Hyperiondev

LESSON

Logical Problem Solving & Error Detection techniques

Visit our website

Introduction

WELCOME TO THE INTRODUCTION TO THE JDBC TASK!

In this lesson, we are going to dive into the topic of logical problem solving using logic gates, truth tables, boolean algebra, and Karnaugh maps. We will also look at error detection and correction methods. At the end, as usual, there is a task for you to put into practice what you have learned.

INTENDED LEARNING OUTCOMES

You will know that you have succeeded at this task if, by the time you complete it, you are able to:

- Describe different approaches to problem solving
- Use logical operators in descriptions of rules and relationships in a problem situation
- Simplify Boolean expressions with Boolean algebra and Karnaugh maps
- Describe the basic concepts of error detection

Refer to the self-assessment table provided after the compulsory task near the end of the document for a checklist to assess your own degree of achievement of each of these learning outcomes. Try to gauge your own learning and address any weak areas before submitting your task, which will become part of your formative assessment portfolio.

There are a number of concepts you need to understand in order to be able to demonstrate logical problem solving and error detection techniques. These are explained below, drawing from the Bella Visage Learner Guide for the *National Certificate: Information Technology (Systems Development)* (licenced for use by HyperionDev).

This lesson addresses Unit Standard 115367: Demonstrate logical problem solving and error detection techniques.

Specific Outcomes:

1. **SO1: Describe different approaches to problem solving. (Range: Top-down; Bottom-up; Systems Approach (at least two))**
2. **SO2: Use logical operators in descriptions of rules and relationships in a problem situation.**
3. **SO3: Simplify Boolean expressions with Boolean algebra and Karnaugh maps. (Range: Use up to 4 variables.)**
4. **SO4:Describe the basic concepts of error detection.**

APPROACHES TO LOGICAL PROBLEM SOLVING AND ERROR DETECTION (SO1)

In this section, you're going to be introduced to different problem solving techniques and identify situations where specific problem solving techniques would be more suitable than others. We're going to consider the top-down problem solving approach with relatable real life problems, and facilitate break down problems pictorially.

Top-down and **bottom-up** problem-solving strategies are both strategies of information processing and knowledge ordering, used in a variety of fields including the humanities and sciences, management and organisation, and software development in IT. In practice, they can be seen as a style of thinking and ordering steps.

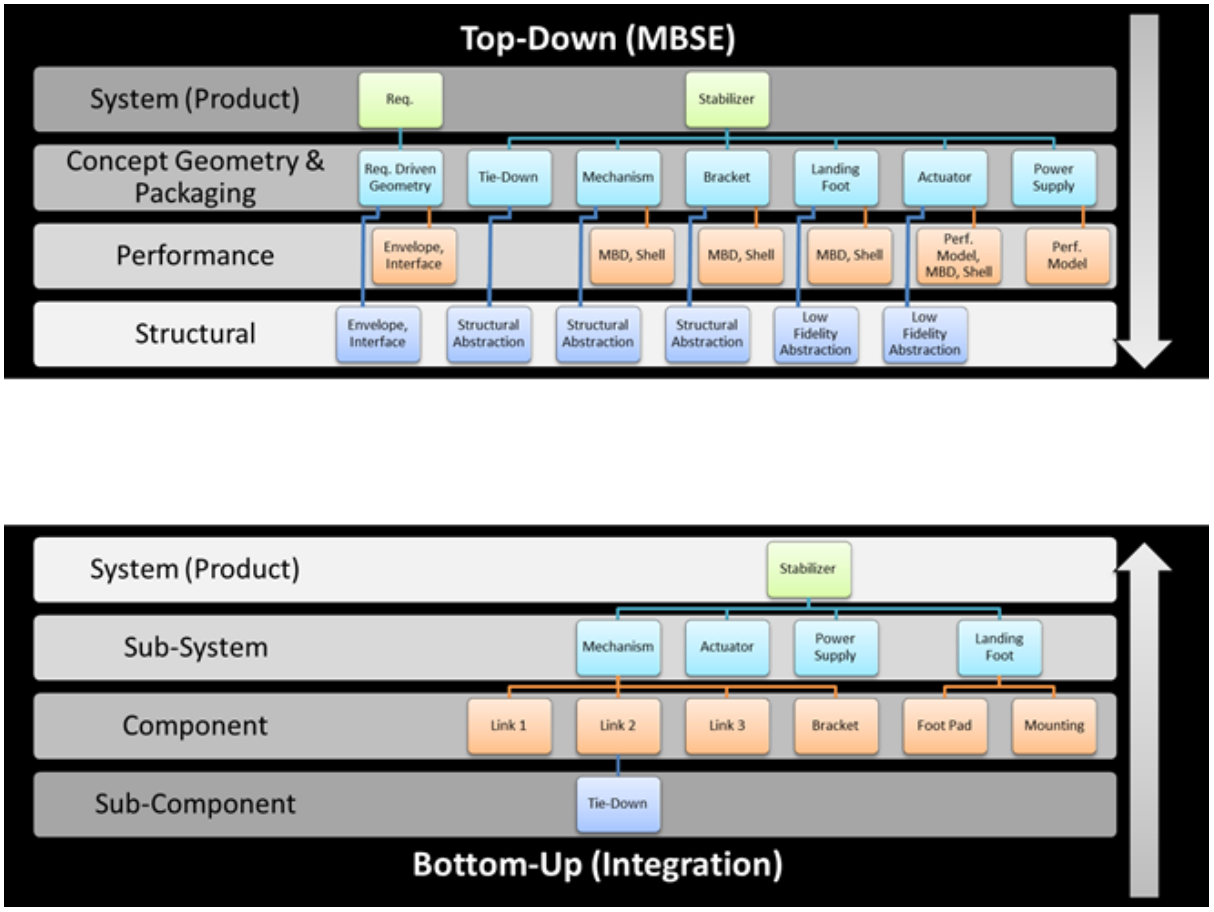
A **top-down approach** (also known as stepwise design and in some cases used as a synonym of decomposition) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", a concept used to make problems easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model. Essentially, the top down approach starts with the big picture, and breaks down from there into smaller segments.

