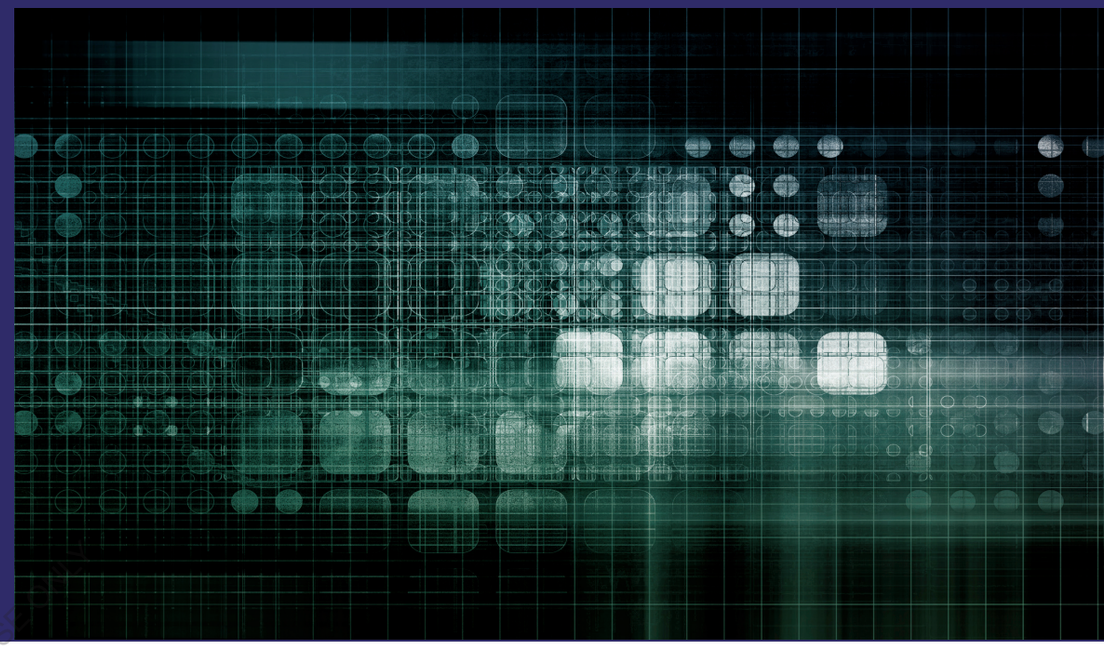
VHDL is a description language for digital electronic circuits that is used in die rent levels of abstraction.The VHDL acronym stands for VHSIC (Very High-Speed Integrated Circuits) Hardware Description Language. This means that VHDL can be used to accelerate the design process. It is very important to point out that VHDL is NOT a programming language. Therefore, knowing its syntax does not necessarily mean being able to designing digital circuits with it. VHDL is an HDL (Hardware Description Language), which allows describing both asynchronous and synchronous circuits.
For this purpose, we need to read this book through which VHDL is learned and programmed.
This book is useful for those who want to start working in VHDL and learn it.

Elaf A.Saeed is a systems and control engineer at the University of Al-Nahrain, college of information engineering, Iraq. she works in the fields of control, embedded systems, web design and has a talent of programming. Elaf was ranking as the first student of her stage for all years in B.Sc .

978-620-2-53100-9

Basics VHDL Lab

A.Saeed



Elaf A.Saeed

Basics VHDL Lab

Basics Lab for Learning VHDL Learning with Experiments

LAP LAMBERT Academic Publishing

Elaf A.Saeed

Basics VHDL Lab

Elaf A.Saeed

Basics VHDL Lab

Basics Lab for Learning VHDL Learning with Experiments

LAP LAMBERT Academic Publishing

Imprint
Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this work is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Cover image: www.ingimage.com

Publisher:
LAP LAMBERT Academic Publishing
is a trademark of
International Book Market Service Ltd., member of OmniScriptum Publishing Group
17 Meldrum Street, Beau Bassin 71504, Mauritius

Printed at: see last page
ISBN: 978-620-2-53100-9

To My Parents and To Our Family Who Made This Accomplished Possible

Table of Contents

Abbreviation Table

Abbreviation	Meaning
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
VHDL	Very High-Speed Integration Circuit HDL
UUT	Unit Under Test
RTL	Resistor-transistor switches
Ex-Or	Exclusive-OR
LSB	Least Significant Bit
MSB	Most Significant Bit
MUX	Multiplexer
Demux	Demultiplexer
IC	Integrated Circuit
OE	Output Enable
FSM	Finite State Machine
FF	Flip-Flop
LFSR	Linear-Feedback Shift Register
NCC	Nonconventional Counter

INTRODUCTION TO BOOK

The purpose of this book is to provide students and young engineers with a guide to help them develop the skills necessary to be able to use VHDL for introductory and intermediate level digital design. These skills will also give you the ability and the confidence to continue on with VHDL-based digital design. In this way, you will also take steps toward developing the skills required to implement more advanced digital design systems.

This book helps readers create good VHDL descriptions and simulate VHDL designs. The book is divided into four chapters, covering aspects ranging from the very basics of VHDL syntax and the module concept, to VHDL logic circuit implementations.

In the first chapter, the FPGA devices and VHDL language are explained in detail. The second chapter explains the software tools that are used in the design the VHDL code. In the third chapter, offer information on the simulation of VHDL programs and demonstrate how to define data types other than the standard ones available in VHDL libraries. The four chapter, a set of experiments has been made in which to learn how to program using VHDL.

The book offers discussion at the end of each chapter, inviting readers to learn VHDL by doing it and writing good code.

CHAPTER ONE

INTRODUCTION TO FPGA AND VHDL

Elaf A.Saeed

CHAPTER ONE
INTRODUCTION TO FPGA AND VHDL

Old Ways of Implementing Digital Circuits

- **Discrete Logic**: based on gates or small packages containing small digital building blocks, as shown in figure 1.1.

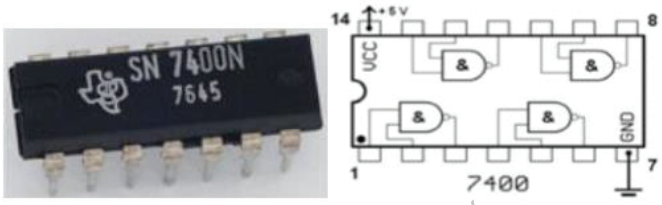


Figure 1.1: NAND Gate SN7400N

- De Morgan's theorem – theoretically we only need 2input NAND or NOR gates to build anything.
- Tedious, expensive, slow, prone to wiring errors.

In figure 1.2 that shown the Digital Circuit Design example.

Figure 1.2: Digital Circuit

Early Integrated Circuits based on Gate Arrays

- **Rows of gates**: often identical in structure
- Connected to form customer specific circuits.
- Can be **full-custom** (completely customized).

- Can be **semi-custom** (it is made with standard rows of gates but leaving the gates unconnected)
- **ASIC.**

In figure 2.3 that shown the IC based on gate array structure.

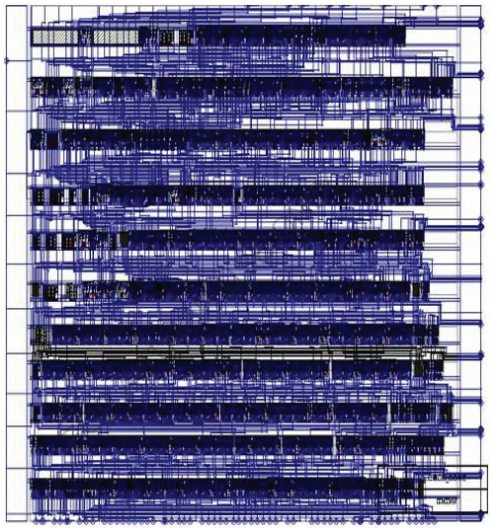


Figure 1.3: IC Based on Gate Array Structure

Modern Digital Design – Full Custom IC

- Intel Core i7.
- > 3-4 billion trans.
- you can also customize everything – each transistor and each wiring connected in a full-custom manner.
- Designing such a circuit is **very expensive**, **highly risky**, and **once designed**, it cannot be changed easily.
- This drives the rise of the Field Programmable Gate Array.

In figure 1.4 that shown the modern digital design.

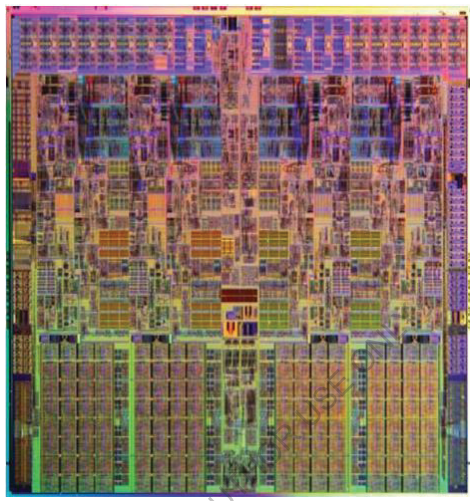


Figure 1.4: Modern Digital Design

Field Programmable Gate Arrays (FPGAs)

• Combining idea from **Programmable Logic Devices** & **Gate Arrays**.

• With Programmable Logic Device, the user can program what the logic gate does (be it a NAND or NOR or some form of **SUM-of PRODUCT** implementation) or an adder, you as a user, can "program" the chip to perform that logic function. In figure 1.5 that shown the field programmable Gate Array (FPGA).

CHAPTER ONE
INTRODUCTION TO FPGA AND VHDL

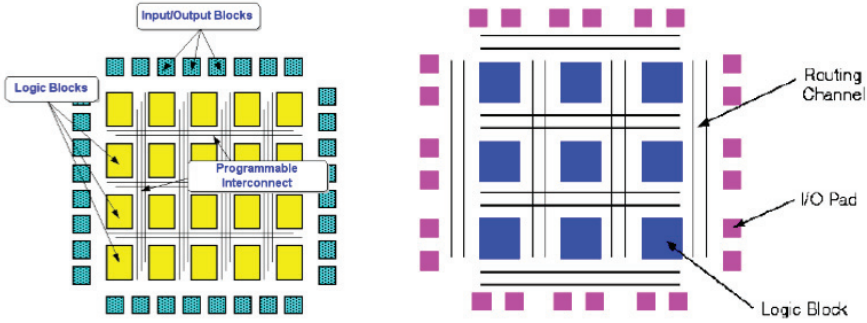


Figure 1.5: Field Programmable Logic Gate (FPGA)

• Now we can add another layer of **user programmability** – you can program how these logic gates are connected together! In that way, we have a general programmable logic chip.

• Unlike the **microprocessor** where the program is just the instruction to fix digital hardware, here you can program the hardware itself!

FPGA Configuration

• Programming an **FPGA** is called "**configuration**". In programming a computer or microprocessor, we send to the computer instruction codes as **'1'**s and **'0'**s. These are interpreted (or decoded) by the computer which will follow the instruction to perform tasks.

• The **microprocessor** needs to be fed these program codes continuously for it to function. In **FPGAs**, you only need to configure the chip ONCE on power-up.

• You download to the chip a **BITSTREAM** (also bits in '1's and '0's), which determines the logic functions performed by the Logic Elements, and the interconnecting switches in order to connect the different LEs together to make up your circuit.

• Once the bitstream is received, the FPGA no longer needs to read the 1's and 0's again, very unlike a microprocessor which has to continually decoding the machine instructions.

• That's why we say that we configure an FPGA (instead of programming an FPGA, although the two words are used interchangeably).

FPGA Design Flow

In figure 1.6 that shown the FPGA design flow.

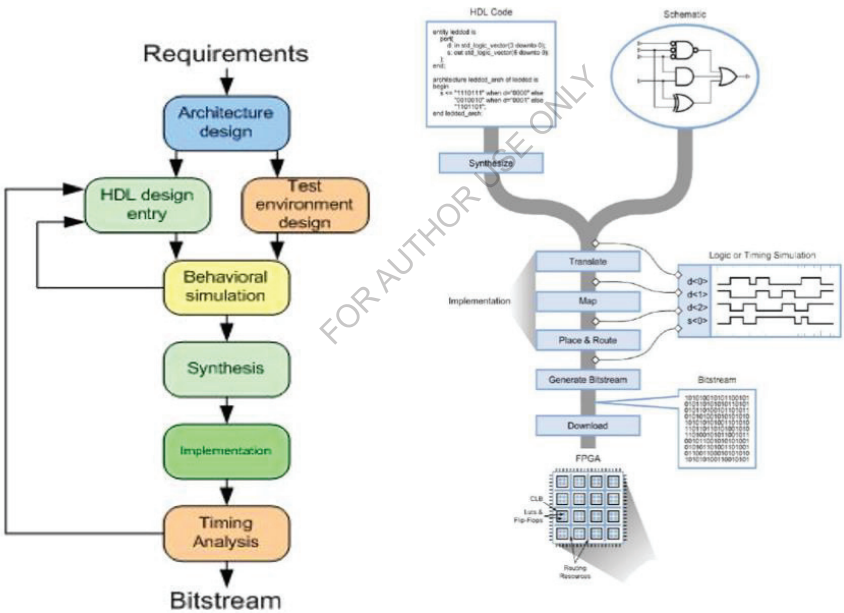


Figure 1.6: FPGA Design Flow

FPGA Programming Languages

• **HDL** Language

- VHDL

- Verilog

• Schematic Design

• **VHDL** = VHSIC Hardware Description Language "

VHSIC = Very High-Speed Integrated Circuits.

• Describes behavior of electronic system

• Based on this description, the system will be implemented.

What is HDL?

HDL stands for **Hardware Description Language**. It is a **programming language** that is used to **describe, simulate,** and **create** hardware like digital circuits (ICS). HDL is mainly used to discover the faults in the design before implementing it in the hardware. The main advantage of HDLs is that it provides **flexible modeling capabilities** and can express the large complex designs ($>10^7$ gates).

there are many HDLs available in the market, but VHDL and Verilog are the most popular HDLs.

What is VHDL?

VHDL stands for **Very High-Speed Integration Circuit HDL** (**Hardware Description Language**). It is an **IEEE** (**Institute of Electrical and Electronics Engineers**) standard hardware description language that is used to describe and simulate the behavior of complex digital circuits.

The most popular examples of VHDL are **Odd Parity Generator**, **Pulse Generator**, **Priority Encoder**, **Behavioral Model for 16 words**, **8bit RAM**, etc.

VHDL supports the following features:

- Design methodologies and their features.
- Sequential and concurrent activities.

- Design exchange.
- Standardization.
- Documentation.
- Readability.
- Large-scale design.
- A wide range of descriptive capability.

What is Verilog?

Verilog is also an **HDL** (Hardware Description Languages) for describing electronic circuits and systems. It is used in both **hardware simulation** and **synthesis**.

The most popular examples of Verilog are **network switch**, **a microprocessor**, **a memory**, **a simple flip-flop**, etc.

Difference between VHDL and Verilog

In table 1.1 that shown the different between VHDL and Verilog.

Table 1.1: The Different between VHDL and Verilog

VHDL	Verilog
It allows the user to define data types.	It does not allow the user to define data types.
It supports the Multi-Dimensional array.	It does not support the Multi-Dimensional array.
It allows concurrent procedure calls.	It does not allow concurrent calls.
A mod operator is present.	A mod operator is not present.
Unary reduction operator is not present.	Unary reduction operator is present.

CHAPTER ONE
INTRODUCTION TO FPGA AND VHDL

It is more difficult to learn.	It is easy to learn.

History of VHDL

- VHDL was developed by the **Department of Defence** (**DOD**) in 1980.
- **1980:** The Department of Defence wanted to make circuit design self-documenting.
- **1983:** The development of VHDL began with a joint effort by IBM, Inter-metrics, and Texas Instruments.
- **1985 (VHDL Version 7.2):** The final version of the language under the government contract was released.
- **1987:** DOD permitted for commercial purpose, and VHDL became IEEE Standard 1076-1987.
- **1993:** VHDL was re-standardized to enhance the language
- **1996:** A VHDL package used with synthesis tools and became a part of the IEEE 1076 standard.
- **1999:** Analog Mixed Signal extension (VHDL-AMS)
- **2008:** IEEE Standard 1076-2008 (New features) was released.

Why VHDL?

VHDL is used for the following purposes:

- For Describing hardware.
- As a modeling language.
- For a simulation of hardware.
- For early performance estimation of system architecture.
- For the synthesis of hardware.

CHAPTER ONE
INTRODUCTION TO FPGA AND VHDL

Advantages of VHDL

A list of advantages of VHDL is given below:

- It supports various design methodologies like Top-down approach and Bottom-up approach.
- It provides a **flexible design language**.
- It allows better design management.
- It allows detailed implementations.
- It supports a multi-level abstraction.
- It provides **tight coupling** to lower levels of design.
- It supports all **CAD tools**.
- It strongly supports **code reusability** and **code sharing**.

Disadvantages of VHDL

A list of disadvantages of VHDL is given below:

- It requires specific knowledge of the structure and syntax of the language.
- It is more difficult to visualize and troubleshoot a design.
- Some VHDL programs cannot be synthesized.
- VHDL is more difficult to learn.

CHAPTER TWO

SOFTWARE TOOLS

Elaf A.Saeed

CHAPTER TWO
SOFTWARE TOOLS

FPGA Design Tool: ISE, Vivado, Quartus, …

- Nowadays, there are two major players in the FPGA domain: **Xilinx** and **Altera** (now part of Intel). These two company dominate 90% of the FPGA market with roughly equal share. Others (Actel, Lattice Logic).
- Completely SOC.

In figure 2.1 that shown the software tool for FPGA.

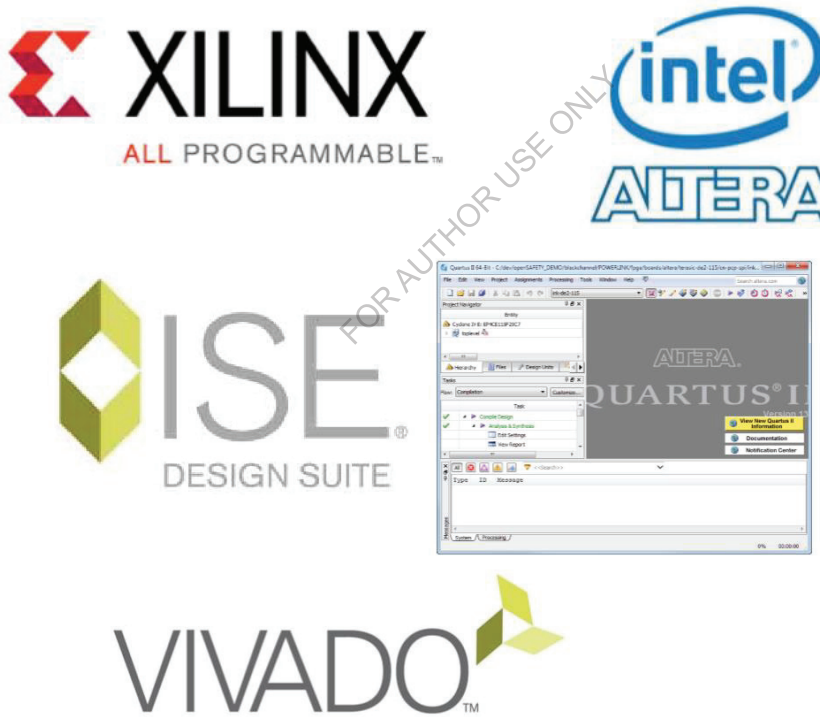


Figure 2.1: Software Tools for FPGA

In this book we use the ISE program.

ISE Installation Steps

- Download the ISE Program.
- Select Download Location Directory, as shown in figure 2.2.

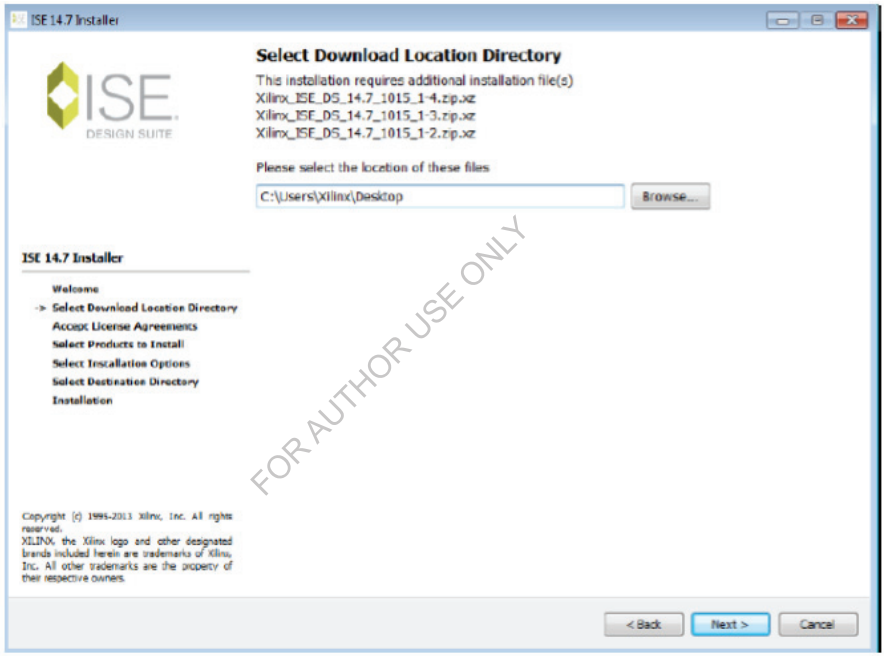


Figure 2.2: Select Download Location

- Accepting Software Licenses, as shown in figure 2.3.

