B0, to a wire, OUT
- Assumption: Buffer data inputs can take on any combination of values 0 and 1
- Resulting Rule: At least one buffer output value must be Hi-Z. Why?
- How many valid buffer output combinations exist?
- What is the rule for *n* 3-state buffers connected to wire, OUT?
- How many valid buffer output combinations exist?

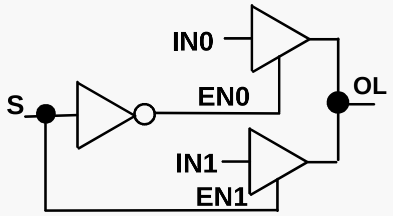
Resolution Table

B1	B0	OUT
0	Hi-Z	0
1	Hi-Z	1
Hi-Z	0	0
Hi-Z	1	1
Hi-Z	Hi-Z	Hi-Z

3-State Logic Circuit

- Data Selection Function: If s = 0, OL = IN0, else OL = IN1
- Performing data selection with 3-state buffers:

EN0	IN0	EN1	IN1	OL
0	X	1	0	0
0	X	1	1	1
1	0	0	X	0
1	1	0	X	1
0	X	0	X	X



- Since EN0 = $\overline{S}$ and EN1 = S, one of the two buffer outputs is always Hi-Z plus the last row of the table never occurs.

Terms of Use