A **bottom-up approach** is the piecing together of systems to give rise to more complex systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of information processing aligned for things like using incoming data from the environment to form a perception. Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output). In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies'' may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

In the software development process, the top-down and bottom-up approaches play a key role.

Top-down approaches emphasise planning and a complete understanding of the system. It is inherent that no coding can begin until a sufficient level of detail has been reached in the design of at least some part of the system. Top-down approaches are implemented by attaching the stubs in place of the module. This, however, delays testing of the ultimate functional units of a system until significant design is complete.

Bottom-up approaches emphasise coding and early testing, which can begin as soon as the first module has been specified. This approach, however, runs the risk that modules may be coded without having a clear idea of how they link to other parts of the system, and that such linking may not be as easy as first thought. Reusability of code is one of the main benefits of the bottom-up approach



Top-Down vs. Bottom-Up approaches

LOGICAL PROBLEM SOLVING USING LOGIC GATES (SO2)

In this lesson, you're going to use logical operators in drawing truth tables, and combine different operators to form Boolean expressions by setting up truth tables We will .apply this to examples of problem situations where a specific operator can be used, and consider which of the operators should be used to represent given situations.

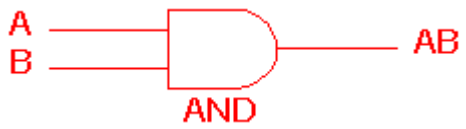
TRUTH TABLES

Truth tables are used to help show the function of a logic gate. If you are unsure about truth tables and need guidance on how to go about drawing them for individual gates or logic circuits then use the truth table section link. We will look at Truth tables again in the next section on Boolean algebra.

Logic Gates

Digital systems are said to be constructed by using logic gates. These gates are the AND, OR, NOT, NAND, NOR, EXOR and EXNOR gates. The basic operations are described below with the aid of truth tables.

AND gate



2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

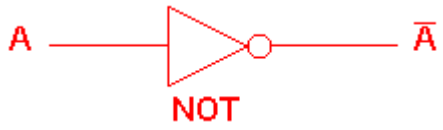
The AND gate is an electronic circuit that gives an output of TRUE, i.e. 1., only if **all** its inputs are TRUE (1). A dot (.) is used to show the AND operation i.e. A**.**B, as per the 3rd column in the truth table above). Bear in mind that this dot is sometimes omitted i.e. AB .

OR gate

2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

The OR gate is an electronic circuit that gives an output of TRUE (1) if **one or more** of its inputs are TRUE. A plus (+) is used to show the OR operation, as per the 3rd column in the truth table above).

NOT gate



NOT gate	
A	$\overline{A}$
0	1
1	0

The NOT gate is an electronic circuit that outputs an inverted version of the input. (it is also known as an **inverter**). If the input variable is **A**, the inverted output is expressed as **NOT A**. This is also shown as A' (say "A prime"), or A with a bar over the top, as per the 3rd column in the truth table above.

NAND gate

2 Input NAND gate		
A	B	$\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

This is a **NOT-AND** gate which is equal to an **AND gate followed by a NOT gate**. The outputs of all NAND gates are TRUE (1) if any of the inputs are FALSE (0). The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

NOR gate

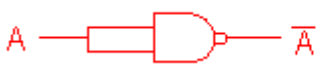
2 Input NOR gate		
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

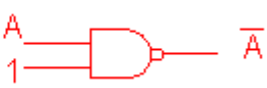
This is a **NOT-OR** gate which is equal to an **OR gate followed by a NOT gate**. The outputs of all NOR gates are FALSE (0)if any of the inputs are TRUE (1).