CHAPTER TWO
SOFTWARE TOOLS

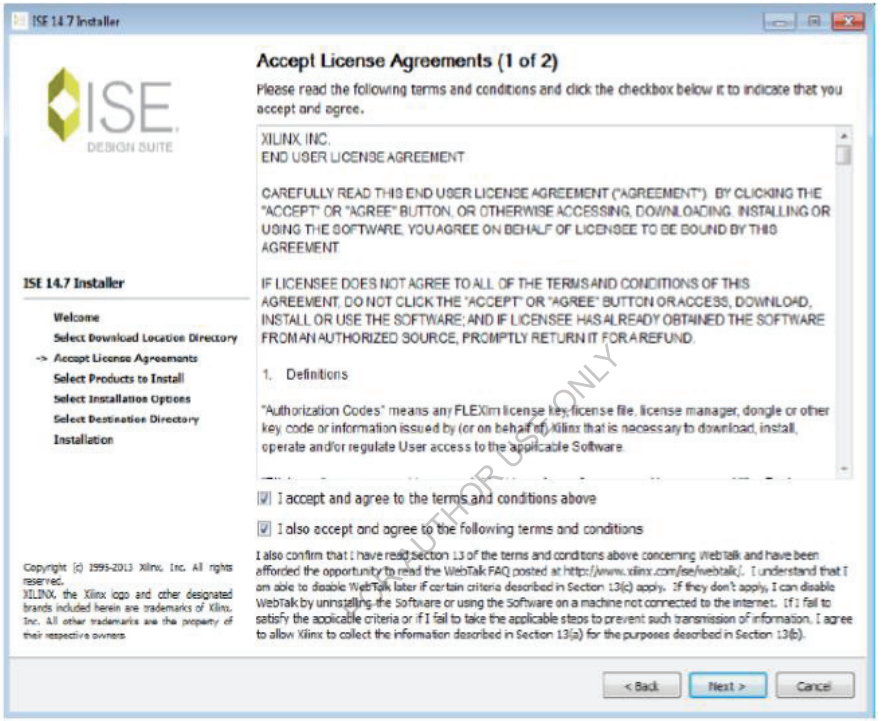


Figure 2.3: Accepting Software Licenses

- Select Destination Directory, as shown in figure 2.4.

CHAPTER TWO
SOFTWARE TOOLS

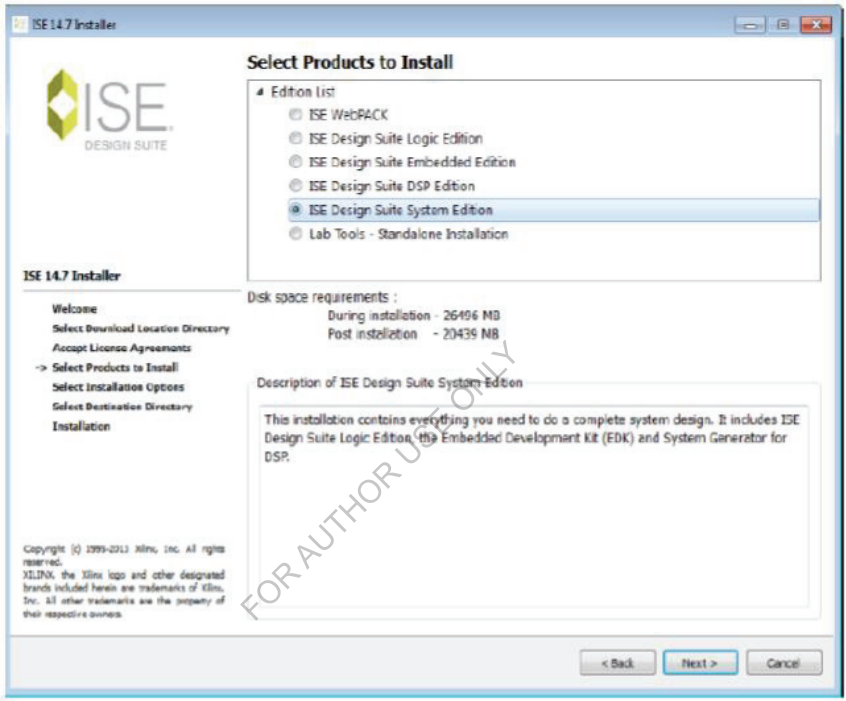


Figure 2.4: Select Destination Directory

- Installation Complete, as shown in figure 2.5.

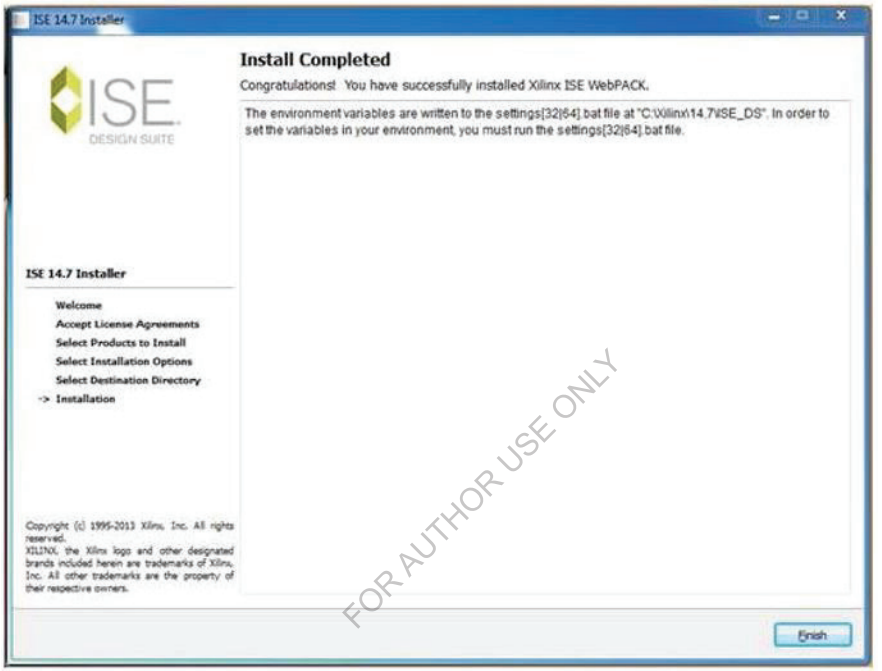


Figure 2.5: Instillation Complete

• Load License, as shown in figure 2.6.

CHAPTER TWO
SOFTWARE TOOLS

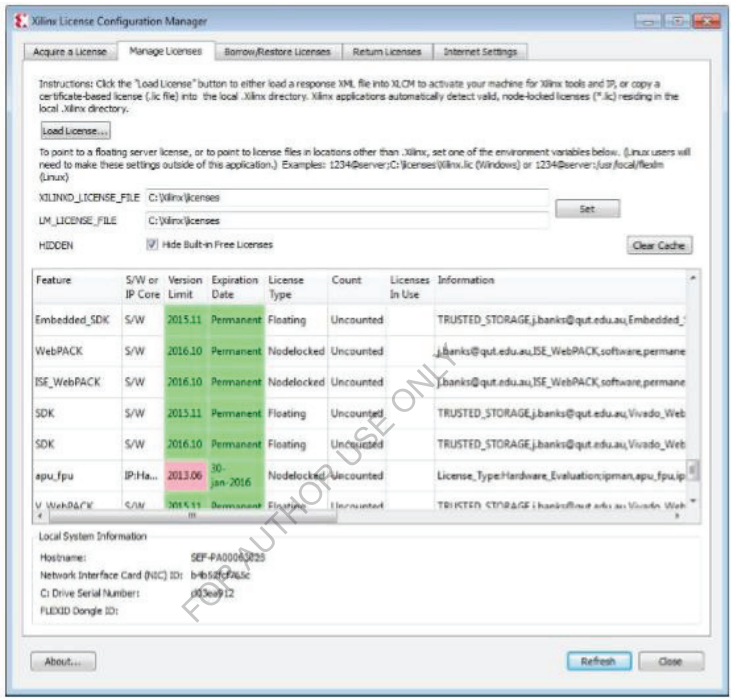


Figure 2.6: Load License

In figure 2.7 that shown the ISE program window.

CHAPTER TWO
SOFTWARE TOOLS

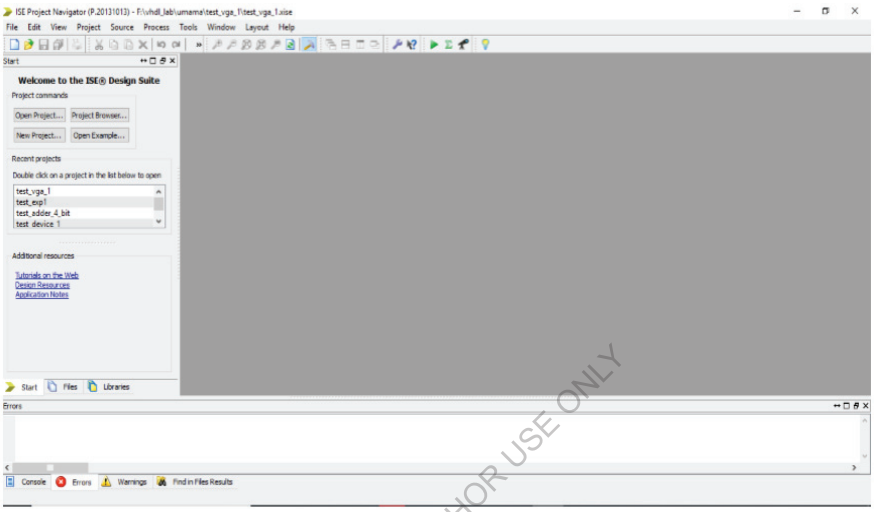


Figure 2.7: ISE Program Window

ISE Staring New Project

Click on New Project➔Enter File Name and Path, as shown in figure 2.8.

CHAPTER TWO
SOFTWARE TOOLS

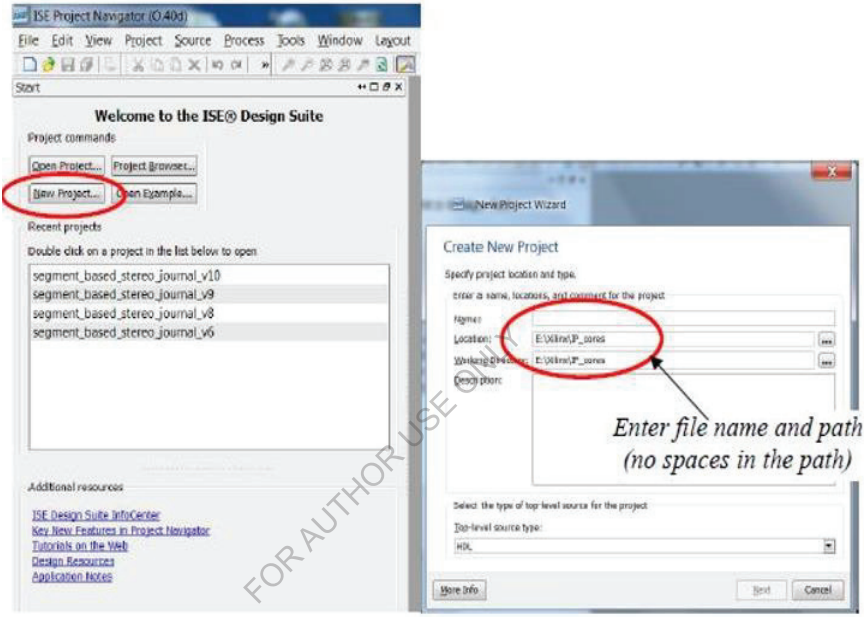


Figure 2.8: Make New Project

Specify the device and project properties, as shown in figure 2.9.

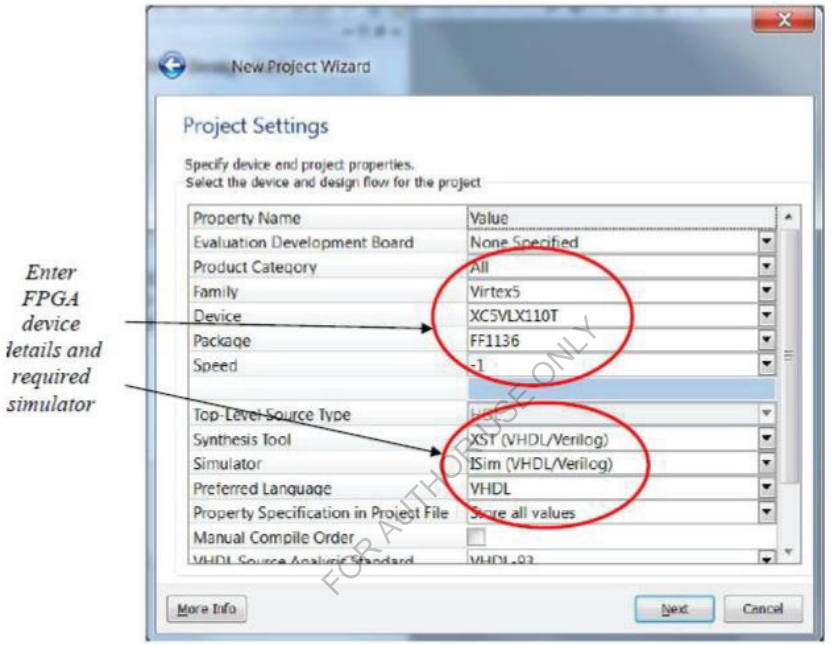


Figure 2.9: Specify the Device and Project Properties

Right Click on the device name ➔ select New Project, as shown in figure 2.10.

CHAPTER TWO
SOFTWARE TOOLS

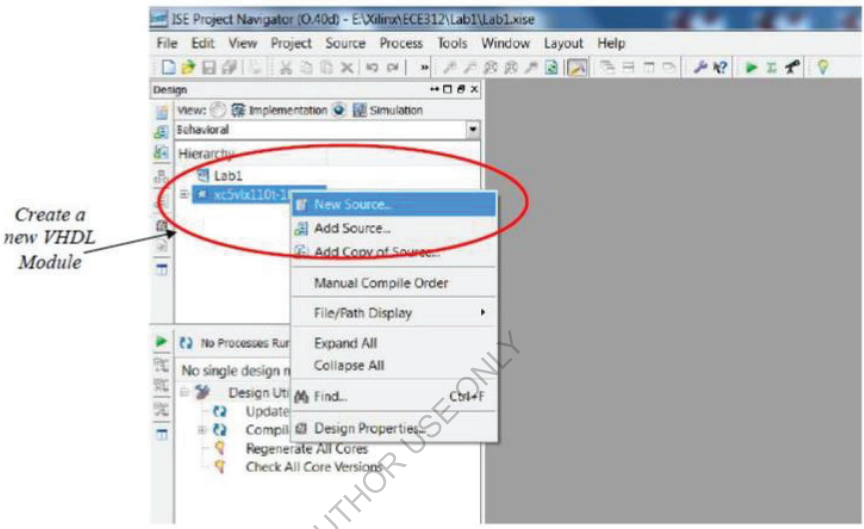


Figure 2.10: Create a new VHDL Module

Select VHDL Module➔write the file name and location➔NEXT, as shown in figure 2.11.

CHAPTER TWO
SOFTWARE TOOLS

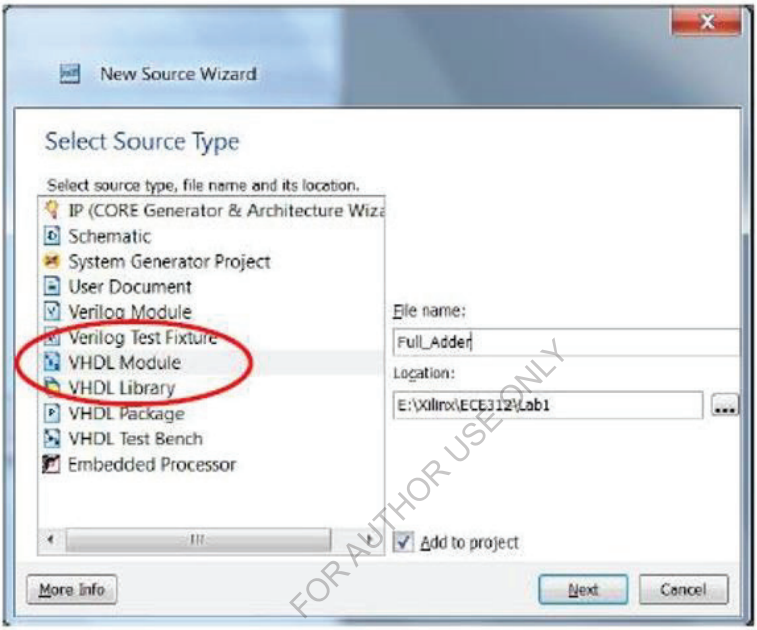


Figure 2.11: Select VHDL Module

In the Architecture Name tab write "**Behavioral**" and enter the appropriate inputs and outputs of the or gate in the Port Name tab. Select Next and Finish to create the VHDL module, as shown in figure 2.12.

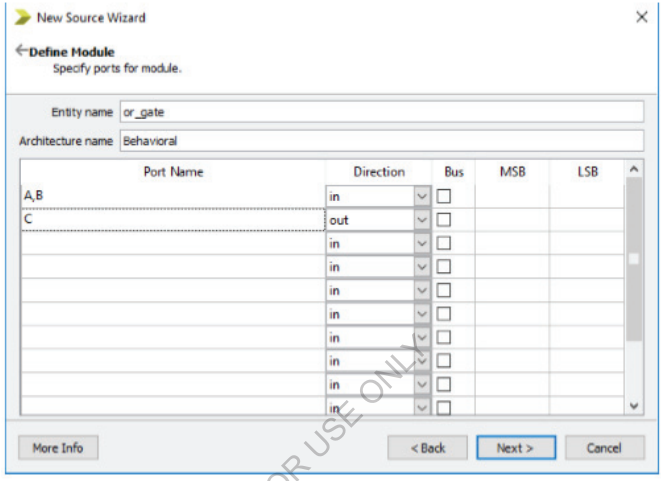


Figure 2.12: Properties Inputs and Outputs

A VHDL module should now appear in project navigator, as shown below. This provides the entity and architecture layout for your circuit.

Edit the **VHDL file** to realize the 1-bit or gate circuit. When you have coded the circuit in VHDL run the syntax checker: This is available in the process window under Synthesize-XST→ Check Syntax, see figure 2.13 below. When the design is error-free, synthesize the VHDL file: Double click on Synthesize-XST.

CHAPTER TWO
SOFTWARE TOOLS

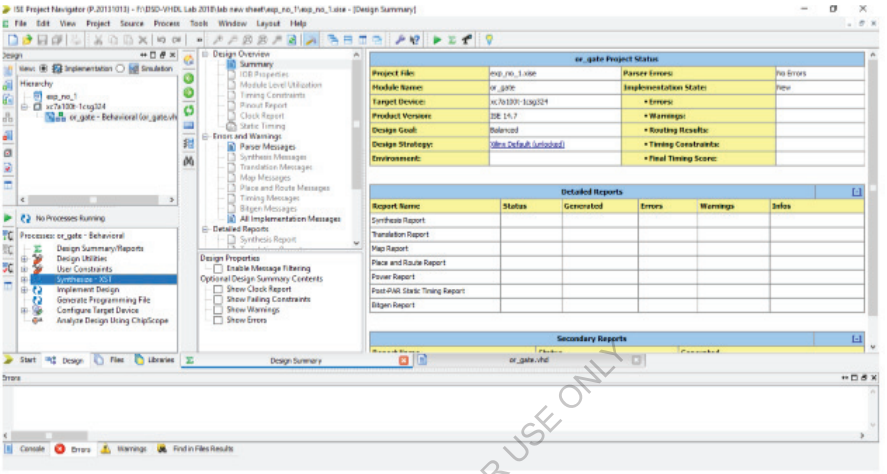


Figure 2.13: Design Summery Window

Now examine the synthesis report – this is available by double-clicking Synthesize-Report in the panel Design Summary. Scroll down the file to find the section entitled Final Report, as shown in Figure 2.14 below. Note the number of LUTs and IBUF/OBUFs that have been synthesized to create the circuit.

CHAPTER TWO
SOFTWARE TOOLS

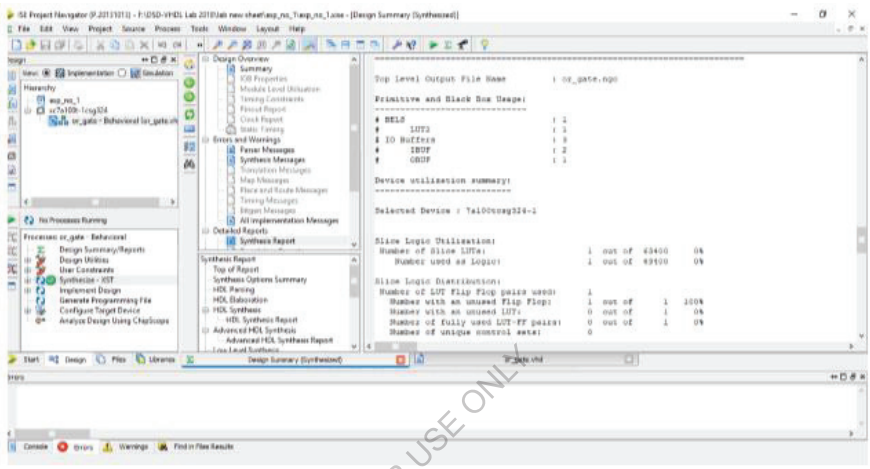


Figure 2.14: Synthesis Report

Behavioral Simulation of a Designed Circuit In this section, we will introduce the concept of test bench and show how to verify the function of our design by behavioral simulation.

What is a test bench?

A test bench is an entity (usually a VHDL/Verilog program) which is used to verify the correctness of a design. The design to be verified is called **Unit Under Test (UUT)**. The test bench supplies stimuli to the design, observes the outputs of the design, and compares the observed outputs with the expected values. If any mismatch happens, the test bench issues certain messages signifying that there are errors in the design. Figure 2.15 below shows the concept of test bench.

CHAPTER TWO
SOFTWARE TOOLS

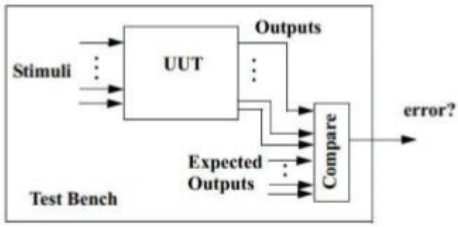


Figure 2.15: UUT

Advanced EDA tools such as Xilinx ISE usually have the capability to automatically generate the test bench. All the users need to do is specifying the waveforms of the stimuli and the expected outputs; the software produces the testbench program which can be tailored later on.

Build the test bench by specifying waveforms

In the **Design Panel**, select "Simulation", click on the name of project (for example: 'or_gate - Behavioral'), and then double-click 'Simulate Behavioral Model'. You may need to hit the '+' beside 'ISim Simulator', as shown in Figure 2.15.

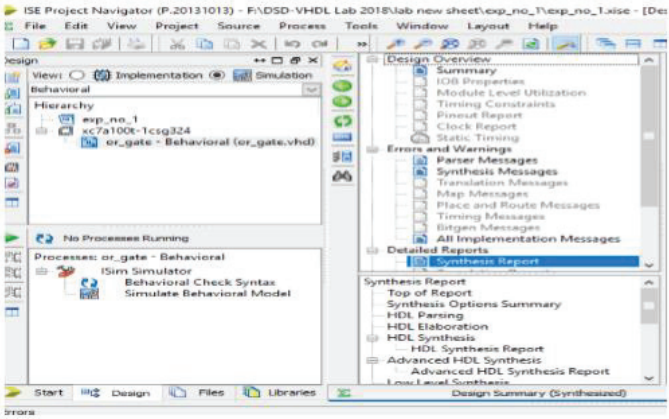


Figure 2.16: Build Test Bench

In the window that shown in figure 2.17 that opens, change to the 'Default.wcfg' tab:

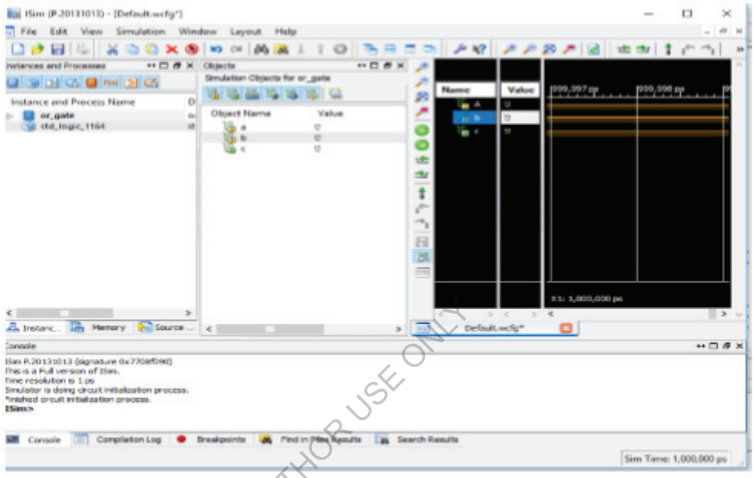


Figure 2.17: Test Bench Window

(3) You can rename the inputs/outputs by right-clicking them and hitting 'rename'. If you forget which ports are which, you can look back in ISE Project Navigator (not the simulator). You can also right-click on a particular signal to initialize it to a constant value, or define it as a clock.

(4) Using the waveform display, fill in the truth table. You may need to scroll the waveform to start at time 0. You can click on different times in the waveform and just read S/Cout directly off. Fill in the observations based on this.

(5) Close the ISim window, it will ask if you really want to exit the application, hit "Yes".

Build the test bench using VHDL coding Another way to simulate the circuit to verify its expected Boolean behavior is by manually creating a Testbench for the circuit. Tasks (1) Right click on xc5vlx110t-1ff1136 icon in the sources window and select New Source,

then select VHDL Test Bench and enter a filename and follow through to Finish, as shown in Figure 2.18.

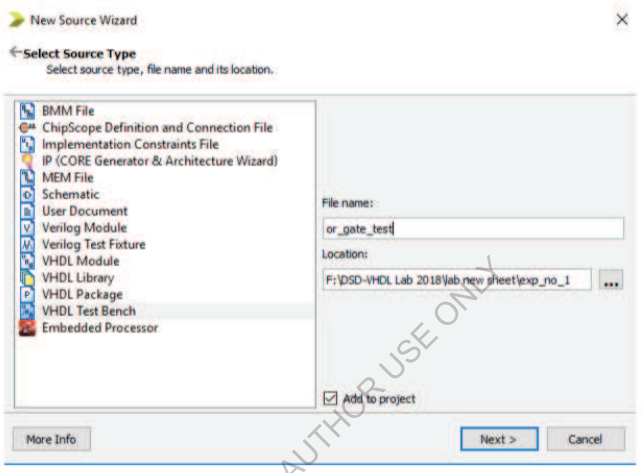


Figure 2.18: Create Test Bench

You should see the same window in figure 2.19.

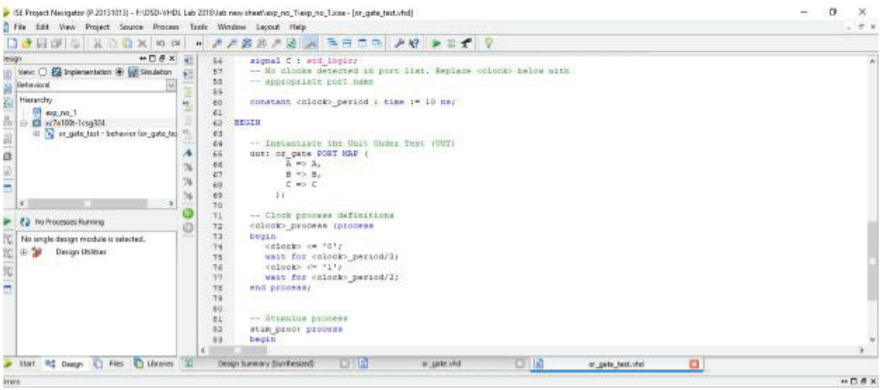


Figure 2.19: Test Bench Window

Note: You will need to remove the term <clock>_period in the file and replace it with clock_period. You will also need to comment the lines of the process that generates the clock, as our circuit is a combinational one.

Now select the Testbench file or_gate_test, in the top window in the left; Select the process Simulate Behavioral Model and click-on RUN. This will invoke the ISim simulator and execute the Testbench file or_gate_test.

Schematic Window

To display the schematic of circuit design in VHDL that is by

click on the Tools ➔ select Schematic Viewer ➔ Select RTL

You will see a window with a circuit diagram designed with VHDL as shown in the following example in figure 2.20.

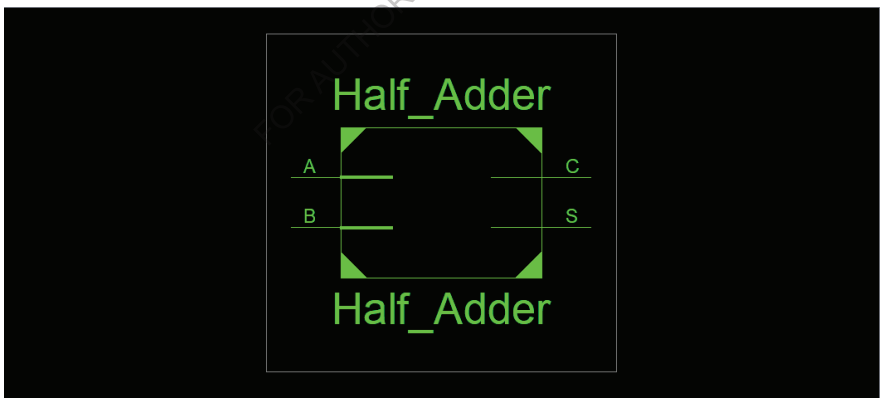


Figure 2.20: Schematic of Half Adder Circuit

Double click on the circuit in figure 2.20 will display the internal design of circuit, as shown in figure 2.21.

CHAPTER TWO
SOFTWARE TOOLS

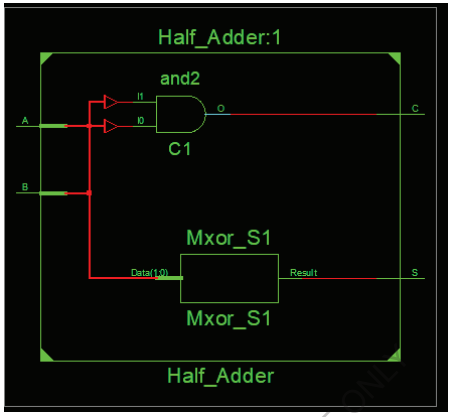


Figure 2.21: Internal Schematic of Half Adder Circuit

Display Truth Table and Equation

To display the truth table and equation of VHDL design by

Click on the Tools ➔ select Schematic Viewer ➔ Select Technology ➔ OK

You will see a window in figure 2.22.

Click on the circuit name ➔ Add ➔ Create Schematic

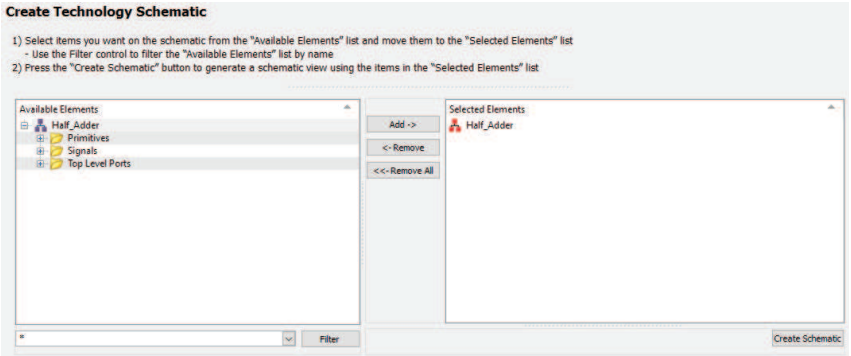
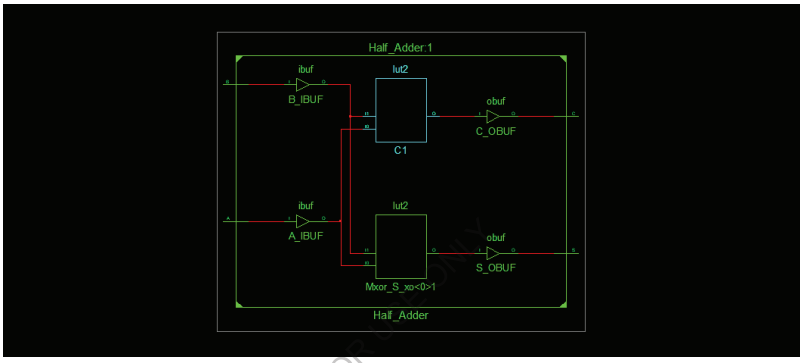
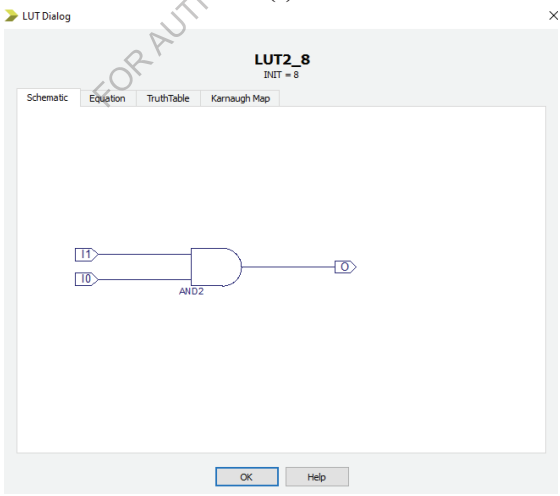


Figure 2.22: Create Technology Schematic

For example, the schematic in the window of figure 2.23 (a) appear. Doble click on any block that display the truth table, equation, K-map, and schematic of this block, as shown in figure 2.23 (b).



(a)



(b)

Figure 2.23: Display the Truth Table, equation, K-map, and schematic

CHAPTER THREE

VHDL PROGRAM ELEMENTS

Elaf A.Saeed

CHAPTER THREE
VHDL PROGRAM ELEMENTS

Basic Elements of VHDL

In figure 3.1 that shown the visualization of a design entity in the hardware description language VHDL.

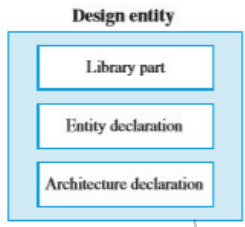


Figure 3.1: visualization of a design entity in the hardware description language VHDL

There are the following three basic elements of VHDL:

1- THE LIBRARY PART

To make a complete VHDL design, we need to place a library part that consists of a library clause and a use clause within each design entity so that we can use a standard logic library with its data type definitions, functions, and procedures. Once you write a VHDL design, you must analyze, compile, or synthesize the code with a vendor's design tool. We will use the terms analyze, compile, and synthesize interchangeably. The synthesize process identifies syntactical errors—that is, those errors governing the rules of the language. The library is a storage place that contains packages that supply information for your design. The library also provides storage for your compiled VHDL code. Figure 3.2 shows three types of libraries that are available for VHDL designs. These are the main libraries we will use.

CHAPTER THREE
VHDL PROGRAM ELEMENTS

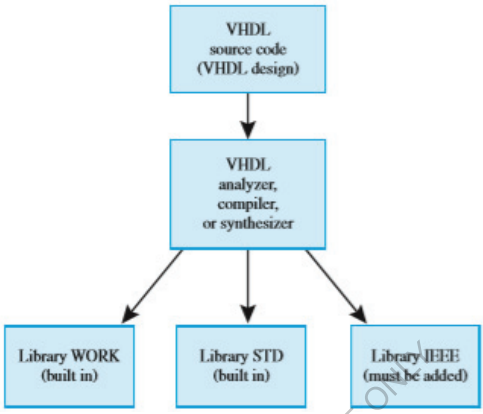


Figure 3.2: Available library units

The **WORK** and **STD** libraries in Figure 3.2 are implicitly declared (built in), but **IEEE** library is not and must be added to a VHDL design via a library clause to make it visible to the design. The **WORK** library is the default library, and this is the storage place for the current design you are working with and where that design is placed after it is compiled. The **STD** library is the storage place for the miscellaneous and logical operators such as not, and, and nor. The **STD** library also contains relational operators such as =, >, <. The **IEEE** library is the storage place for **9-value** data types called **std_logic** in the package **IEEE. STD_LOGIC_1164**. The main reason for using the IEEE library is for design portability—that is, you can use your VHDL source code with many different software vendors. The syntax below shows a library clause and a use clause for a VHDL design.

Syntax

```
--These are the library and use clauses (The Library Part)
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

CHAPTER THREE
VHDL PROGRAM ELEMENTS

The library **IEEE** clause makes the **IEEE library** visible to a design. A library contains packages. The **IEEE.STD_LOGIC_1164** package is specified by the use clause so that data type definitions, functions, and procedures that reside in the package are visible to the design. A specific component name can be used inside the package, but the wild card **ALL** is used to indicate that all the declarations inside the package can be used.

2- THE ENTITY DECLARATION

The Entity is used to specify the input and output ports of the circuit. An Entity usually has one or more ports that can be inputs (**in**), outputs (**out**), input-outputs (**inout**), or **buffer**. An **Entity** may also include a set of generic values that are used to declare properties of the circuit.

You can declare an entity using the following syntax:

- **Simplified syntax**

```
    entity entity_name is
ort (
port_1_name : mode data_type;
ort_2_name : mode data_type;
            .....
            Port_n_name : mode data_type
            );
   end entity_name;
```

Example:

```
entiy orgate is
  port (
          a : in   std_logic;
          b : in   std_logic;
          c : out  std_logic
        );
end orgate;
```

- **Using generic**

If an entity is generic, then it must be declared before the ports. Generic does not have a mode, so it can only pass information into the entity.

Syntax:

```
entity entity_name is
   generic (
            generic_1_name : data_type;
            generic_2_name : data_type;
             ........
            generic_n_name : data_type
             );
port (
        port_1_name : mode data_type;
         port_2_name : mode data_type;
         ........
         Port_n_name : mode data_type
          );
   end entity_name;
```

Example:

```
entity Logic_Gates is
generic (Delay : Time := 10ns);
port (
      Input1 : in std_logic;
      Input2 : in std_logic;
      Output : out std_logic
      );
end Logic_Gates;
```

Rules for writing Port name:

- Port name consist of letters, digits, and underscores.
- It always begins with a letter.
- Port name is case insensitive.

Modes of Port

in	Input port
out	Output port
inout	Bidirectional port
buffer	Buffered output port

3- THE ARCHITECTURE DECLARATION

Architecture is the actual description of the design, which is used to describe how the circuit operates. It can contain both concurrent and sequential statements.

An architecture can be declared using the following syntax:

```
architecture architecture_name of entity_name is
begin
   (concurrent statements )
end architecture_name;
```

CHAPTER THREE
VHDL PROGRAM ELEMENTS

Example:

```
architecture synthesis of andgate is
begin
        c <= a AND b;
end synthesis;
```

Types of Modeling styles in VHDL

There are 3 types of modeling styles in VHDL:

1. Data flow modeling (Design Equations)

Only (1) Boolean equations, (2) conditional signal assignments (CSAs), and (3) selected signal assignments (SSAs) can be used in a dataflow design style. These signal assignments are concurrent statements (CSs) because they are evaluated by the VHDL compiler concurrently or at the same time. The order in which they are written is not important. The simplest form of a dataflow design style is a Boolean equation. The equation implies how the hardware should be created; therefore, implication is used to create the hardware required for a circuit when using a dataflow design style.

2. Behavioral modeling (Explains Behavioral)

Only (1) Boolean equations, (2) if statements, and (3) case statements can be used in a behavioral design style. These equations and statements must be placed inside a process. The complete process is a concurrent statement; however, the statements inside the process are evaluated sequentially by the VHDL compiler—that is, in the order in which they are written in the process. A behavioral architecture declaration creates the structure for a circuit by inference (creating the circuit from deduction by reasoning from the general to the specific).

3. Structural modeling (Connection of sub modules)

CHAPTER THREE
VHDL PROGRAM ELEMENTS

Hardware blocks or components are used in a structural design style. An annotated circuit or schematic must be provided to use this design style—that is, a schematic with all the input, output, and internal signals clearly labeled. The schematic is separated into hardware blocks or components and then simply connected together just like wiring a digital circuit using gates. The installation or placement of the hardware blocks or components are referred to in VHDL as instantiation and their interconnections are referred to as port mapping. Instantiation and port-mapping statements are concurrent statements because they are evaluated at the same time. The order in which they are written is not important. A structural architecture declaration creates the structure for a circuit by the way you wire it up or connect the components. As you will see later, a combination of design styles can also be used together in an architecture declaration to generate a hardware circuit for a system. We will refer to a collection of modules or components that form a hardware circuit in VHDL as a system. When we use a combination of design styles, we will call the name of the architecture mixed to indicate that a mixture of design styles is being used.

VHDL objects

VHDL uses the following three types of objects:

1. Constants

Constant is an object which can only hold a single value that cannot be changed during the whole code.

Example: constant number_of_bytes integer **:=8**;

2. Variables

A variable also holds a single value of a given type. The value of the variable may be changed during the simulation by using variable assignment operator.

Variables are used in the processes and subprograms.

Variables are assigned by the assignment operator "**:=**".

Example: variable index: integer **:=0**;

3. Signals

Signals can be declared in architecture and used anywhere within the architecture. Signals are assigned by the assignment operator "**<=**".

Example:

```
Signal sig1: std_logic;
Sig1 <= '1'
```

Data Types in VHDL

Data Types are the abstract representation of stored data.

There are the following data types in VHDL -

1. Scalar Types

- **Integer**

  Integer data types are the set of positive and negative whole numbers.

- **Floating point**

  Floating point data types are the set of positive and negative numbers that contain a decimal point.

- **Enumeration**

  Enumeration data type is used to increase the readability of the code.

- **Physical**

  Physical data type describes objects in terms of a base unit, multiples of base unit, and a specified range.

2. Composite Types

- **Arrays**

  Arrays are used to hold multiple values of the same types under a single identifier

- **Record**

  Records are used to specify one or more elements, and each element has a different name and different type.

VHDL Operators

VHDL Operators are used for constructing the expressions.

There are the following types of operators in VHDL:

1. Logical Operators

Logical Operators are used to control the program flow. When the logical operators combined with signals or variables, then it is used to create combinational logic.

VHDL supports the following logical operators:

- and
- or
- nand
- nor
- xor
- xnor
- not

2. Relational Operators

In VHDL, relational operators are used to compare two operands of the same data type, and the received result is always of the Boolean type.

VHDL supports the following Relational Operators:

- = Equal to
- /= Not Equal to
- < Less than
- > Greater than

- <= Less than or equal to
- >= Greater than or equal to

3. Arithmetic Operators

Arithmetic Operators are used to perform arithmetic operations. These operators are **numeric types**, such as **integer** and **real**.

VHDL uses the following Arithmetic Operators:

- + Addition
- - Subtraction
- * Multiplication
- / Division
- & Concatenation
- mod Modulus
- rem Remainder
- abs Absolute Value
- ** Exponentiation

4. Shift Operators

In VHDL, shift operator is used to perform the bit manipulation on the data by shifting and rotating the bits of its first operand right or left.

VHDL supports the following Miscellaneous Operators:

- Sll shift logical left
- Srl shift logical right
- Sla shift arithmetic left
- Sra shift arithmetic right
- Rol rotate left
- Ror rotate right

CHAPTER FOUR

VHDL EXPERIEMENTS

Elaf A.Saeed

CHAPTER FOUR
VHDL EXPERIEMENTS

Introduction

In this chapter, we will do a set of experiments on digital circuits such as logic gates, flip flop gates, counters etc. By using the ISE simulation program.

Buffer & Inverter Logic Gates

Aim of Experiment

In this experiment, we will do the two logic gates are buffer and inverter.

Theory

NOT Gate

Inverting **NOT gates** are single input devices which have an output level that is normally at logic level "**1**" and goes "**LOW**" to a logic level "**0**" when its single input is at logic level "**1**", in other words its "**inverts**" (**complements**) its input signal. The output from a NOT gate only returns "**HIGH**" again when its input is at logic level "**0**" giving us the Boolean expression of: **A = Q.**

Then we can define the operation of a single input digital logic NOT gate as being:

"If A is NOT true, then Q is true"

Transistor NOT Gate

A simple 2-input logic NOT gate can be constructed using an **(RTL) Resistor-transistor switches** as shown below in figure 4.1 with the input connected directly to the transistor base. The transistor must be saturated "**ON**" for an inverted output "**OFF**" at Q.

CHAPTER FOUR
VHDL EXPERIEMENTS

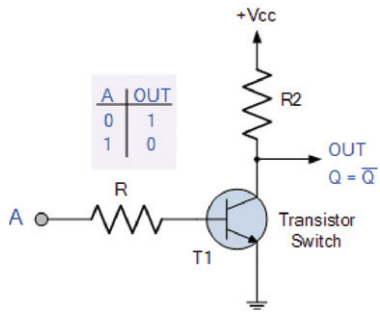


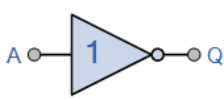
Figure 4.1: transistor not gate

Logic NOT Gates are available using digital circuits to produce the desired logical function. The standard NOT gate is given a symbol whose shape is of a triangle pointing to the right with a circle at its end. This circle is known as an "inversion bubble" and is used in NOT, NAND and NOR symbols at their output to represent the logical operation of the NOT function. This bubble denotes a signal inversion (complementation) of the signal and can be present on either or both the output and/or the input terminals.

The Logic NOT Gate Truth Table

In table 4.1 that shows the NOT gate truth table.

Table 4.1: NOT Gate Truth Table

Symbol	Truth Table	
 Inverter or NOT Gate	A	Q
	0	1
	1	0
Boolean Expression **Q = not A or $\overline{A}$**	Read as inverse of **A** gives Q	

Logic NOT gates provide the complement of their input signal and are so called because when their input signal is "**HIGH**" their output state will NOT be "**HIGH**". Likewise, when their input signal is "**LOW**" their output state will NOT be "**LOW**". As they are single input devices, logic NOT gates are not normally classed as "**decision**" making devices or even as a gate, such as the AND or OR gates which have two or more logic inputs. Commercially available NOT gates IC's are available in either 4 or 6 individual gates within a single IC package.

The **"bubble" (o)** present at the end of the NOT gate symbol above denotes a signal inversion (complementation) of the output signal. But this bubble can also be present at the gates input to indicate an **active-LOW** input. This inversion of the input signal is not restricted to the NOT gate only but can be used on any digital circuit or gate as shown with the operation of inversion being exactly the same whether on the input or output terminal. The easiest way is to think of the bubble as simply an inverter.

Signal Inversion using **Active-low input Bubble**, as shown in figure 4.2.

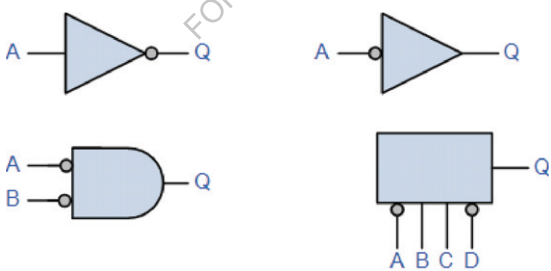


Figure 4.2: Bubble Notation for Input Inversion

Buffer Gate

If we were to connect two inverter gates together, as shown in figure 4.3. so that the output of one fed into the input of another, the two inversion functions would "cancel" each other out so that there would be no inversion from input to final output:

CHAPTER FOUR
VHDL EXPERIEMENTS

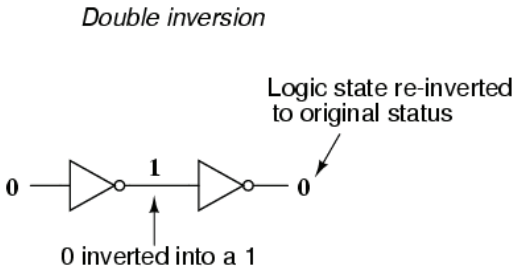
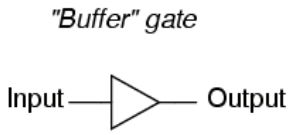


Figure 4.3: Double Inversion

While this may seem like a pointless thing to do, it does have practical application. Remember that gate circuits are **signal amplifiers**, regardless of what logic function they may perform. A weak signal source (one that is not capable of sourcing or sinking very much current to a load) may be boosted by means of two inverters like the pair shown in the previous illustration. The logic level is unchanged, but the full current-sourcing or -sinking capabilities of the final inverter are available to drive a load resistance if needed. For this purpose, a special logic gate called a buffer is manufactured to perform the same function as two inverters. Its symbol is simply a triangle, with no inverting "bubble" on the output terminal:



Input	Output
0	0
1	1

Figure 4.4: Buffer Gate

CHAPTER FOUR
VHDL EXPERIEMENTS

VHDL Code & Simulation

CODE (NOT Gate)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity NOT_Gate is
    Port ( X : in  STD_LOGIC;
           F : out  STD_LOGIC);
end NOT_Gate;

architecture Behavioral of NOT_Gate is

begin
F<= NOT X;

end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY NOT_Gate_test IS
END NOT_Gate_test;
ARCHITECTURE behavior OF NOT_Gate_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT NOT_Gate
    PORT(
         X : IN  std_logic;
         F : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal X : std_logic := '0';
   --Outputs
   signal F : std_logic;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: NOT_Gate PORT MAP (
          X => X,
          F => F
        );
   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 50 ns.
      wait for 50 ns;
      -- insert stimulus here
```

```
    x<='0';
    wait for 75 ns;
    x<='1';
    wait for 75 ns;
    x<='0';
    wait for 75 ns;
    x<='1';
    wait;
end process;
END;
```

In figure 4.5 that shows the Signal and schematic of Inverter gate.

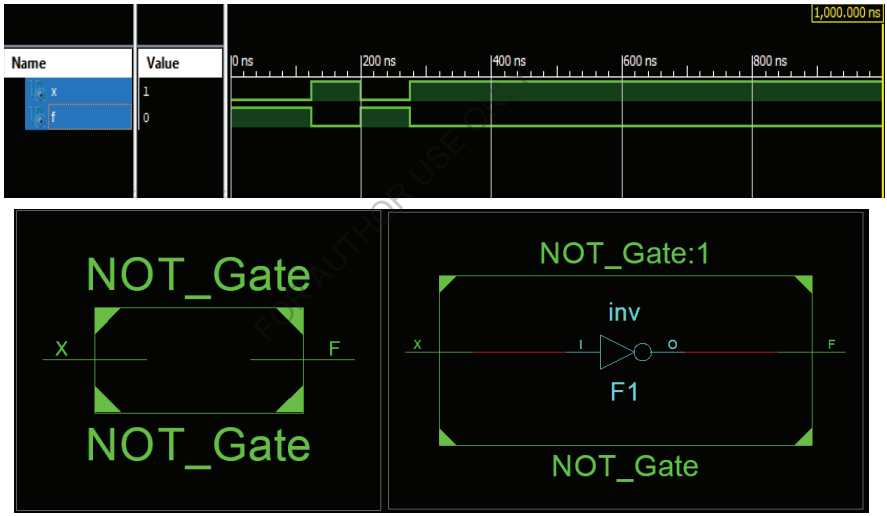


Figure 4.5: signal and Schematic of Inverter Gate

CODE (Buffer Gate)

CHAPTER FOUR
VHDL EXPERIEMENTS

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Buffer_Gate is
    Port ( X : in  STD_LOGIC;
           Fbuf : out  STD_LOGIC);
end Buffer_Gate;
architecture Behavioral of Buffer_Gate is
begin
Fbuf <= X;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Buffer_Gate_test IS
END Buffer_Gate_test;
ARCHITECTURE behavior OF Buffer_Gate_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT Buffer_Gate
    PORT(
         X : IN  std_logic;
         Fbuf : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal X : std_logic := '0';
   --Outputs
   signal Fbuf : std_logic;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: Buffer_Gate PORT MAP (
          X => X,
          Fbuf => Fbuf
        );
   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 50 ns.
      wait for 50 ns;
      -- insert stimulus here

     X<='0';
     wait for 100 ns;
     X<='1';
     wait;
  end process;
 END;
```

In figure 4.5 that shows the Signal and schematic of buffer gate.

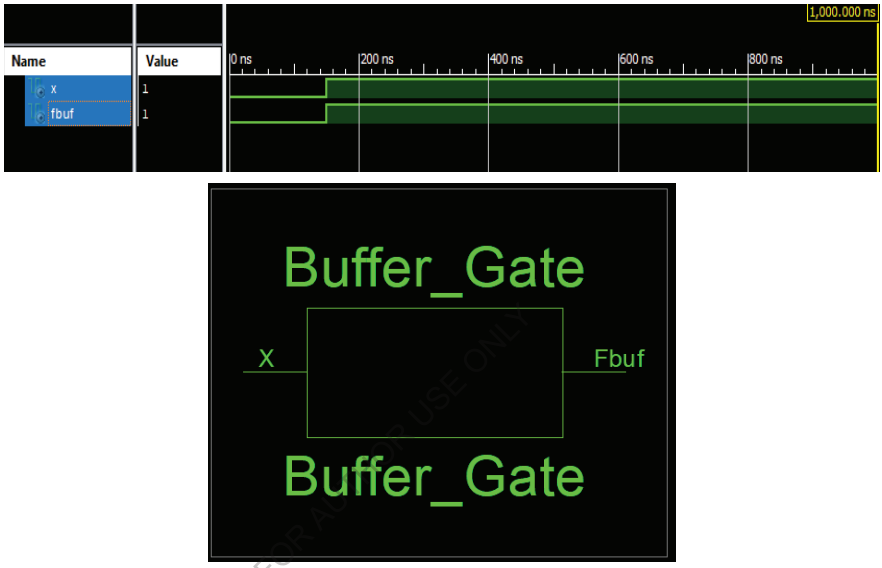


Figure 4.6: signal and Schematic of Buffer Gate

Discussion

1- Write and simulate the circuit in figure 4.3.

CHAPTER FOUR
VHDL EXPERIEMENTS

AND & OR Logic Gates

Aim of Experiment

In this experiment, we will do the two logic gates are AND and OR.

Theory

AND Gate

The output state of a digital logic AND gate only returns "**LOW**" again when ANY of its inputs are at a logic level "**0**". In other words, for a logic **AND gate**, any LOW input will give a LOW output. In figure 4.7 that shown the AND gate.

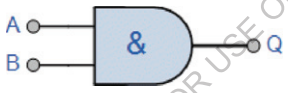


Figure 4.7: AND Gate

The logic or Boolean expression given for a digital logic AND gate is that for Logical Multiplication which is denoted by a single dot or full stop symbol, (.) giving us the Boolean expression of: A.B = Q.

Then we can define the operation of a digital 2-input logic AND gate as being:

"If both A and B are true, then Q is true"

2-input Transistor AND Gate

A simple 2-input logic AND gate can be constructed using **RTL Resistor-transistor switches** connected together as shown below in figure 4.8 with the inputs connected directly to the transistor bases. Both transistors must be saturated "**ON**" for an output at **Q**.

CHAPTER FOUR
VHDL EXPERIEMENTS

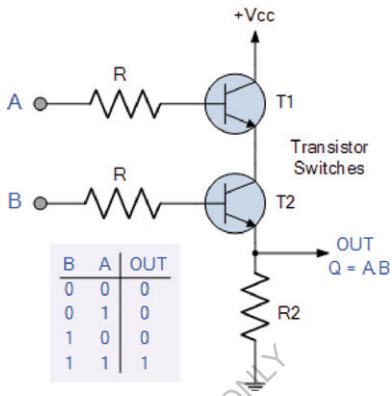


Figure 4.8: 2-Inputs Transistor AND Gate

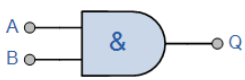
Logic AND Gates are available using digital circuits to produce the desired logical function and is given a symbol whose shape represents the logical operation of the AND gate.

Digital Logic "AND" Gate Types

The 2-input Logic AND Gate

Table 4.2 that shown the truth table of 2-inputs NAND gate.

Table 4.2: 2-Inputs NAND Gate Truth Table

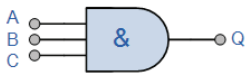
Symbol	Truth Table		
	B	A	Q
 2-input AND Gate	0	0	0
	0	1	0
	1	0	0
	1	1	1
Boolean Expression **Q = A.B**	Read as A **AND** B gives Q		

CHAPTER FOUR
VHDL EXPERIEMENTS

The 3-input Logic AND Gate

Table 4.3 that shown the truth table of 3-inputs NAND gate.

Table 4.3: 3-Inputs NAND Gate Truth Table

Symbol	Truth Table			
 3-input AND Gate	C	B	A	Q
	0	0	0	0
	0	0	1	0
	0	1	0	0
	0	1	1	0
	1	0	0	0
	1	0	1	0
	1	1	0	0
	1	1	1	1
Boolean Expression **Q = A.B.C**	Read as A **AND** B **AND** C gives Q			

Because the Boolean expression for the logic AND function is defined as (.), which is a binary operation, AND gates can be cascaded together to form any number of individual inputs. However, commercially available AND gate IC's are only available in standard **2**, **3**, or **4-input packages**. If additional inputs are required, then standard AND gates will need to be **cascaded together** to obtain the required input value, for example.

Multi-input AND Gate

In figure 4.9 that shows the 6-input AND function.

CHAPTER FOUR
VHDL EXPERIEMENTS

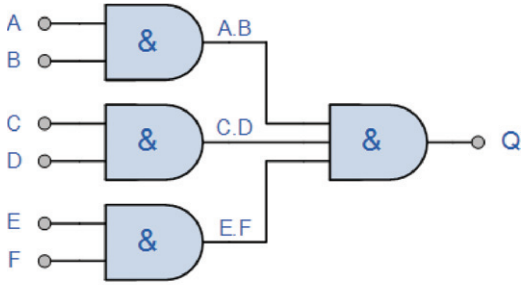


Figure 4.9: 6-Inputs AND Function

The Boolean Expression for this 6-input AND gate will therefore be:

Q = (A.B). (C.D). (E.F)

In other words:

A AND B AND C AND D AND E AND F gives Q

If the number of inputs required is an odd number of inputs any "unused" inputs can be held HIGH by connecting them directly to the power supply using suitable "Pull-up" resistors.

Commonly available digital logic AND gate IC's include:

TTL Logic AND Gate	CMOS Logic AND Gate
74LS08 Quad 2-input	CD4081 Quad 2-input
74LS11 Triple 3-input	CD4073 Triple 3-input
74LS21 Dual 4-input	CD4082 Dual 4-input

In figure 4.10 that shows the **7408 Quad 2-input AND Gate.**

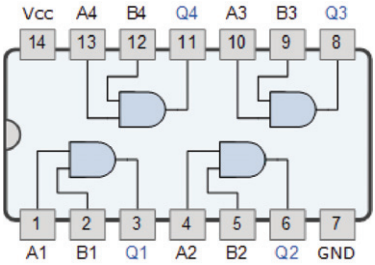


Figure 4.10: IC AND Gate

OR Gate

The output, **Q** of a "**Logic OR Gate**" only returns "**LOW**" again when ALL of its inputs are at a logic level "**0**". In other words, for a logic OR gate, any "**HIGH**" input will give a "**HIGH**", logic level "**1**" output. In figure 4.11 that shown the OR gate symbol.

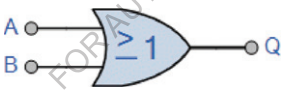


Figure 4.11: OR Gate

The logic or Boolean expression given for a digital logic **OR gate** is that for Logical Addition which is denoted by a plus sign, (+) giving us the Boolean expression of: **A+B = Q**.

Thus, a **logic OR gate** can be correctly described as an "**Inclusive OR gate**" because the output is true when both of its inputs are true (**HIGH**). Then we can define the operation of a 2-input logic OR gate as being:

"If either A or B is true, then Q is true"

2-input Transistor OR Gate

A simple **2-input inclusive OR gate** can be constructed using **RTL Resistor-transistor switches** connected together as shown below in figure 4.12 with the inputs connected

directly to the transistor bases. Either transistor must be saturated "**ON**" for an output at **Q**.

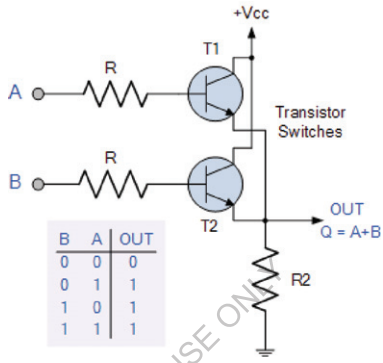


Figure 4.12: 2-Inputs Transistor OR Gate

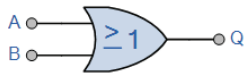
Logic OR Gates are available using digital circuits to produce the desired logical function and is given a symbol whose shape represents the logical operation of the OR gate.

Digital Logic "OR" Gate Types

The 2-input Logic OR Gate

In table 4.4 that shown the truth table of 2-input logic OR gate.

Table 4.4: 2-Inputs OR Gate Truth Table

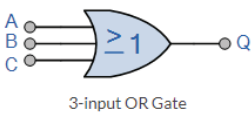
Symbol	Truth Table		
	B	A	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	1
Boolean Expression **Q = A+B**	Read as A **OR** B gives Q		

CHAPTER FOUR
VHDL EXPERIEMENTS

The 3-input Logic OR Gate

In table 4.5 that shown the truth table of 3-input logic OR gate.

Table 4.5: 3-Inputs OR Gate Truth Table

Symbol	Truth Table			
 3-input OR Gate	C	B	A	Q
	0	0	0	0
	0	0	1	1
	0	1	0	1
	0	1	1	1
	1	0	0	1
	1	0	1	1
	1	1	0	1
	1	1	1	1
Boolean Expression **Q = A+B+C**	Read as A **OR** B **OR** C gives Q			

Like the AND gate, the OR function can have any number of individual inputs. However, commercially available OR gates are available in 2, 3, or 4 inputs types. Additional inputs will require gates to be cascaded together for example.

Multi-input OR Gate

In figure 4.13 that shown the Multi-input OR gate.

CHAPTER FOUR
VHDL EXPERIEMENTS

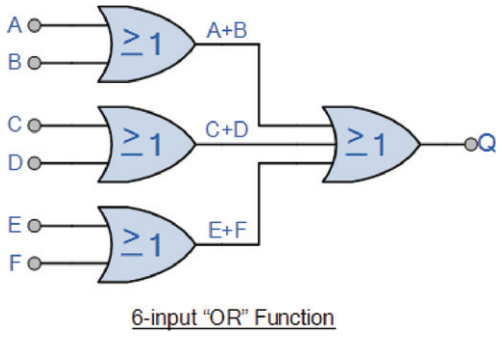


Figure 4.13: Multi-Input OR Gate

The Boolean Expression for this 6-input OR gate will therefore be:

Q = (A+B) +(C+D) +(E+F)

In other words:

A OR B OR C OR D OR E OR F gives Q

If the number of inputs required is an odd number of inputs any "unused" inputs can be held LOW by connecting them directly to ground using suitable "Pull-down" resistors. Commonly available digital logic OR gate IC's include:

TTL Logic OR Gates	CMOS Logic OR Gates
74LS32 Quad 2-input	CD4071 Quad 2-input
	CD4075 Triple 3-input
	CD4072 Dual 4-input

In figure 4.14 that shown the **7432 Quad 2-input Logic OR Gate**.

CHAPTER FOUR
VHDL EXPERIEMENTS

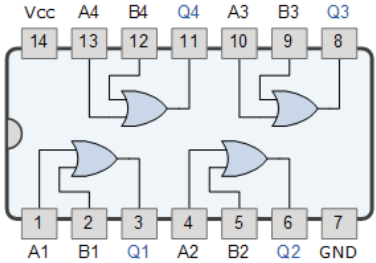


Figure 4.14: 7432 Quad 2-input Logic OR Gate

VHDL Code & Simulation

CODE (AND Gate):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND_2IN is
    Port ( A,B : in  STD_LOGIC;
           Q : out  STD_LOGIC);
end AND_2IN;

architecture Behavioral of AND_2IN is

begin
Q<= A AND B;

end Behavioral;
```