The symbol is an OR gate with a small circle on the output, where once again the small circle represents inversion.

Making NOT gates out of NAND and NOR gates

The diagrams below show two ways that the NAND logic gate can be configured to produce a NOT gate. This can also be done using NOR logic gates in the same way.



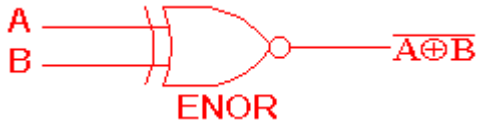


EXOR gate

2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

This **Exclusive-OR** (EXOR or XOR) gate is a circuit which will give an output of TRUE (1) if **either, but not both**, of its two inputs are TRUE (1). An encircled plus sign (⊕) is used to show the EOR operation.

EXNOR gate



2 Input EXNOR gate		
A	B	$\overline{A⊕B}$
0	0	1
0	1	0
1	0	0
1	1	1

This **Exclusive-NOR** (EXNOR or XNOR) gate circuit does the opposite to the EXOR gate. It will give an output of FALSE (0) if **either, but not both**, of its two inputs are TRUE (1). The symbol is an EXOR gate with a small circle on the output. As usual the small circle represents inversion.

The NAND and NOR gates are called *universal functions*, as with either one the AND, OR and NOT functions can be generated.

Note: A function in *sum of products* form can be implemented using NAND gates by replacing all AND and OR gates by NAND gates. A function in product of sums form can be implemented using NOR gates by replacing all AND and OR gates by NOR gates.

Table 1: Logic gate symbols

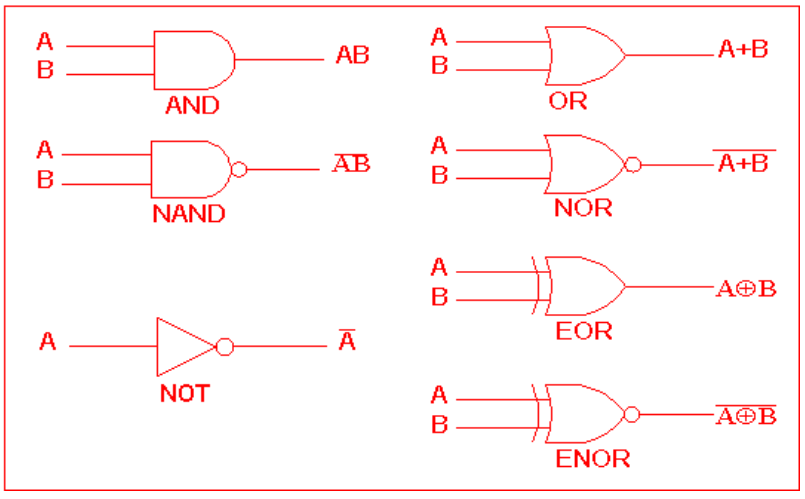


Table 2 is a summary truth table of the input/output combinations for the NOT gate together with all possible input/output combinations for the other gate functions. Also note that a truth table with 'n' inputs has 2^n rows. You can compare the outputs of different gates.

Table 2: Logic gates representation using the Truth table

NOT gate	
A	$\overline{A}$
0	1
1	0

INPUTS		OUTPUTS					
A	B	AND	NAND	OR	NOR	EXOR	EXNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

Example

A **NAND** gate can be used as a NOT gate using either of the following wiring configurations.

(You can check this out using a truth table as well.)

BOOLEAN EXPRESSIONS AND THEIR SIMPLIFICATION *(SO3)*

In this lesson, you're going to consider the rules of Boolean algebra, and use these to simplify basic expressions. You're also going to be introduced to Karnaugh maps to represent Boolean expressions, and practice simplifying these by writing down the simplified expression from the map.

LAWS OF BOOLEAN ALGEBRA

The following is a set of simplified ways to remember the laws of Boolean algebra:

1. Anything ANDed with a 0 is equal to 0 → A.0 = 0
2. Anything ANDed with a 1 is equal to itself →. A. 1 = A
3. Anything ORed with a 0 is equal to itself. → A + 0 = A
4. Anything ORed with a 1 is equal to 1 → A + 1 = 1
5. Anything ANDed with itself is equal to itself → A. A = A
6. Anything ORed with itself is equal to itself → A + A = A
7. Anything ANDed with its own complement equals 0 → A.$\bar{A}$ = 0
8. Anything ORed with its own complement equals 1 → A+$\bar{A}$ = 1
9. Anything complemented twice is equal to the original → $\bar{\bar{A}}$ = A
10. The two variable rule → A +$\bar{A}$B = A + B
11. De Morgan's theorem:

$\overline{A+B} = \bar{A}.\bar{B}$	NOT(A OR B)=(NOT A) AND (NOT B)
$\overline{A.B} = \bar{A}+\bar{B}$	NOT (A AND B) = (NOT A) OR (NOT B)