Test Bench:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY AND_2IN_test IS
END AND_2IN_test;

ARCHITECTURE behavior OF AND_2IN_test IS
-- Component Declaration for the Unit Under Test (UUT)
 COMPONENT AND_2IN
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         Q : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   --Outputs
   signal Q : std_logic;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: AND_2IN PORT MAP (
          A => A,
          B => B,
          Q => Q
        );
   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 50 ns;
      -- insert stimulus here
      A<='0';
      B<='0';
      wait for 50 ns;
      A<='0';
      B<='1';
      wait for 50 ns;
      A<='1';
      B<='0';
      wait for 50 ns;
      A<='1';
      B<='1';
      wait;
   end process;
 END;
```


In figure 4.15 that shown the Signal and schematic of AND gate.

CHAPTER FOUR
VHDL EXPERIEMENTS

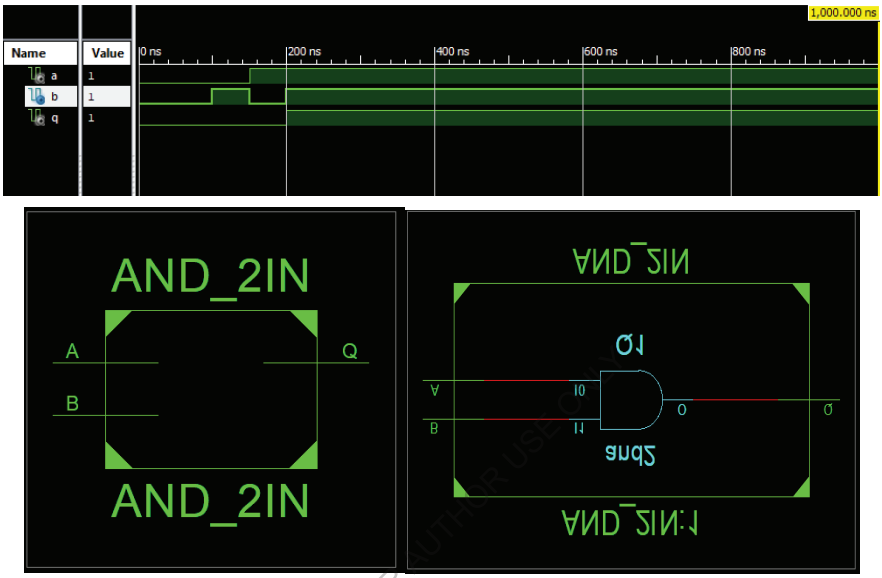


Figure 4.15: signal and Schematic of AND Gate

CODE (OR Gate):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity OR_2IN is
    Port ( A,B : in  STD_LOGIC;
           Q : out  STD_LOGIC);
end OR_2IN;

architecture Behavioral of OR_2IN is

begin
Q<= A OR B;
end Behavioral;
```

Test Bench:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY OR_2IN_test IS
END OR_2IN_test;
ARCHITECTURE behavior OF OR_2IN_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT OR_2IN
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         Q : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   --Outputs
   signal Q : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: OR_2IN PORT MAP (
       A => A,
       B => B,
       Q => Q
     );
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
   A<= '0';
   B<= '0';
   wait for 50 ns;
   A<= '0';
   B<= '1';
   wait for 50 ns;
   A<= '1';
   B<= '0';
   wait for 50 ns;
   A<= '1';
   B<= '1';
   wait;
end process;
END;
```


In figure 4.16 that shows the Signal and schematic of OR gate.

CHAPTER FOUR
VHDL EXPERIEMENTS

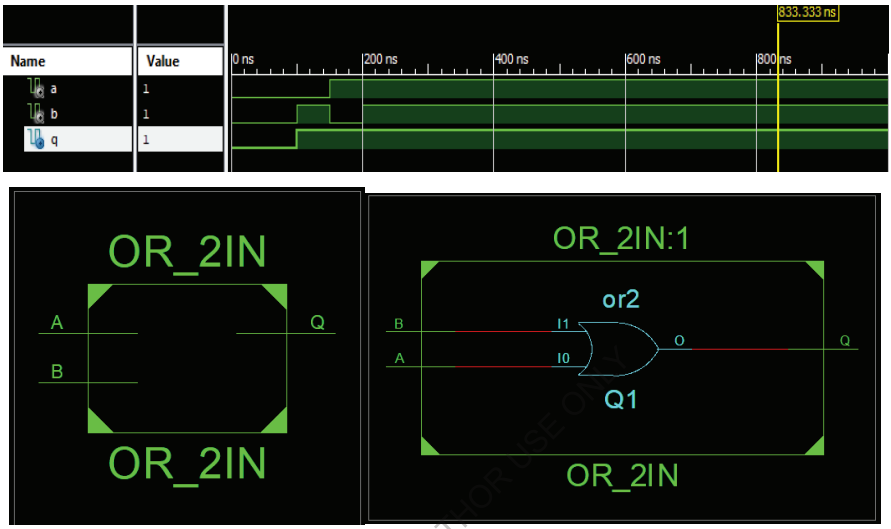


Figure 4.16: signal and Schematic of OR Gate

Discussion

1- Write and Simulate the 3-input AND gate.

2- Write and Simulate the circuit in figure 4.9.

3- Write and Simulate the 3-input OR gate.

4- Write and Simulate the circuit in figure 4.13.

CHAPTER FOUR
VHDL EXPERIEMENTS

NAND & NOR Logic Gates

Aim of Experiment

In this experiment, we will do the two logic gates are NAND and NOR.

Theory

NAND Gate

The Logic NAND Gate is a combination of a digital logic AND gate and a NOT gate connected together in series. In figure 4.17 that shown the NAND gate symbol.

Figure 4.17: NAND Gate Symbol

The **NAND (Not – AND) gate** has an output that is normally at logic level "**1**" and only goes "**LOW**" to logic level "**0**" when **ALL** of its inputs are at logic level "**1**". The **Logic NAND Gate** is the reverse or "***Complementary***" form of the AND gate we have seen previously.

Logic NAND Gate Equivalence

In figure 4.18 that shown the logic NAND gate equivalence.

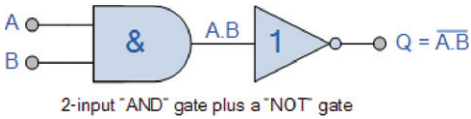


Figure 4.18: NAND Gate Symbol

The logic or Boolean expression given for a **logic NAND gate** is that for **Logical Addition**, which is the opposite to the **AND gate**, and which it performs on the complements of the inputs. The Boolean expression for a **logic NAND gate** is denoted by a single dot or full stop symbol, (**.**) with a line or Overline, ($\overline{\ }$) over the expression to

signify the NOT or logical negation of the NAND gate giving us the Boolean expression of: **A.B = Q**.

Then we can define the operation of a 2-input digital logic NAND gate as being:

"If both A and B are true, then Q is NOT true"

Transistor NAND Gate

A simple **2-input logic NAND gate** can be constructed using **RTL Resistor-transistor switches** connected together as shown below in figure 4.19 with the inputs connected directly to the transistor bases. Either transistor must be cut-off "**OFF**" for an output at **Q**.

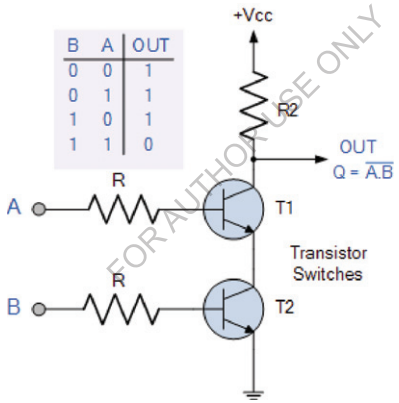


Figure 4.19: Transistor NAND Gate

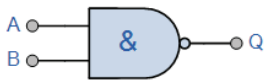
Logic NAND Gates are available using **digital circuits** to produce the desired logical function and is given a symbol whose shape is that of a standard AND gate with a circle, sometimes called an "**inversion bubble**" at its output to represent the **NOT gate symbol** with the logical operation of the **NAND gate** given as.

The Digital Logic "NAND" Gate

2-input Logic NAND Gate

In table 4.6 that shown the truth table of 2-input logic NAND gate.

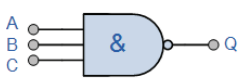
Table 4.6: 2-Inputs NAND Gate Truth Table

Symbol	Truth Table		
 2-input NAND Gate	B	A	Q
	0	0	1
	0	1	1
	1	0	1
	1	1	0
Boolean Expression $Q = \overline{A.B}$	Read as A **AND** B gives **NOT** Q		

3-input Logic NAND Gate

In table 4.7 that shown the truth table of 3-input logic NAND gate.

Table 4.7: 3-Inputs NAND Gate Truth Table

Symbol	Truth Table			
 3-input NAND Gate	C	B	A	Q
	0	0	0	1
	0	0	1	1
	0	1	0	1
	0	1	1	1
	1	0	0	1
	1	0	1	1
	1	1	0	1
	1	1	1	0
Boolean Expression $Q = \overline{A.B.C}$	Read as A **AND** B **AND** C gives **NOT** Q			

As with the **AND function** seen previously, the **NAND function** can also have any number of individual inputs and commercially available NAND Gate IC's are available in standard **2**, **3**, or **4** input types. If additional inputs are required, then the standard **NAND gates** can be cascaded together to provide more inputs for example.

A 4-input NAND Function

In figure 4.20 that shown a 4-input NAND Function.

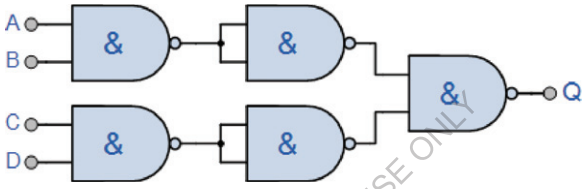


Figure 4.20: 4-input NAND Function

The Boolean Expression for this **4-input logic NAND** gate will therefore be: **Q = A.B.C.D**

If the number of inputs required is an odd number of inputs any "unused" inputs can be held HIGH by connecting them directly to the power supply using suitable "Pull-up" resistors. The Logic NAND Gate function is sometimes known as the Sheffer Stroke Function and is denoted by a vertical bar or upwards arrow operator, for example, A NAND **B = A|B or A↑B**.

The "Universal" NAND Gate

The Logic NAND Gate is generally classed as a "**Universal**" gate because it is one of the most commonly used logic gate types. **NAND gates** can also be used to produce any other type of logic gate function, and in practice the **NAND gate** forms the basis of most practical logic circuits.

By connecting them together in various combinations the three basic gate types of AND, OR and NOT function can be formed using only **NAND gates**, for example.

Various Logic Gates using only NAND Gates

In figure 4.21 that shown the Various Logic Gates using only NAND Gates.

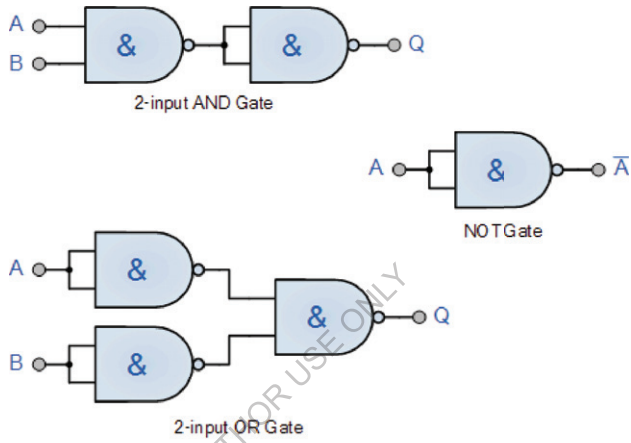


Figure 4.21: Various Logic Gates using only NAND Gates

As well as the three common types above, Exclusive-OR, Exclusive-NOR and **standard NOR gates** can be formed using just individual **NAND gates**.

Commonly available digital logic **NAND gate IC's** include:

TTL Logic NAND Gates	CMOS Logic NAND Gates
74LS00 Quad 2-input	CD4011 Quad 2-input
74LS10 Triple 3-input	CD4023 Triple 3-input
74LS20 Dual 4-input	CD4012 Dual 4-input
74LS30 Single 8-input	

In figure 4.22 that shown the **7400 Quad 2-input Logic NAND Gate**.

CHAPTER FOUR
VHDL EXPERIEMENTS

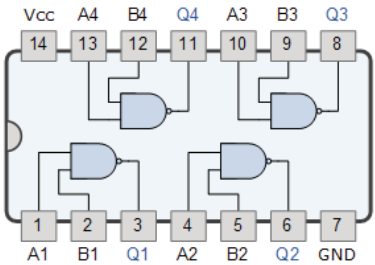


Figure 4.22: 7400 Quad 2-input Logic NAND Gate

NOR Gate

The **Logic NOR Gate** is a combination of the digital logic **OR gate** and an **inverter** or **NOT gate** connected together in series. In figure 4.23 that shown the logic NOR gate symbol.

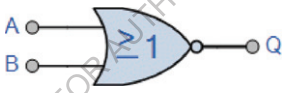


Figure 4.23: Logic NOR Gate Symbol

The **inclusive NOR (Not-OR)** gate has an output that is normally at logic level "**1**" and only goes "**LOW**" to logic level "**0**" when any of its inputs are at logic level "**1**". The Logic **NOR Gate** is the reverse or "**Complementary**" form of the **inclusive OR gate** we have seen previously.

Logic NOR Gate Equivalent

In figure 4.24 that shown the logic NOR gate equivalent.

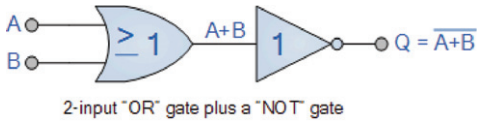


Figure 4.24: Logic NOR Gate Equivalent

The logic or Boolean expression given for a logic NOR gate is that for Logical Multiplication which it performs on the complements of the inputs. The Boolean expression for a logic NOR gate is denoted by a plus sign, (+) with a line or Overline, (¯) over the expression to signify the NOT or logical negation of the NOR gate giving us the Boolean expression of: **A+B = Q**.

Then we can define the operation of a 2-input digital logic NOR gate as being:

"If both A and B are NOT true, then Q is true"

Transistor NOR Gate

A simple **2-input logic NOR gate** can be constructed using **RTL Resistor-transistor switches** connected together as shown below in figure 4.25 with the inputs connected directly to the transistor bases. Both transistors must be cut-off "**OFF**" for an output at **Q**.

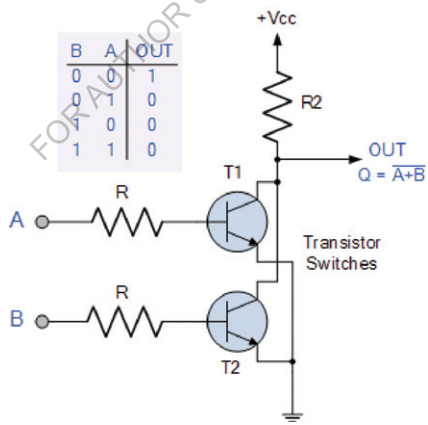


Figure 4.25: Transistor NOR Gate

Logic NOR Gates are available using digital circuits to produce the desired logical function and is given a symbol whose shape is that of a standard OR gate with a circle, sometimes called an "inversion bubble" at its output to represent the NOT gate symbol with the logical operation of the NOR gate given as.

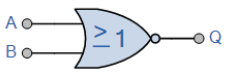
CHAPTER FOUR
VHDL EXPERIEMENTS

The Digital Logic "NOR" Gate

2-input NOR Gate

In table 4.8 that shown the truth table of 2-input logic NOR gate.

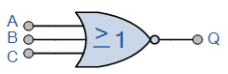
Table 4.8: 2-Inputs NOR Gate Truth Table

Symbol	Truth Table		
 2-input NOR Gate	B	A	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	0
Boolean Expression $Q = \overline{A+B}$	Read as A **OR** B gives **NOT** Q		

3-input NOR Gate

In table 4.9 that shown the truth table of 3-input logic NOR gate.

Table 4.9: 3-Inputs NOR Gate Truth Table

Symbol	Truth Table			
 3-input NOR Gate	C	B	A	Q
	0	0	0	1
	0	0	1	0
	0	1	0	0
	0	1	1	0
	1	0	0	0
	1	0	1	0
	1	1	0	0
	1	1	1	0
Boolean Expression $Q = \overline{A+B+C}$	Read as A **OR** B **OR** C gives **NOT** Q			

As with the OR function, the NOR function can also have any number of individual inputs and commercially available NOR Gate IC's are available in standard 2, 3, or 4 input types. If additional inputs are required, then the standard NOR gates can be cascaded together to provide more inputs for example.

A 4-input NOR Function

In figure 4.26 that shown the 4-input NOR Function.

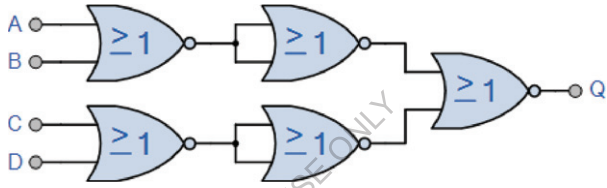


Figure 4.26: 4-input NOR Function

The Boolean Expression for this **4-input NOR gate** will therefore be: **Q = A+B+C+D**

If the number of inputs required is an **odd number** of inputs any "**unused**" inputs can be held **LOW** by connecting them directly to **ground** using suitable "**Pull-down**" resistors. The Logic NOR Gate function is sometimes known as the Pierce Function and is denoted by a downwards arrow operator as shown, **A↓B**.

The "Universal" NOR Gate

Like the **NAND gate** seen in the last section, the **NOR gate** can also be classed as a "**Universal**" type gate. **NOR gates** can be used to produce any other type of logic gate function just like the NAND gate and by connecting them together in various combinations the three basic gate types of **AND**, **OR** and **NOT** function can be formed using only **NOR gates**, for example.

Various Logic Gates using only NOR Gates

In figure 4.27 that shown the Various Logic Gates using only NOR Gates.

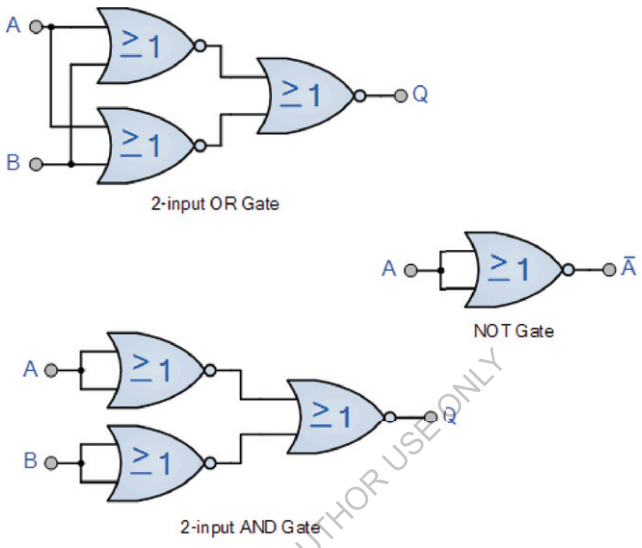


Figure 4.27: Various Logic Gates using only NOR Gates

As well as the three common types above, **Exclusive-OR**, **Exclusive-NOR** and **standard NOR gates** can also be formed using just individual **NOR gates**.

Commonly available **digital logic NOR** gate IC's include:

TTL Logic NOR Gates	CMOS Logic NOR Gates
74LS02 Quad 2-input	CD4001 Quad 2-input
74LS27 Triple 3-input	CD4025 Triple 3-input
74LS260 Dual 4-input	CD4002 Dual 4-input

In figure 4.28 that shown the **7402 Quad 2-input NOR Gate**.

CHAPTER FOUR
VHDL EXPERIEMENTS

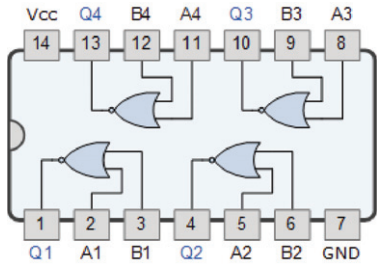


Figure 4.28: 7402 Quad 2-input NOR Gate

VHDL Code & Simulation

CODE (NAND Gate)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity NAND_Gate is
    Port ( A,B : in  STD_LOGIC;
           F : out  STD_LOGIC);
end NAND_Gate;
architecture Behavioral of NAND_Gate is
begin
F <= A NAND B;
end Behavioral;
```


Test Bench

CHAPTER FOUR
VHDL EXPERIEMENTS

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY NAND_Gate_test IS
END NAND_Gate_test;
ARCHITECTURE behavior OF NAND_Gate_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT NAND_Gate
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         F : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   --Outputs
   signal F : std_logic;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: NAND_Gate PORT MAP (
          A => A,
          B => B,
          F => F
        );
   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 50 ns.
      wait for 50 ns;
      -- insert stimulus here
      A <= '0';
      B <= '0';
      wait for 75 ns;
      A <= '0';
      B <= '1';
      wait for 75 ns;
      A <= '1';
      B <= '0';
      wait for 75 ns;
      A <= '1';
      B <= '1';
      wait;
   end process;
END;
```


In figure 4.29 that shown the Signal and schematic of NAND gate.

CHAPTER FOUR
VHDL EXPERIEMENTS

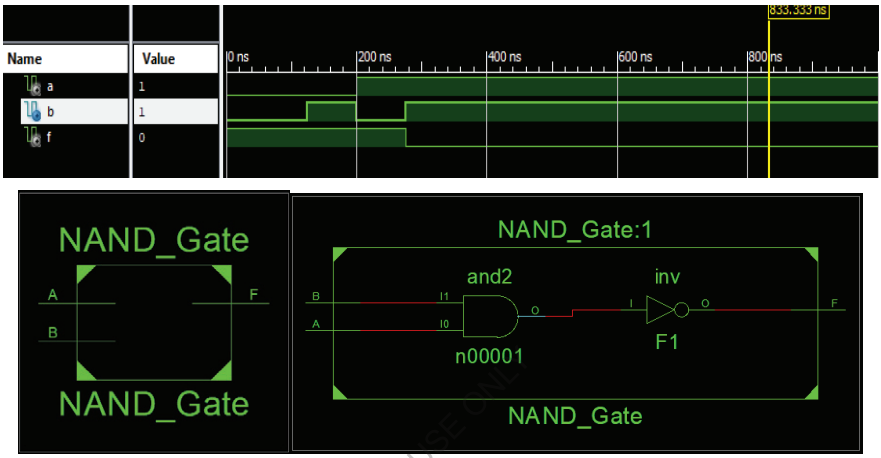


Figure 4.29: signal and Schematic of NAND Gate

CODE (NAND Gate)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity NOR_Gate is
    Port ( A,B : in  STD_LOGIC;
           F : out  STD_LOGIC);
end NOR_Gate;
architecture Behavioral of NOR_Gate is
begin
F <= A NOR B;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY NOR_Gate_test IS
END NOR_Gate_test;
ARCHITECTURE behavior OF NOR_Gate_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT NOR_Gate
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         F : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   --Outputs
   signal F : std_logic;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: NOR_Gate PORT MAP (
          A => A,
          B => B,
          F => F
        );
   -- Stimulus process
   stim_proc: process
   begin
     -- hold reset state for 50 ns.
     wait for 50 ns;
     -- insert stimulus here
     A <= '0';
     B <= '0';
     wait for 100 ns;
     A <= '0';
     B <= '1';
     wait for 100 ns;
     A <= '1';
     B <= '0';
     wait for 100 ns;
     A <= '1';
     B <= '1';
     wait;
  end process;
END;
```


In figure 4.30 that shows the Signal and schematic of NOR gate.

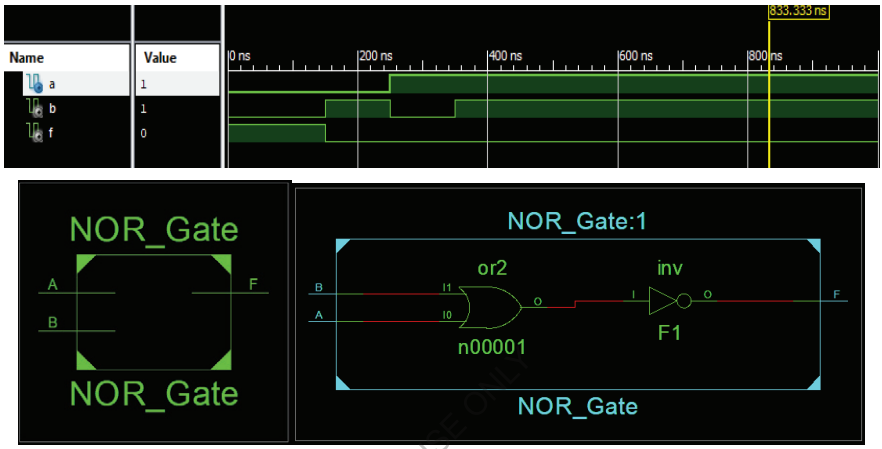


Figure 4.30: signal and Schematic of NOR Gate

Discussion

1- Write and Simulate the 3-input NAND gate.

2- Write and Simulate the 4-input NAND gate.

3- Write and Simulate the 3-input NOR gate.

4- Write and Simulate the 4-input NOR gate.

CHAPTER FOUR
VHDL EXPERIEMENTS

XOR & XNOR Logic Gates

Aim of Experiment

In this experiment, we will do the two logic gates are XOR and XNOR.

Theory

Exclusive-OR (XOR)Gate

The Exclusive-OR logic function is a very useful circuit that can be used in many different types of computational circuits.

In the previous experiments, we saw that by using the three principal gates, the **AND Gate**, the **OR Gate** and the **NOT Gate**, we can build many other types of logic gate functions, such as a **NAND** Gate and a **NOR** Gate or any other type of digital logic function we can imagine.

But there are two other types of digital logic gates which although they are not a basic gate in their own right as they are constructed by combining together other logic gates, their output Boolean function is important enough to be considered as complete logic gates. These two "**hybrid**" logic gates are called the **Exclusive-OR (Ex-OR)** Gate and its complement the **Exclusive-NOR (Ex-NOR)** Gate.

Previously, we saw that for a **2-input OR gate**, if A = "1", OR B = "1", OR BOTH A + B = "1" then the output from the digital gate must also be at a logic level "**1**" and because of this, this type of logic gate is known as an **Inclusive-OR** function. The logic gate gets its name from the fact that it includes the case of Q = "1" when both A and B = "1".

If, however, a logic output "1" is obtained when ONLY A = "1" or when ONLY B = "1" but NOT both together at the same time, giving the binary inputs of "01" or "10", then the output will be "1". This type of gate is known as an **Exclusive-OR function** or more

commonly an **Ex-Or function** for short. This is because its Boolean expression excludes the "OR BOTH" case of Q = "1" when both A and B = "1".

In other words, the output of an **Exclusive-OR** gate ONLY goes "**HIGH**" when its two input terminals are at "**DIFFERENT**" logic levels with respect to each other.

An odd number of logics "1's" on its inputs gives a logic "1" at the output. These two inputs can be at logic level "1" or at logic level "0" giving us the Boolean expression of: **Q = (A ⊕ B) = A.B + A.B**.

The **Exclusive-OR** Gate function, or **Ex-OR** for short, is achieved by combining standard logic gates together to form more complex gate functions that are used extensively in **building arithmetic logic circuits**, **computational logic comparators** and **error detection circuits**.

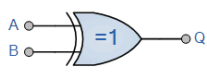
The two-input "Exclusive-OR" gate is basically a modulo two adders, since it gives the sum of two binary numbers and as a result are more complex in design than other basic types of logic gate. The truth table, logic symbol and implementation of a 2-input Exclusive-OR gate is shown below.

The Digital Logic "Exclusive-OR" Gate

2-input Ex-OR Gate

In table 4.10 that shown the truth table of 2-input logic XOR gate.

Table 4.10: 2-Inputs XOR Gate Truth Table

Symbol	Truth Table		
 2-input Ex-OR Gate	B	A	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = A ⊕ B	A **OR** B but NOT **BOTH** gives Q		

Giving the Boolean expression of: **Q = AB + AB**

The truth table above shows that the output of an **Exclusive-OR** gate ONLY goes "**HIGH**" when both of its two input terminals are at "**DIFFERENT**" logic levels with respect to each other. If these two inputs, A and B are both at logic level "1" or both at logic level "0" the output is a "0" making the gate an "odd but not the even gate". In other words, the output is "1" when there are an odd number of 1's in the inputs.

This ability of the Exclusive-OR gate to compare two logic levels and produce an output value dependent upon the input condition is very useful in computational logic circuits as it gives us the following Boolean expression of: **Q = (A ⊕ B) = A.B + A.B**.

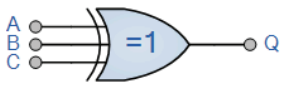
The logic function implemented by a 2-input Ex-OR is given as either: "A OR B but NOT both" will give an output at Q. In general, an Ex-OR gate will give an output value of logic "1" ONLY when there are an **ODD** number of 1's on the inputs to the gate, if the two numbers are equal, the output is "0".

Then an Ex-OR function with more than two inputs is called an "odd function" or modulo-2-sum (Mod-2-SUM), not an **Ex-OR**. This description can be expanded to apply to any number of individual inputs as shown below for a 3-input Ex-OR gate.

3-input Ex-OR Gate

In table 4.11 that shown the truth table of 3-input logic XOR gate.

Table 4.11: 3-Inputs XOR Gate Truth Table

Symbol	Truth Table			
 3-input Ex-OR Gate	C	B	A	Q
	0	0	0	0
	0	0	1	1
	0	1	0	1
	0	1	1	0
	1	0	0	1
	1	0	1	0
	1	1	0	0
	1	1	1	1
Boolean Expression Q = A ⊕ B ⊕ C	"Any **ODD** Number of Inputs" gives Q			

Giving the Boolean expression of: Q = ABC + ABC + ABC + ABC

The symbol used to denote an Exclusive-OR odd function is slightly different to that for the standard Inclusive-OR Gate. The logic or Boolean expression given for a logic OR gate is that of logical addition which is denoted by a standard plus sign.

The symbol used to describe the Boolean expression for an Exclusive-OR function is a plus sign, (+) within a circle (O). This exclusive-OR symbol also represents the mathematical "direct sum of sub-objects" expression, with the resulting symbol for an Exclusive-OR function being given as: (⊕).

We said previously that the Ex-OR function is not a basic logic gate but a combination of different logic gates connected together. Using the 2-input truth table above, we can

expand the Ex-OR function to: (A+B).(A.B) which means that we can realize this new expression using the following individual gates

Ex-OR Function Realization using NAND gates

Exclusive-OR Gates are used mainly to build circuits that perform arithmetic operations and calculations especially Adders and Half-Adders as they can provide a "carry-bit" function or as a controlled inverter, where one input passes the binary data and the other input is supplied with a control signal.

Commonly available digital logic Exclusive-OR gate IC's include:

TTL Logic Ex-OR Gates	CMOS Logic Ex-OR Gates
74LS86 Quad 2-input	CD4030 Quad 2-input

In figure 4.31 that shown the **47486 Quad 2-input Exclusive-OR Gate**.

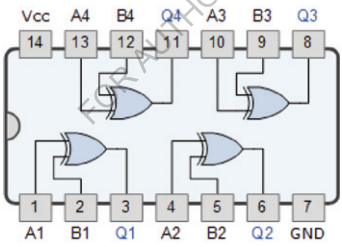


Figure 4.31: 47486 Quad 2-input Exclusive-OR Gate

Exclusive-NOR Gate

The Exclusive-NOR Gate function is a digital logic gate that is the reverse or complementary form of the **Exclusive-OR function**. In figure 4.32 that shown the XOR gate symbol.

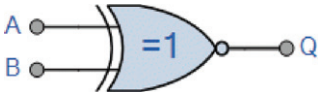


Figure 4.32: XOR Gate Symbol

CHAPTER FOUR
VHDL EXPERIEMENTS

Basically the "**Exclusive-NOR**" gate is a combination of the **Exclusive-OR** gate and the **NOT gate** but has a truth table similar to the standard **NOR gate** in that it has an output that is normally at logic level "1" and goes "LOW" to logic level "0" when ANY of its inputs are at logic level "1".

However, an output "1" is only obtained if BOTH of its inputs are at the same logic level, either binary "1" or "0". For example, "00" or "11". This input combination would then give us the Boolean expression of: $\mathbf{Q = (A \oplus B) = \overline{A}.B + A.\overline{B}}$

Then the output of a digital logic Exclusive-**NOR gate** ONLY goes "**HIGH**" when its two input terminals, A and B are at the "SAME" logic level which can be either at a logic level "1" or at a logic level "0". In other words, an even number of logics "1's" on its inputs gives a logic "1" at the output, otherwise is at logic level "0".

Then this type of gate gives and output "1" when its inputs are "**logically equal**" or "**equivalen**t" to each other, which is why an Exclusive-NOR gate is sometimes called an **Equivalence Gate**.

The logic symbol for an Exclusive-NOR gate is simply an Exclusive-OR gate with a circle or "inversion bubble", (o) at its output to represent the NOT function. Then the Logic Exclusive-NOR Gate is the reverse or "Complementary" form of the **Exclusive-OR gate**, (**A ⊕ B**) we have seen previously.

Ex-NOR Gate Equivalent

In figure 4.33 that shown the XNOR gate symbol.

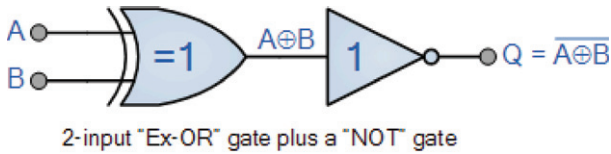


Figure 4.33: XNOR Gate Symbol

CHAPTER FOUR
VHDL EXPERIEMENTS

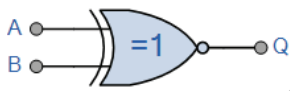
The Exclusive-NOR Gate, also written as: "**Ex-NOR**" or "**XNOR**", function is achieved by combining standard gates together to form more complex gate functions and an example of a 2-input **Exclusive-NOR** gate is given below.

The Digital Logic "Ex-NOR" Gate

2-input Ex-NOR Gate

In table 4.12 that shown the truth table of 2-input logic XNOR gate.

Table 4.12: 2-Inputs XNOR Gate Truth Table

Symbol	Truth Table		
 2-input Ex-NOR Gate	B	A	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	1
Boolean Expression $Q = \overline{A \oplus B}$	Read if A **AND** B the **SAME** gives Q		

Giving the Boolean expression of: **$Q = AB + \overline{A}\overline{B}$**

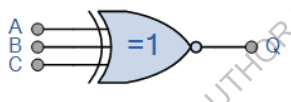
The logic function implemented by a **2-input Ex-NOR** gate is given as "when both A AND B are the SAME" will give an output at Q. In general, an **Exclusive-NOR gate** will give an output value of logic "1" ONLY when there are an **EVEN** number of 1's on the inputs to the gate (the inverse of the Ex-OR gate) except when all its inputs are "LOW". Then an **Ex-NOR** function with more than two inputs is called an "**even function**" or modulo-2-sum (**Mod-2-SUM**), not an **Ex-NOR**. This description can be expanded to

apply to any number of individual inputs as shown below for a 3-input Exclusive-NOR gate.

3-input Ex-NOR Gate

In table 4.13 that shown the truth table of 3-input logic XNOR gate.

Table 4.13: 3-Inputs XNOR Gate Truth Table

Symbol	Truth Table			
 3-input Ex-NOR Gate	C	B	A	Q
	0	0	0	1
	0	0	1	0
	0	1	0	0
	0	1	1	1
	1	0	0	0
	1	0	1	1
	1	1	0	1
	1	1	1	0
Boolean Expression $Q = \overline{A \oplus B \oplus BC}$	Read as "any **EVEN** number of Inputs" gives Q			

Giving the Boolean expression of: **Q = ABC + ABC + ABC + ABC**

We said previously that the **Ex-NOR** function is a combination of different basic logic gates Ex-OR and a NOT gate, and by using the 2-input truth table above, we can expand the Ex-NOR function to: $\mathbf{Q = \overline{A \oplus B} = (\overline{A}.\overline{B}) + (A.B)}$ which means we can realize this new expression using the following individual gates.

Ex-NOR Gate Equivalent Circuit

In figure 4.34 that shown the Ex-NOR Gate Equivalent Circuit.

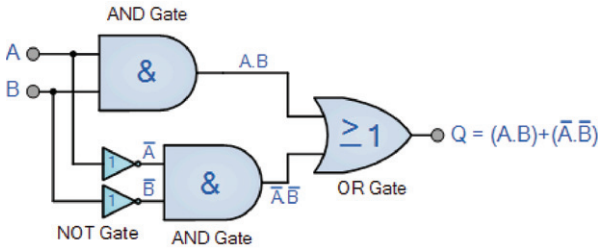


Figure 4.34: Ex-NOR Gate Equivalent Circuit

One of the main disadvantages of implementing the Ex-NOR function above is that it contains three different types logic gates the AND, NOT and finally an OR gate within its basic design. One easier way of producing the Ex-NOR function from a single gate type is to use NAND gates as shown below.

Ex-NOR Function Realization using NAND gates

In figure 4.35 that shown the Ex-NOR Function Realization using NAND gates.

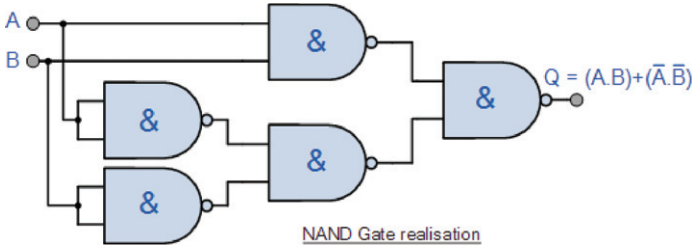


Figure 4.35: Ex-NOR Function Realization using NAND gates

Ex-NOR gates are used mainly in electronic circuits that perform **arithmetic operations** and data checking such as **Adders**, **Subtractors** or **Parity Checkers**, etc. As the Ex-NOR gate gives an output of logic level "1" whenever its two inputs are equal it can be used to

compare the magnitude of two binary digits or numbers and so Ex-NOR gates are used in **Digital Comparator circuits**.

Commonly available digital logic Exclusive-NOR gate IC's include:

TTL Logic Ex-NOR Gates	CMOS Logic Ex-NOR Gates
74LS266 Quad 2-input	CD4077 Quad 2-input

In figure 4.36 that shown the **74266 Quad 2-input Ex-NOR Gate**.

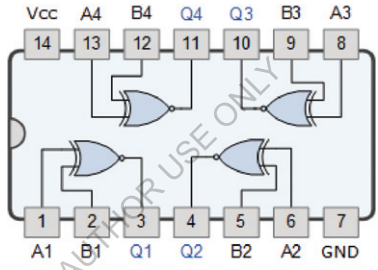


Figure 4.36: 74266 Quad 2-input Ex-NOR Gate

VHDL Code & Simulation

CODE (XOR Gate)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity XOR_Gate is
    Port ( A,B : in  STD_LOGIC;
           F : out  STD_LOGIC);
end XOR_Gate;
architecture Behavioral of XOR_Gate is
begin
F <= A XOR B;
end Behavioral;
```

Test Bench

CHAPTER FOUR
VHDL EXPERIEMENTS

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY NOR_Gate_test IS
END NOR_Gate_test;
ARCHITECTURE behavior OF NOR_Gate_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT XOR_Gate
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         F : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   --Outputs
   signal F : std_logic;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: XOR_Gate PORT MAP (
          A => A,
          B => B,
          F => F
        );
   -- Stimulus process
   stim_proc: process
   begin
     -- hold reset state for 50 ns.
     wait for 50 ns;
     -- insert stimulus here
     A <= '0';
     B <= '0';
     wait for 75 ns;
     A <= '0';
     B <= '1';
     wait for 75 ns;
     A <= '1';
     B <= '0';
     wait for 75 ns;
     A <= '1';
     B <= '1';
     wait;
  end process;
  END;
```


In figure 4.37 that shown the Signal and schematic of XOR gate.