Source: https://grace.bluegrass.kctcs.edu/~kdunn0001/files/Simplification/4_Simplification_print.html

BOOLEAN EXPRESSIONS/FUNCTIONS

Boolean algebra deals with binary variables and logical operation. A **Boolean Function** is described by an algebraic expression called a Boolean expression which consists of binary variables, the constants 0 and 1, and the logical operation symbols. Consider the following example:

$F(A, B, C, D) = A + \overline{BC} + ADC$ Equation No. 1

Boolean Function — Boolean Expression

Here the left side of the equation represents the output Y. So we can state equation no. 1:

$Y = A + \overline{BC} + ADC$

Truth Table Formation

A truth table represents a table having all combinations of inputs and their corresponding result.

It is possible to convert the switching equation into a truth table. For example consider the following switching equation.

$F(A, B, C) = A + BC$

The output will be TRUE (1) if A = 1 or BC = 1 or both are 1. The truth table for this equation is shown by Table (a). The number of rows in the truth table is 2^n where n is the number of input variables (n=3 for the given equation). Hence there are $2^3 = 8$ possible input combination of inputs.

Inputs			Output
A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Methods to simplify the Boolean function

The methods used for simplifying the Boolean function are as follows.

- Karnaugh-map or K-map.
- NAND gate method

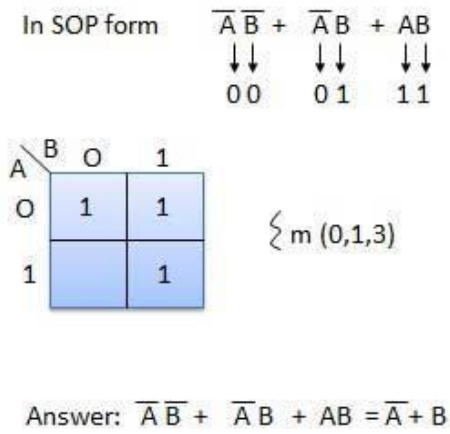
Karnaugh-map or K-map

The Boolean theorems and De-Morgan's theorems are useful in manipulating the logical expression. We can realise the logical expression using gates. The number of logic gates required for the realisation of a logical expression should be reduced to the minimum possible value by using the K-map method. This method can be used in two ways, the sum of products form and the product of sums form.

SUM OF PRODUCTS (SOP) FORM

The SOP form is in the form of the sum of three terms AB, AC, BC with each individual term being a product of two variables, e.g. A.B or A.C etc. Therefore, the expression is known as an expression in SOP form. The sum and products in SOP form are not the actual additions or multiplications. In fact they are the OR and AND functions. Remember that 0 is the value given for barred (FALSE) variables, and 1 is the value given for the unbarred (TRUE) variables. The SOP form is indicated by Σ.

An example of SOP form is as follows:

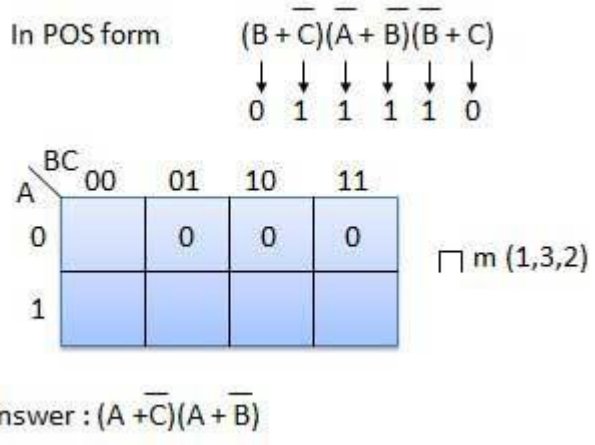


Answer: $\overline{A}\,\overline{B} + \overline{A}B + AB = \overline{A} + B$

PRODUCT OF SUMS (POS) FORM

The SOP form is the product of three terms (A+B), (B+C) and (A+C) with each term being the sum of two variables. Such expressions are said to be in the product of sums (POS) form. Remember that 0 is the value given for barred (FALSE) variables, and 1 is the value given for the unbarred (TRUE) variables. The POS form is indicated by $\prod$.

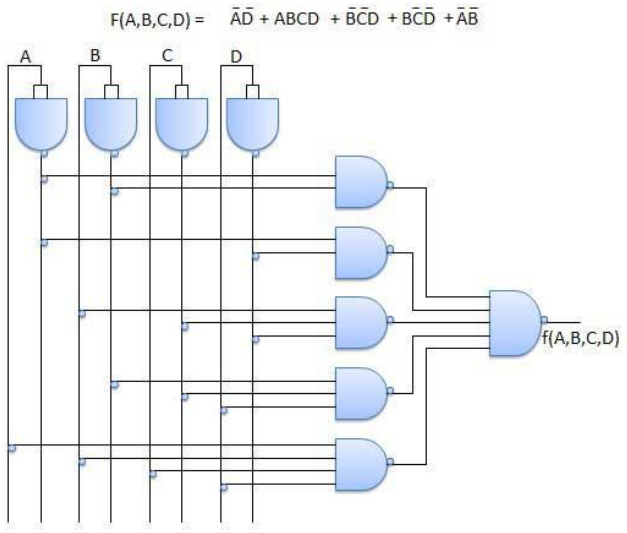
An example of POS form is as follows.



Answer : $(A + \overline{C})(\overline{A} + \overline{B})$

NAND gates Realization

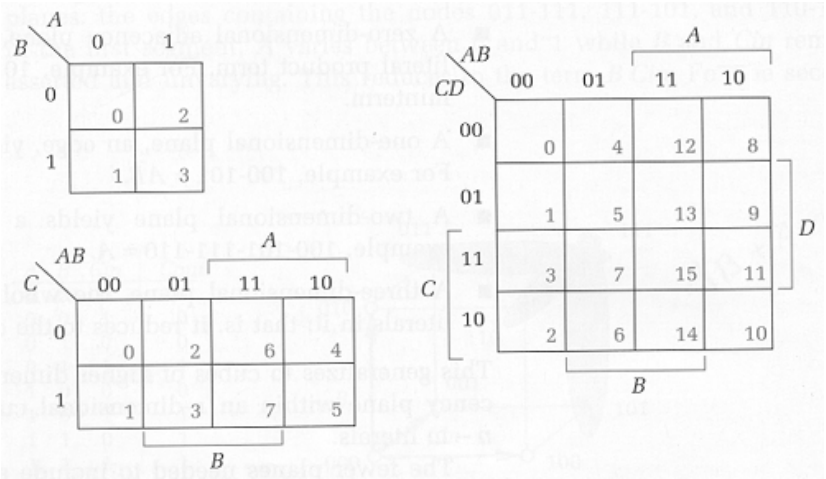
NAND gates can be used to simplify Boolean functions as shown in the example below.



KARNAUGH MAPS

A Karnaugh Map, or K-Map, is a grid-like representation of a truth table. It is really just another way of presenting a truth table, but the mode of presentation gives more insight. A Karnaugh map has zero and one entries at different positions. Each position in a grid corresponds to a truth table entry.

A K-map shows the value of a function for every combination of input values just like a truth table, but a K-map spatially arranges the values so it is easy to find common terms that can be factored out. The image below shows the form of a 2, 3, and 4 variable K-map.



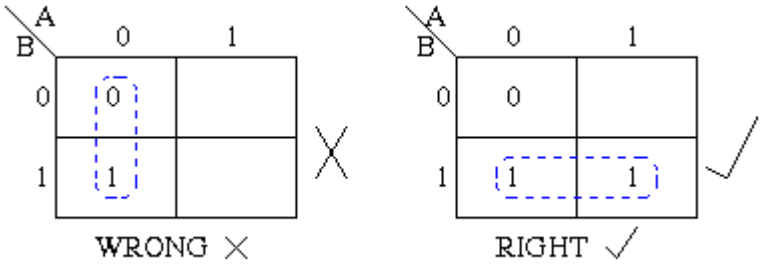
How Can a Karnaugh Map Help?

At first, it might seem that the Karnaugh Map is just another way of presenting the information in a truth table. In one way that's true. However, any time you have the opportunity to use another way of looking at a problem, this may offer advantages. In the case of the Karnaugh Map the advantage is that the Karnaugh Map is designed to present the information in a way that allows easy grouping of terms that can be combined.

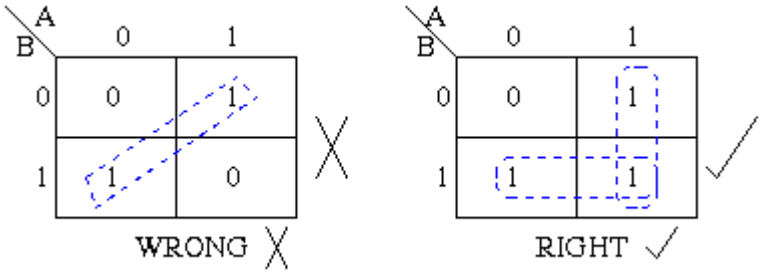
Karnaugh Maps - Rules of Simplification

The Karnaugh map uses the following rules for the simplification of expressions by *grouping* together adjacent cells containing *ones*.