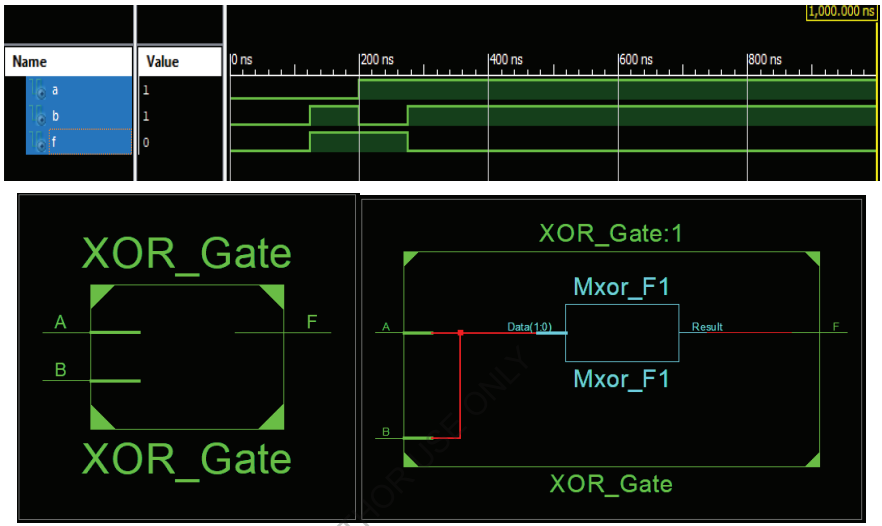


Figure 4.37: signal and Schematic of XOR Gate

CODE (XNOR Gate)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity XNOR_Gate is
    Port ( A,B : in  STD_LOGIC;
           F : out  STD_LOGIC);
end XNOR_Gate;
architecture Behavioral of XNOR_Gate is
begin
F <= A XNOR B;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY XNOR_Gate_test IS
END XNOR_Gate_test;
ARCHITECTURE behavior OF XNOR_Gate_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT XNOR_Gate
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         F : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   --Outputs
   signal F : std_logic;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: XNOR_Gate PORT MAP (
          A => A,
          B => B,
          F => F
        );
   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 50 ns.
      wait for 50 ns;
      -- insert stimulus here
      A <= '0';
      B <= '0';
      wait for 75 ns;
      A <= '0';
      B <= '1';
      wait for 75 ns;
      A <= '1';
      B <= '0';
      wait for 75 ns;
      A <= '1';
      B <= '1';
      wait;
  end process;
END;
```


In figure 4.38 that shows the Signal and schematic of XNOR gate.

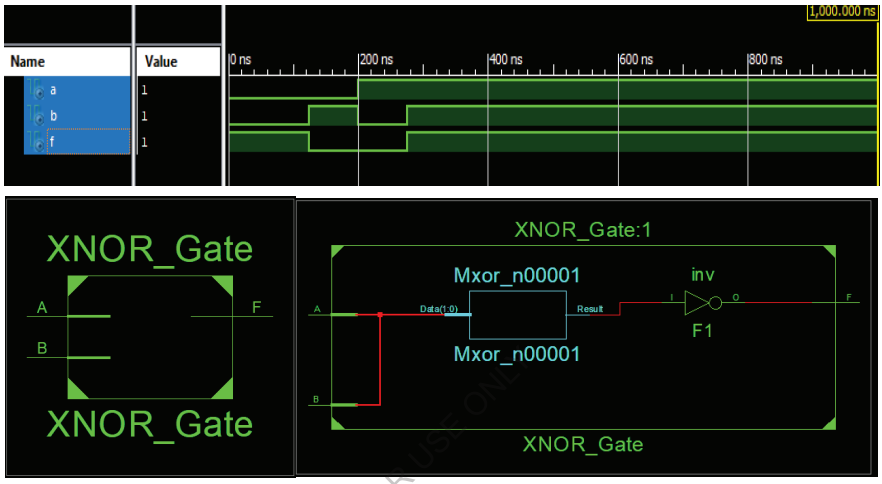


Figure 4.38: signal and Schematic of XNOR Gate

Discussion

1- Write and Simulate the 3-input XOR gate.

2- Write and Simulate the equation $\mathbf{Q = \overline{A}.B + A.\overline{B}}$.

3- Write and Simulate the 3-input XNOR gate.

4- Write and Simulate the equation $\mathbf{Q = (\overline{A}.\overline{B}) + (A.B)}$.

CHAPTER FOUR
VHDL EXPERIEMENTS

VHDL Design for any Boolean Function Written in Canonical

Aim of Experiment

In this experiment, we will learn the Canonical Boolean function with VHDL.

Theory

Boolean Function Representation

The use of **switching devices** like **transistors** give rise to a special case of the Boolean algebra called as **switching algebra**. In switching algebra, all the variables assume one of the two values which are **0** and **1**. In Boolean algebra, **0** is used to represent the '**open**' state or '**false**' state of logic gate. Similarly, **1** is used to represent the '**closed**' state or '**true**' state of logic gate. A **Boolean expression** is an expression which consists of variables, constants (**0-false** and **1-true**) and logical operators which results in true or false. A **Boolean function** is an algebraic form of **Boolean expression**. A Boolean function of **n-variables** is represented by f(x1, x2, x3….xn). By using **Boolean laws** and **theorems**, we can simplify the Boolean functions of digital circuits.

There are two types of canonical forms:

- Sum-of-min terms or Canonical **SOP**.
- Product-of- max terms or Canonical **POS**.

Boolean functions can be represented by using **NAND gates** and also by using **K-map** (Karnaugh map) method. We can standardize the Boolean expressions by using by two standard forms.

SOP form – Sum Of Products form

POS form – Product Of Sums form

Standardization of Boolean equations will make the implementation, evolution and simplification easier and more systematic.

Sum of Product (SOP) Form

The **sum-of-products (SOP)** form is a method (or form) of simplifying the Boolean expressions of logic gates. In this SOP form of Boolean function representation, the variables are operated by **AND** (product) to form a product term and all these product terms are **ORed** (summed or added) together to get the final function.

A sum-of-products form can be formed by adding (or summing) two or more product terms using a Boolean addition operation. Here the product terms are defined by using the AND operation and the sum term is defined by using OR operation.

The **sum-of-products form** is also called as **Disjunctive Normal Form** as the product terms are ORed together and Disjunction operation is logical OR. Sum-of-products form is also called as **Standard SOP**. SOP form representation is most suitable to use them in **FPGA** (Field Programmable Gate Arrays).

Examples

$$AB + ABC + CDE$$

$$(AB)^{-} + ABC + CD\,E^{-}$$

- SOP form can be obtained by Writing an AND term for each input combination, which produces HIGH output.
- Writing the input variables if the value is 1, and write the complement of the variable if its value is 0.
- OR the AND terms to obtain the output function.

Ex: Boolean expression for majority function F = A'BC + AB'C + ABC '+ ABC

Truth table:

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Now write the input variables combination with high output. **F = AB + BC + AC**.

Checking

By Idempotence law, we know that

$$([ABC + ABC)] + ABC) = (ABC + ABC) = ABC$$

Now the function F = A'BC + AB'C + ABC '+ ABC

$$= A'BC + AB'C + ABC' + ([ABC + ABC)] + ABC)$$

$$= (ABC + ABC') + (ABC + AB'C) + (ABC + A'BC)$$

$$= AB (C + C') + A (B + B') C + (A + A') BC$$

$$= AB + BC + AC.$$

Product of Sums (POS) Form

The product of sums form is a method (or form) of simplifying the Boolean expressions of logic gates. In this POS form, all the variables are ORed, i.e. written as sums to form sum terms.

All these sum terms are ANDed (multiplied) together to get the product-of-sum form. This form is exactly opposite to the SOP form. So, this can also be said as "Dual of SOP form". Here the sum terms are defined by using the OR operation and the product term is defined by using AND operation. When two or more sum terms are multiplied by a Boolean OR

operation, the resultant output expression will be in the form of product-of-sums form or POS form.

The product-of-sums form is also called as Conjunctive Normal Form as the sum terms are ANDed together and Conjunction operation is logical AND. Product-of-sums form is also called as Standard POS.

Examples

$$(A+B) * (A + B + C) * (C +D)$$

$$(A+B)^{-} * (C + D + E^{-})$$

POS form can be obtained by

- Writing an OR term for each input combination, which produces LOW output.
- Writing the input variables if the value is 0, and write the complement of the variable if its value is 1.
- AND the OR terms to obtain the output function.

Ex: Boolean expression for majority function F = (A + B + C) (A + B + C ‘) (A + B’ + C) (A’ + B + C).

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Now write the input variables combination with high output. F = AB + BC + AC.

Checking

By Idempotence law, we know that

$$[(A + B + C)\ (A + B + C)]\ (A + B + C) = [(A + B + C)]\ (A + B + C) = (A + B + C)$$

Now the function

$$F = (A + B)\ (B + C)\ (A + C)$$

$$= (A + B + C)\ (A + B + C\ ')\ (A + B' + C)\ (A' + B + C)$$

$$= [(A + B + C)\ (A + B + C)]\ (A + B + C)\ (A + B + C\ ')\ (A + B' + C)\ (A' + B + C)$$

$$= [(A + B + C)\ (A + B + C\ ')]\ [(A + B + C)\ (A' + B + C)]\ [(A + B + C)\ (A + B' + C)]$$

$$= [(A + B) + (C * C\ ')]\ [(B + C) + (A * A')]\ [(A + C) + (B * B')]$$

$$= [(A + B) + 0]\ [(B + C) + 0]\ [(A + C) + 0] = (A + B)\ (B + C)\ (A + C)$$

Canonical Form (Standard SOP and POS Form)

Any Boolean function that is expressed as a **sum of minterms** or as a **product of max terms** is said to be in its "**canonical form**".

It mainly involves in two Boolean terms, "**minterms**" and "**maxterms**".

When the **SOP** form of a Boolean expression is in canonical form, then each of its product term is called '**minterm**'. So, the canonical form of sum of products function is also known as "**minterm canonical form**" or Sum-of-minterms or standard canonical SOP form.

Similarly, when the **POS** form of a Boolean expression is in canonical form, then each of its sum term is called '**maxterm**'. So, the canonical form of product of sums function is also known as "**maxterm** canonical form or **Product-of sum** or standard canonical POS form".

Min terms

A **minterm** is defined as the product term of **n** variables, in which each of the **n** variables will appear once either in its **complemented** or **un-complemented** form. The min term is denoted as **mi** where **i** is in the range of $\mathbf{0 \leq i < 2^n}$.

A variable is in complemented form, if its value is assigned to **0**, and the variable is un-complimented form, if its value is assigned to **1**.

For a 2-variable (x and y) Boolean function, the possible minterms are:

x'y', x'y, xy' and xy.

For a 3-variable (x, y and z) Boolean function, the possible minterms are:

x'y'z', x'y'z, x'yz', x'yz, xy'z', xy'z, xyz' and xyz.

1 – Minterms = minterms for which the function F = 1.

0 – Minterms = minterms for which the function F = 0.

Any Boolean function can be expressed as the sum (OR) of its 1- min terms. The representation of the equation will be

F(list of variables) = Σ(list of 1-min term indices)

Ex: F (x, y, z) = Σ (3, 5, 6, 7)

The inverse of the function can be expressed as a sum (OR) of its 0- min terms. The representation of the equation will be

F(list of variables) = Σ(list of 0-min term indices)

Ex: F' (x, y, z) = Σ (0,1, 2, 4)

Examples of canonical form of sum of products expressions (min term canonical form):

i) Z = XY + XZ′

ii) F = XYZ′ + X′YZ + X′YZ′ + XY′Z + XYZ

In standard SOP form, the maximum possible product terms for n number of variables are given by 2^n. So, for 2 variable equations, the product terms are 22 = 4. Similarly, for 3 variable equations, the product terms are 23 = 8.

Max terms

A **max term** is defined as the product of n variables, within the range of $0 \leq i < 2^n$. The max term is denoted as Mi. In max term, each variable is complimented, if its value is assigned to 1, and each variable is un-complimented if its value is assigned to 0.

For a 2-variable (x and y) Boolean function, the possible max terms are:

x + y, x + y', x' + y and x' + y'.

For a 3-variable (x, y and z) Boolean function, the possible maxterms are:

x + y + z, x + y + z', x + y' + z, x + y' + z', x' + y + z, x' + y + z', x' + y' + z and x' + y' + z'.

1 – Max terms = max terms for which the function F = 1.

0 – max terms = max terms for which the function F = 0.

Any Boolean function can be expressed the product (AND) of its 0 – max terms. The representation of the equation will be

F(list of variables) = Π (list of 0-max term indices)

Ex: F (x, y, z) = Π (0, 1, 2, 4)

The inverse of the function can be expressed as a product (AND) of its 1 – max terms. The representation of the equation will be

F(list of variables) = Π (list of 1-max term indices)

Ex: F' (x, y, z) = Π (3, 5, 6, 7)

Examples of canonical form of product of sums expressions (max term canonical form):

i. Z = (X + Y) (X + Y′)

ii. F = (X′ + Y + Z′) (X′ + Y + Z) (X′ + Y′ + Z′)

In standard POS form, the maximum possible sum terms for n number of variables are given by 2^n. So, for 2 variable equations, the sum terms are 22 = 4. Similarly, for 3 variable equations, the sum terms are **23 = 8**.

Table for 2n min terms and 2n max terms

The below table will make you understand about the representation of the mean terms and max terms of 3 variables.

Variables			Min terms	Max terms
A	B	C	m_i	M_i
0	0	0	A' B' C' = m 0	A + B + C = M 0
0	0	1	A' B' C = m 1	A + B + C' = M 1
0	1	0	A' B C' = m 2	A + B' + C = M 2
0	1	1	A' B C = m 3	A + B' + C' = M 3
1	0	0	A B' C' = m 4	A' + B + C = M 4
1	0	1	A B' C = m 5	A' + B + C' = M 5
1	1	0	A B C' = m 6	A' + B' + C = M 6
1	1	1	A B C = m 7	A' + B' + C' = M 7

In table 4.14 that shown the truth table this is used in VHDL code.

Table 4.14: Truth Table for Two-1s Function F

(Decimal)	*A*	*B*	*C*	*F*
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

F (A, B, C) = ∑m (3, 5, 6)

Then write the canonical SOP (CSOP) form as:

$$F(A, B, C) = \bar{A}.B.C + A.\bar{B}.C + A.B.\bar{C}$$

The assignment statement for the 1s of the function F can now be written in CSOP form as:

F <= (not A and B and C) or (A and not B and C) or (A and B and not C)

Another way to write an assignment statement for the function F is to write a compact minterm form for the function using its 0s as:

$$F(A,B,C) = \sum m(0,1,2,4,7)$$

The canonical SOP (CSOP) form is written as:

$$\bar{F}\ (A,B,C) = \bar{A}.\bar{B}.\bar{C} + \bar{A}.\bar{B}.C + \bar{A}.B.\bar{C} + A.\bar{B}.\bar{C} + A.B.C$$

The assignment statement for the 0s of the function F can now be written in CSOP form as:

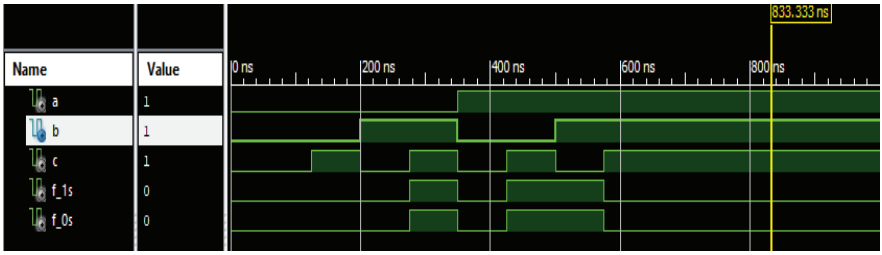
F <= not ((not A and not B and not C) or
(not A and not B and C) or
(not A and B and not C) or
(A and not B and not C) or
(A and B and C))

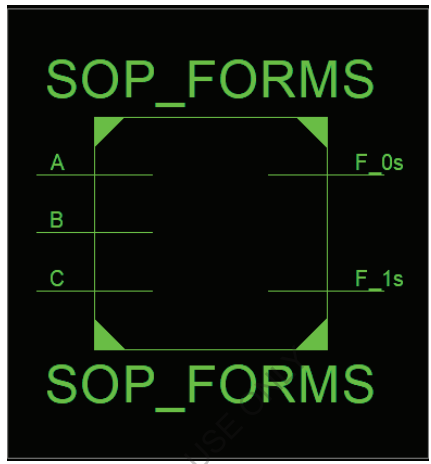
VHDL Code & Simulation

CODE (SOP)

Test Bench

In figure 4.39 that shown the Signal and schematic of SOP function.





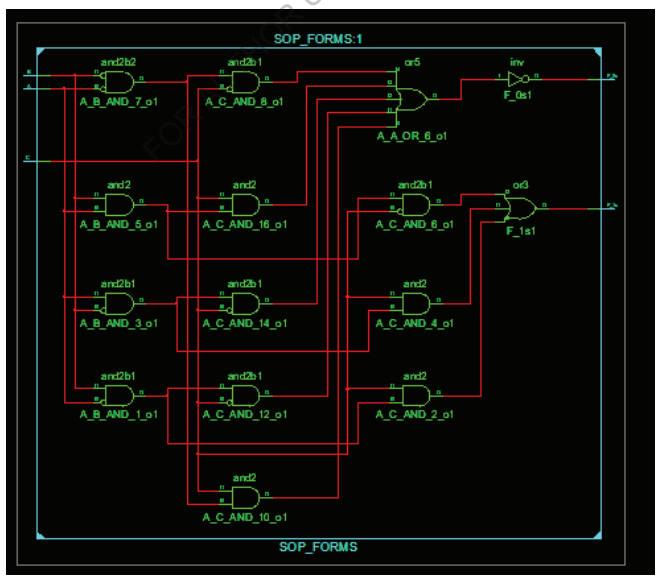


Figure 4.39: signal and Schematic of SOP Function

CHAPTER FOUR
VHDL EXPERIEMENTS

Discussion

1- Obtain the CPOS forms for the following Boolean functions expressed in compact maxterm form: (a) F1(X, Y) =∏M(0,1,2), (b) F2(Y) = ∏M(0), (c) F3(A,B) =∏ M(1,2).

2- Obtain the truth tables for the following Boolean functions expressed in compact minterm form: (a) F1(X, Y) = ∑m (2), (b) F2(Y) = ∑m(1), (c) F3(A,B) =∑m(0,1,2).

CHAPTER FOUR
VHDL EXPERIEMENTS

Half Adder Circuit

Aim of Experiment

In this experiment, we will learn the Half Adder with VHDL.

Theory

Half Adder Circuit

An **adder** is a digital circuit that performs addition of numbers. The **half adder** adds two binary digits called as augend and addend and produces two outputs as **sum** and **carry**; **XOR** is applied to both inputs to produce **sum** and **AND gate** is applied to both inputs to produce **carry**.

By using half adder, you can design simple addition with the help of logic gates.

Let's see an addition of single bits in figure 4.40.

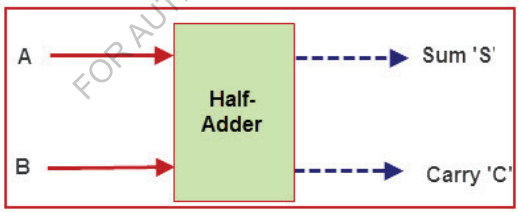


Figure 4.40: Half Adder

0+0 = 0

0+1 = 1

1+0 = 1

1+1 = 10

These are the least possible **single-bit combinations**. But the result for 1+1 is 10, the sum result must be re-written as a 2-bit output. Thus, the equations can be written as

0+0 = 00

0+1 = 01

1+0 = 01

1+1 = 10

The output '1'of '10' is carry-out. 'SUM' is the normal output and 'CARRY' is the carry-out.

Half Adder Truth Table

In table 4.15 that shown the half adder truth table.

Table 4.15: Half Truth Table

INPUTS		OUTPUTS	
A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Now it has been cleared that 1-bit adder can be easily implemented with the help of the XOR Gate for the output 'SUM' and an AND Gate for the 'Carry'. When we need to add, two 8-bit bytes together, we can be done with the help of a full-adder logic. The half-adder is useful when you want to add one binary digit quantities. A way to develop two-binary digit adders would be to make a **truth table and reduce it**. When you want to make a three binary digit adder, do it again. When you decide to make a four-digit adder, do it again. The circuits would be **fast**, but development time is **slow**. In figure 4.41 that shown the half adder logic circuit.

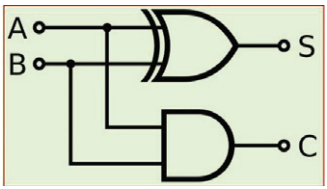


Figure 4.41: Half Adder Logic Circuit

The simplest expression uses the **exclusive OR** function: Sum=AÅB. An equivalent expression in terms of the basic AND, OR, and NOT is: SUM=A|.B+A.B'.

The "Std_Logic_Vector" Data Type

To represent a **group of signals**, VHDL uses **vector data types**. To access an element of a vector, we need to define an index. For example, assume that, as shown in Figure 4.42, we use a vector of length three, a_vec, to represent three values: val_0, val_1, and val_2. To access the value of an element from this vector, we can use the index numbers. For example, a_vec (2) will give the value of the rightmost element of the vector in Figure 4.42, which is val_2.

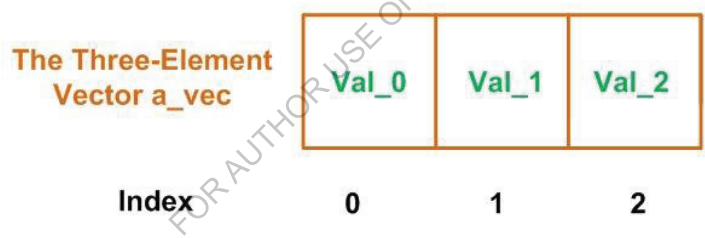


Figure 4.42: The three-element vector a_vec

The **VHDL** keyword "**std_logic_vector**" defines a vector of elements of type **std_logic**. For example, **std_logic_vector (0 to 2)** represents a three-element vector of std_logic data type, with the index range extending from 0 to 2.

Let's use the "**std_logic_vector**" data type to describe the circuit in Figure 4.43. We will use three vectors **a_vec, b_vec**, and **out_vec** to represent the blue, red, and black ports of Figure 4.43.

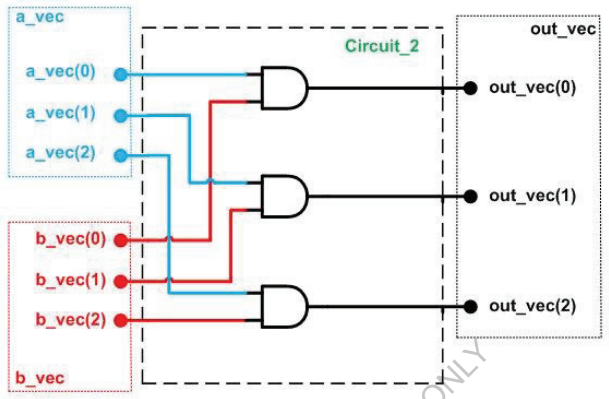


Figure 4.43: Vector Inputs (a, b)

The VHDL code for Figure 4.43 is given below.


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Vectors is
    Port ( a_vec,b_vec : in  STD_LOGIC_VECTOR(0 to 2);
           out_vec : out  STD_LOGIC_VECTOR(0 to 2));
end Vectors;
architecture Behavioral of Vectors is
begin
out_vec <= ( a_vec and b_vec );
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Vectors_test IS
END Vectors_test;
ARCHITECTURE behavior OF Vectors_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Vectors
 PORT(
      a_vec : IN  std_logic_vector(0 to 2);
      b_vec : IN  std_logic_vector(0 to 2);
      out_vec : OUT  std_logic_vector(0 to 2)
     );
 END COMPONENT;
--Inputs
signal a_vec : std_logic_vector(0 to 2) := (others => '0');
signal b_vec : std_logic_vector(0 to 2) := (others => '0');
--Outputs
signal out_vec : std_logic_vector(0 to 2);
-- No clocks detected in port list. Replace <clock> below with
-- appropriate port name
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Vectors PORT MAP (
       a_vec => a_vec,
       b_vec => b_vec,
       out_vec => out_vec
     );
-- Stimulus process

 stim_proc: process
 begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    -- insert stimulus here
    a_vec <= "000";
    b_vec <= "000";
    wait for 75 ns;
    a_vec <= "001";
    b_vec <= "010";
    wait for 75 ns;
    a_vec <= "011";
    b_vec <= "100";
    wait for 75 ns;
    a_vec <= "001";
    b_vec <= "011";
    wait;
 end process;
END;
```


In figure 4.44 that shown the waveform signal.