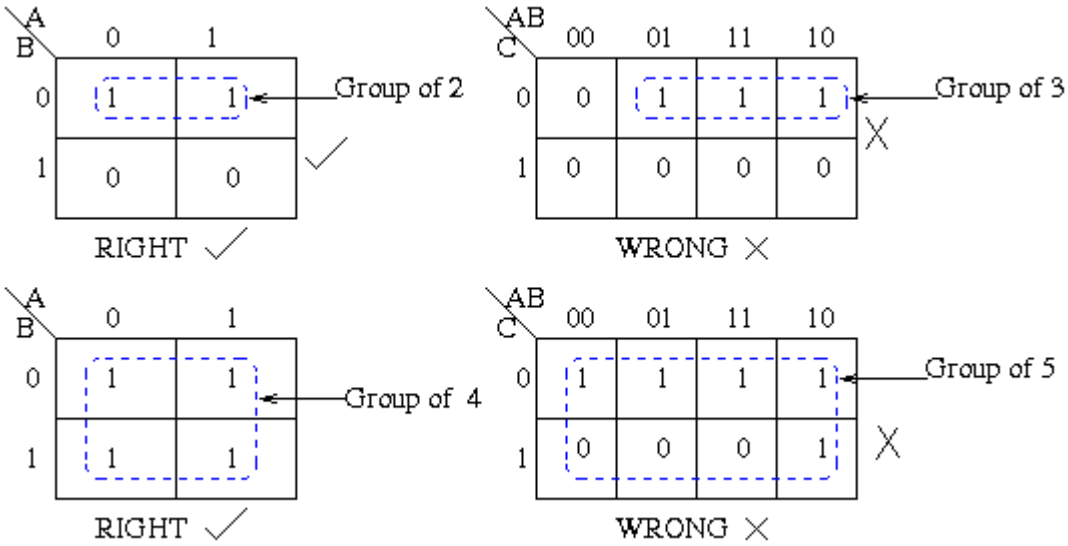
- **Groups may not include any cell containing a zero**



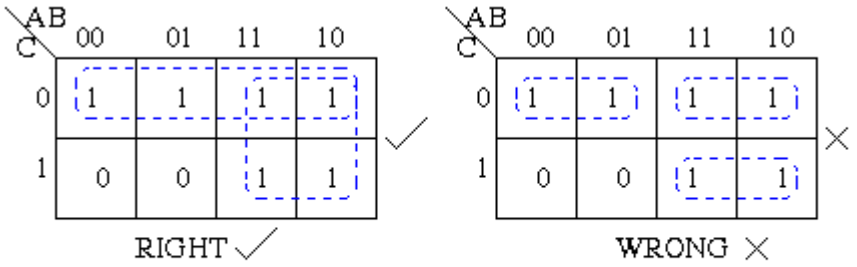
- **Groups may be horizontal or vertical, but not diagonal.**



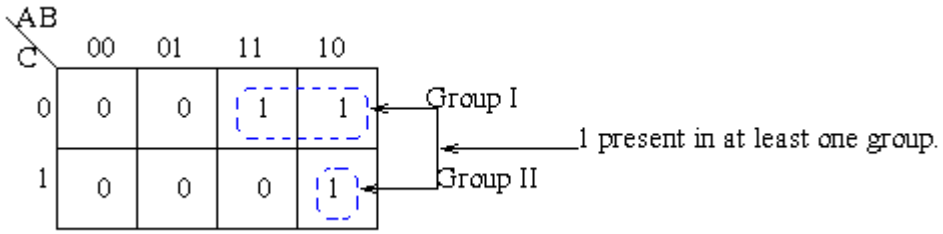
- **Groups must contain 1, 2, 4, 8, or in general 2^n cells.**
  That is if n = 1, a group will contain two 1's since $2^1 = 2$.
  If n = 2, a group will contain four 1's since $2^2 = 4$.



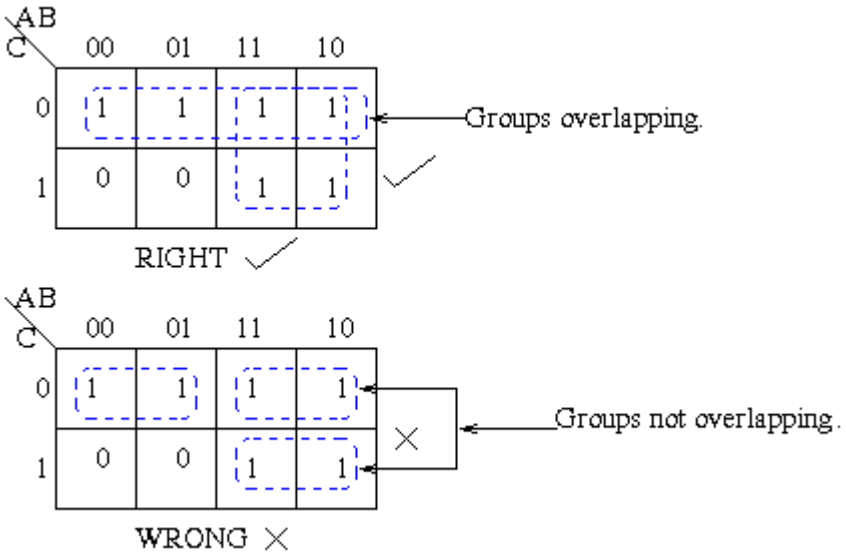
- **Each group should be as large as possible.**



- **Each cell containing a *one* must be in at least one group.**

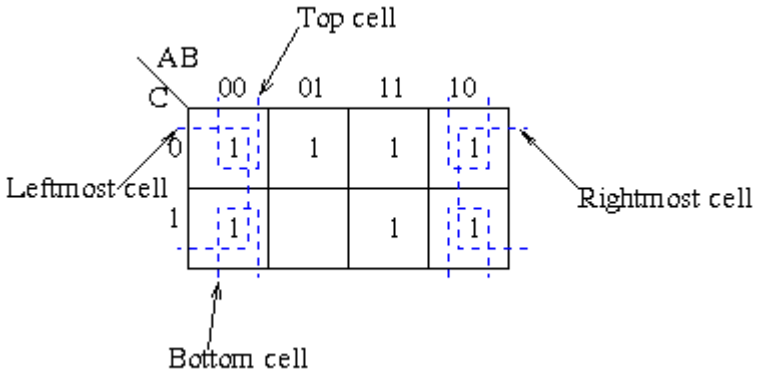


- **Groups may overlap.**

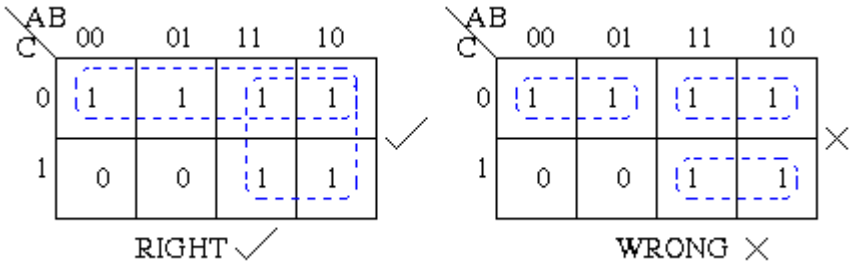


- **Groups may wrap around the table. The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be**

grouped with the bottom cell.



- **There should be as few groups as possible, as long as this does not contradict any of the previous rules.**



K-map rules summary

1. No zeros allowed.
2. No diagonals.
3. Only 2^n (powers-of-2 numbers) of cells in each group .
4. Groups should be as large as possible.
5. Every 1 must be in at least one group.
6. Overlapping is allowed.
7. Wraparound is allowed.
8. Fewest number of groups possible.

BASIC CONCEPTS OF ERROR DETECTION (SO4)

In this lesson, you're going to look at how to identify common causes of errors, error isolation techniques, and various testing techniques.

COMMON CAUSES OF ERRORS

In computer programming, a logic error is a bug in a program that causes it to operate incorrectly, but not to terminate abnormally (or crash). A logic error produces unintended or undesired output or other behavior, although it may not immediately be recognized as such.

Logic errors occur in both compiled and interpreted languages. Unlike a program with a syntax error, a program with a logic error is a valid program in the language, though it does not behave as intended. The only clue to the existence of logic errors is the production of wrong solutions.

Common causes

The mistake could be a simple error in a statement (for example, an incorrect formula), an error in an algorithm, or even the wrong algorithm selected. There are also numerous other causes, such as incorrect type casting, variable scoping, missing code fragments and wrong problem (or requirement) interpretation.

Debugging logic errors

One of the ways to find these type of errors is to output the program's variables to a file or on the screen in order to define the error's location in code. Although this will not work in all cases, for example when calling the wrong subroutine, it is the easiest way to find the problem if the program uses the incorrect results of a bad mathematical calculation.

Truncation errors in numerical integration are of two kinds:

local truncation errors – the error caused by one iteration, and global truncation errors – the cumulative error caused by many iterations.

ERROR ISOLATION TECHNIQUES

What is Error Correction and Detection?

Error detection and correction has great practical importance in maintaining data (information) integrity across noisy Communication Networks channels and less-than-reliable storage media.

Error Correction: Send additional information so incorrect data can be corrected and accepted. Error correction is the additional ability to reconstruct the original, error-free data.

There are two basic ways to design the channel code and protocol for an error correcting system :

- **Automatic Repeat-Request (ARQ) :** The transmitter sends the data and also an error detection code, which the receiver uses to check for errors, and

request retransmission of erroneous data. In many cases, the request is implicit; the receiver sends an acknowledgement (ACK) of correctly received data, and the transmitter re-sends anything not acknowledged within a reasonable period of time.

- **Forward Error Correction (FEC) :** The transmitter encodes the data with an error-correcting code (ECC) and sends the coded message. The receiver never sends any messages back to the transmitter. The receiver decodes what it receives into the "most likely" data. The codes are designed so that it would take an "unreasonable" amount of noise to trick the receiver into misinterpreting the data.

Error Detection: Send additional information so incorrect data can be detected and rejected. Error detection is the ability to detect the presence of errors caused by noise or other impairments during transmission from the transmitter to the receiver.

VARIOUS TESTING TECHNIQUES

The box approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

White-Box testing

White-box testing (also known as **clear box testing**, **glass box testing**, **transparent box testing** and **structural testing**) tests internal structures or workings of a program, as opposed to the functionality exposed to the end-user. In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT).

While white-box testing can be applied at the unit, integration, and system levels of the software testing process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

Techniques used in white-box testing include:

- API testing (application programming interface) – testing of the application using public and private APIs
- Code coverage – creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)

- Fault injection methods – intentionally introducing faults to gauge the efficacy of testing strategies
- Mutation testing methods
- Static testing methods

Black-box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation. The testers are only aware of what the software is supposed to do, not how it does it. Black-box testing methods include: equivalence partitioning, boundary, all-pairs testing, state transition tables, decision table testing, fuzz testing, model-based testing, use case testing, exploratory and specification-based testing.

Compulsory Task 1

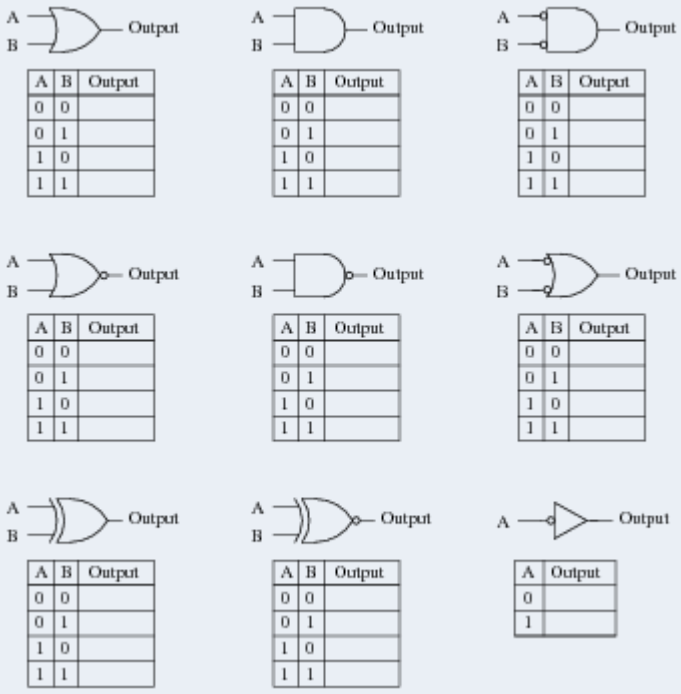
- Describe two different approaches to problem solving.

  (2 x 2 = 4)

- Answer the following and say what the theorem that governs this relationship is called.
  - NOT(A OR B)=
  - NOT (A AND B) =
  - The theorem is:

  (1+1+1 = 3)

- Identify each of these logic gates by name, and complete their respective truth tables



(9+9 = 18)

Self-assessment table

Intended learning outcome (SAQA US 115367)	Assess your own proficiency level for each intended learning outcome. Check each box when the statement becomes true- **I can successfully**:	
Describe different approaches to problem solving	Explain and apply the top- down approach	Explain and apply the bottom-up approach

(SO 1)				
Use logical operators in descriptions of rules and relationships in a problem situation (SO 2)	Understand and be able to create and solve simple truth tables		Understand and be able to use logic gates, and relate these to their truth tables	
Simplify Boolean expressions with Boolean algebra and Karnaugh maps (SO 3)	Use and understand the laws of Boolean algebra	Simplify Boolean expressions	Create and solve Karnaugh maps and associated truth tables	
Describe the basic concepts of error detection (SO 4)	Understand common causes of logic errors	Know how to debug logic errors	Understand use black-box testing	Understand and use white-box testing

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCES

Ciubotaru, B., & Muntean, G. (2013). *Advanced network programming – Principles and techniques: Network application programming with Java* (p. 136). London: Springer Science & Business Media.

Hock-Chuan, C. (2020a). Java Tutorial - An Introduction to Java Database Programming (JDBC) by Examples with MySQL. Retrieved 19 February 2020, from **https://www3.ntu.edu.sg/HOME/EHCHUA/PROGRAMMING/java/JDBC_Basic.html**

Hock-Chuan, C. (2020b). An Introduction to Java Database Programming (JDBC) by Examples. Retrieved 20 February 2020, from **https://www3.ntu.edu.sg/HOME/EHCHUA/PROGRAMMING/java/JDBC_Basic2.html**

Liang, Y. (2011). *Introduction to Java programming: Comprehensive version* (8th ed., pp. 1273-1308). New Jersey: Prentice Hall.

Author unknown. *Electronics*Tutorials. Retrieved 12 October 2021, from **https://www.electronics-tutorials.ws/boolean/bool_6.html**