CHAPTER FOUR
VHDL EXPERIEMENTS

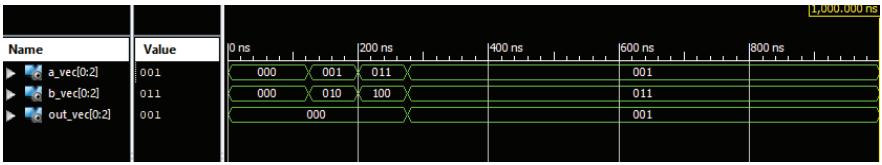


Figure 4.44: Waveform Signal

Interpreting Std_Logic_Vector Data

There is one important point which needs further attention: As shown in the above example, the "**std_logic_vector**" data type is a way to represent a group of signals or a data bus. It is simply a string of ones and zeros, and there is no other interpretation for this string of ones and zeros. In other words, if we assign "011" to a_vec, this doesn't mean that a_vec is equal to 3 (the decimal equivalent of "011").

We cannot assume a weight for the different bit positions of a "**std_logic_vector**" signal. However, we can use type conversion functions and type casting to interpret the string of ones and zeros in a given "**std_logic_vector**" signal as a number.

Ascending or Descending Index Range?

So far, we have used the "std_logic_vector" data type when defining input/output ports. Similarly, we can define a signal of "**std_logic_vector**" type. As an example, consider the following lines of code:

```
signal a: std_logic_vector(0 to 3);
...
a <= "0010"
```

Here, the first line defines a as a signal of type "**std_logic_vector**". The index ranges from 0 to 3. Then, "0010" is assigned to a. With this assignment, as shown in Figure 4.45, we will have a(0)=0, a(1)=0, a(2)=1, and a(3)=0.

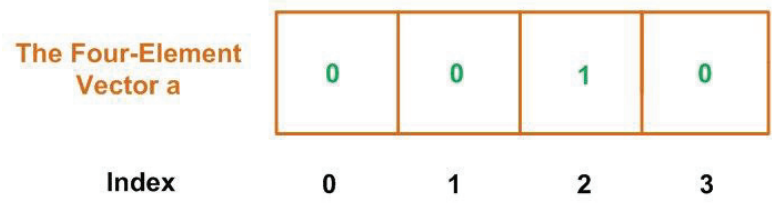


Figure 4.45: Ascending Index of Four Elements

The indexing style of this vector, which uses the keyword "to", is called **ascending**. We can also use the keyword "downto" (instead of "to") when we want a **descending** index range:

```
signal a: std_logic_vector(3 downto 0);
...
a <= "0010"
```

In this case, as shown in Figure 4.46, we'll have a(3)=0, a(2)=0, a(1)=1, and a(0)=0.

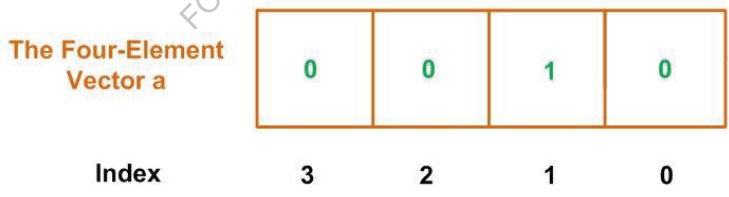


Figure 4.46: Descending Index of Four Elements

The choice between ascending and descending order is often a question of the designer's preferences, though it may be addressed by coding guidelines adopted by a particular organization. The most important thing is to choose one style and then follow it consistently; mixing the two different styles in one project can easily lead to trouble.

CHAPTER FOUR
VHDL EXPERIEMENTS

VHDL Code & Simulation

CODE (half adder by using dataflow)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Half_Adder is
    Port ( A,B : in  STD_LOGIC;
           S,C : out  STD_LOGIC);
end Half_Adder;
architecture Behavioral of Half_Adder is
begin
S <= A XOR B;
C <= A AND B;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Half_Adder_test IS
END Half_Adder_test;
ARCHITECTURE behavior OF Half_Adder_test IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT Half_Adder
    PORT(
         A : IN  std_logic;
         B : IN  std_logic;
         S : OUT  std_logic;
         C : OUT  std_logic
        );
    END COMPONENT;
   --Inputs
   signal A : std_logic := '0';
   signal B : std_logic := '0';
   --Outputs
   signal S : std_logic;
   signal C : std_logic;
BEGIN
   -- Instantiate the Unit Under Test (UUT)
   uut: Half_Adder PORT MAP (
          A => A,
          B => B,
          S => S,
          C => C
        );
```

```
  -- Stimulus process
  stim_proc: process
  begin
     -- hold reset state for 50 ns.
     wait for 50 ns;
     -- insert stimulus here
     A <= '0';
     B <= '0';
     wait for 75 ns;
     A <= '0';
     B <= '1';
     wait for 75 ns;
     A <= '1';
     B <= '0';
     wait for 75 ns;
     A <= '1';
     B <= '1';
     wait;
  end process;
END;
```


In figure 4.47 that shown the Signal and schematic of half adder circuit.

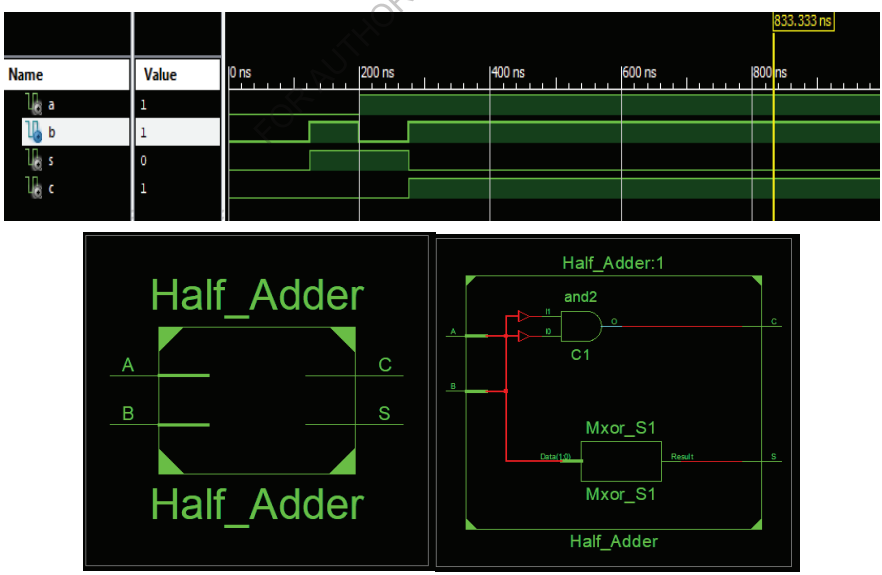


Figure 4.47: signal and Schematic of Half Adder Circuit

CHAPTER FOUR
VHDL EXPERIEMENTS

Half Adder using With-Select-When Statement (SSA)

CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Half_Adder_SSA is
    Port ( A : in  std_logic_vector(0 to 1);
           S,C : out  STD_LOGIC);
end Half_Adder_SSA;

architecture Behavioral of Half_Adder_SSA is

begin
with A select
S <= '0' when "00",
'0' when "11",
'1' when others;

with A select
C <= '1' when "11",
'0' when others;

end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY half_adder_tb IS
END half_adder_tb;

ARCHITECTURE half_adder_tb OF half_adder_tb IS

    COMPONENT halfadder6
    PORT(
         a : IN  std_logic_vector(0 to 1);
         s : OUT  std_logic;
         c : OUT  std_logic
        );
    END COMPONENT;

   signal a : std_logic_vector(0 to 1) := "00";
   signal s : std_logic;
   signal c : std_logic;
```

```
BEGIN

  uut: halfadder6 PORT MAP (
          a => a,
          s => s,
          c => c
        );

   stim_proc: process
   begin

  a <= "00";
      wait for 50 ns;

  a <= "01";
      wait for 50 ns;

  a <= "10";
      wait for 50 ns;

  a <= "11";
      wait;

  end process;

END;
```


In figure 4.48 that shown the Signal and schematic of half adder 2 bits circuit with SSA.

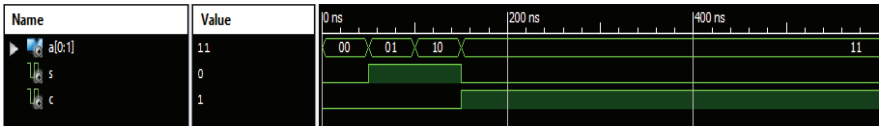


Figure 4.48: signal and Schematic of Half Adder 2 bits Circuit with SSA

Discussion

1- Write and simulate the half adder circuit for 3 bits and by using CSA method.

CHAPTER FOUR
VHDL EXPERIEMENTS

Full Adder Circuit

Aim of Experiment

In this experiment, we will learn the Full Adder with VHDL.

Theory

Full Adder Circuit

This **adder** is difficult to implement than a half-adder. The difference between a half-adder and a full-adder is that the **full-adder** has **three inputs** and **two outputs**, whereas **half adder** has only **two inputs** and **two outputs**. The first two inputs are A and B and the third input is an input carry as C-IN. When a full-adder logic is designed, you string eight of them together to create a byte-wide adder and cascade the carry bit from one adder to the next. In figure 4.49 that shown the full adder.

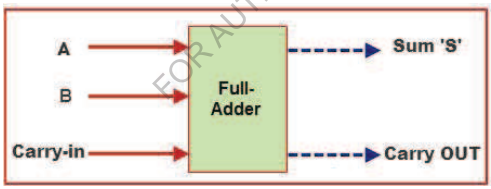


Figure 4.49: Full Adder

The output carry is designated as **C-OUT** and the normal output is designated as **S**.

Full Adder Truth Table

In table 4.16 that shown the full adder truth table.

Table 4.16: Full Adder Truth Table

INPUTS			OUTPUT	
A	B	C-IN	C-OUT	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

With the truth-table, the full adder logic can be implemented. You can see that the output **S** is an **XOR** between the input **A** and the **half-adder**, **SUM** output with **B** and **C-IN** inputs. We take **C-OUT** will only be true if any of the two inputs out of the three are HIGH. So, we can implement a **full adder** circuit with the help of **two half adder circuits**. At first, half adder will be used to add A and B to produce a partial Sum and a second half adder logic can be used to add **C-IN** to the Sum produced by the first half adder to get the final S output, as shown in figure 4.50.

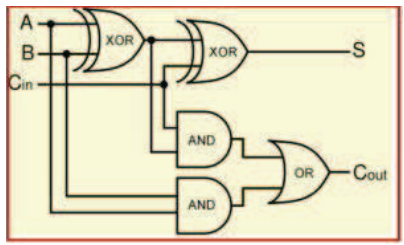


Figure 4.50: Full Adder Logic Circuit

If any of the half adder logic produces a carry, there will be an output carry. So, COUT will be an OR function of the half-adder Carry outputs. Take a look at the implementation of the full adder circuit shown below in figure 4.51.

The implementation of larger logic diagrams is possible with the above full adder logic a simpler symbol is mostly used to represent the operation. Given below in figure 4.51 is a simpler schematic representation of a one-bit full adder.

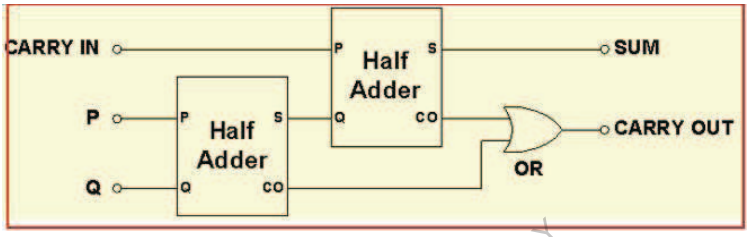


Figure 4.51: Full Adder Design Using Half Adders

VHDL Code & Simulation

CODE (Full adder by using dataflow)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Full_Adder is
    Port ( A,B,C : in  STD_LOGIC;
           sum,carry : out  STD_LOGIC);
end Full_Adder;
architecture Behavioral of Full_Adder is
begin
sum <= A xor B xor C;
carry <= (A and B) or ((A xor B) and C);
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Full_Adder_test IS
END Full_Adder_test;
ARCHITECTURE behavior OF Full_Adder_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Full_Adder
 PORT(
      A : IN  std_logic;
      B : IN  std_logic;
      C : IN  std_logic;
      sum : OUT  std_logic;
      carry : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal A : std_logic := '0';
signal B : std_logic := '0';
signal C : std_logic := '0';
--Outputs
signal sum : std_logic;
signal carry : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Full_Adder PORT MAP (
       A => A,
       B => B,
       C => C,
         sum => sum,
         carry => carry
       );
 -- Stimulus process
 stim_proc: process
 begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    -- insert stimulus here
    A <= '0';
    B <= '0';
    C <= '0';
wait for 75 ns;
    A <= '0';
    B <= '0';
    C <= '1';
wait for 75 ns;
    A <= '0';
    B <= '1';
    C <= '0';
wait for 75 ns;
    A <= '0';
    B <= '1';
    C <= '1';
wait for 75 ns;
    A <= '1';
    B <= '0';
    C <= '0';
wait for 75 ns;
    A <= '1';
    B <= '0';
    C <= '1';
wait for 75 ns;
    A <= '1';
    B <= '1';
    C <= '0';
wait for 75 ns;
    A <= '1';
    B <= '1';
    C <= '1';
    wait;
 end process;
 END;
```

In figure 4.52 that shown the Signal and schematic of full adder circuit.

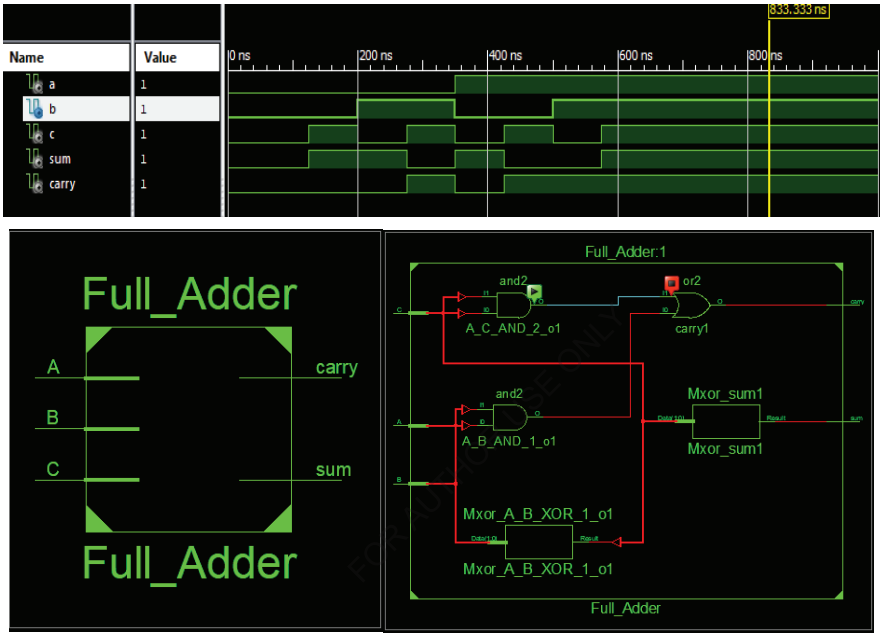


Figure 4.52: Signal and Schematic of Full Adder Circuit

CODE (Full adder by using Vector 3 bits)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Full_Adder_Vector is
    Port ( A,B,C : in  STD_LOGIC_VECTOR(2 downto 0);
           sum,carry : out  STD_LOGIC_VECTOR (2 downto 0));
end Full_Adder_Vector;
architecture Behavioral of Full_Adder_Vector is
begin
sum <= A XOR B XOR C;
carry <= (A and B) or ((A xor B) and C);
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Full_Adder_Vector_test IS
END Full_Adder_Vector_test;
ARCHITECTURE behavior OF Full_Adder_Vector_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Full_Adder_Vector
 PORT(
      A : IN  std_logic_vector(2 downto 0);
      B : IN  std_logic_vector(2 downto 0);
      C : IN  std_logic_vector(2 downto 0);
      sum : OUT  std_logic_vector(2 downto 0);
      carry : OUT  std_logic_vector(2 downto 0)
     );
 END COMPONENT;
--Inputs
signal A : std_logic_vector(2 downto 0) := (others => '0');
signal B : std_logic_vector(2 downto 0) := (others => '0');
signal C : std_logic_vector(2 downto 0) := (others => '0');
--Outputs
signal sum : std_logic_vector(2 downto 0);
signal carry : std_logic_vector(2 downto 0);
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Full_Adder_Vector PORT MAP (
       A => A,
       B => B,
       C => C,

       sum => sum,
       carry => carry
     );
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
   A <= "000";
   B <= "000";
   C <= "000";
   wait for 75 ns;
   A <= "000";
   B <= "000";
   C <= "001";
   wait for 75 ns;
   A <= "000";
   B <= "001";
   C <= "001";
   wait for 75 ns;
   A <= "000";
   B <= "011";
   C <= "000";
   wait for 75 ns;
   A <= "100";
   B <= "000";
   C <= "100";
   wait for 75 ns;
   A <= "101";
   B <= "000";
   C <= "001";
   wait for 75 ns;
   A <= "001";
   B <= "100";
   C <= "010";
   wait for 75 ns;
   A <= "001";
   B <= "001";
   C <= "001";
   wait;
 end process;
END;
```

In figure 4.53 that shown the Signal and schematic of full adder circuit by using vector 3 bits.

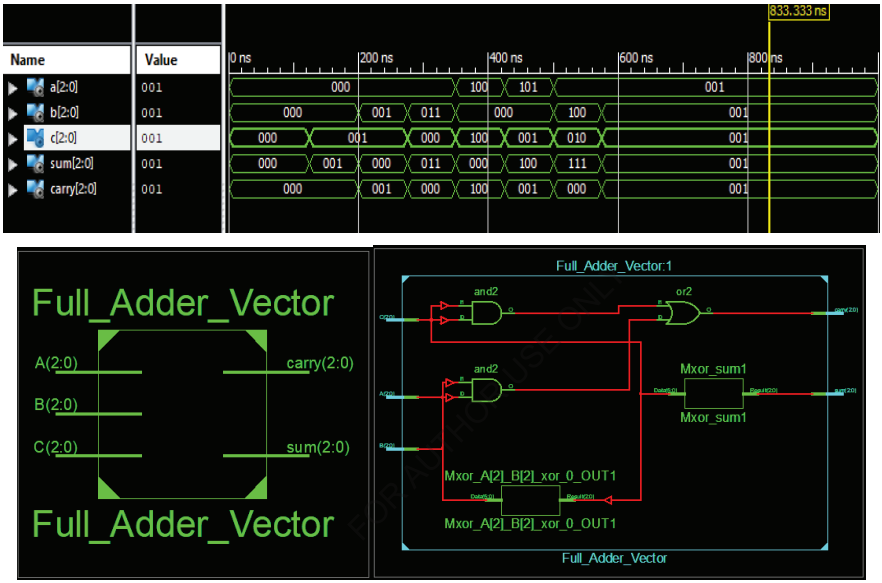


Figure 4.53: Signal and Schematic of Full Adder Circuit by using vector 3 bits

Discussion

1- Write and simulate the half adder circuit for 3 bits and by using SSA method.

CHAPTER FOUR
VHDL EXPERIEMENTS

Addition in VHDL

Aim of Experiment

In this experiment, we will learn the addition in VHDL.

Theory

Addition Library

The **STD_LOGIC_VECTOR** data type can be used in **addition** and **subtraction** operations (+ and -) if the **STD_LOGIC_SIGNED** or the **STD_LOGIC_UNSIGNED** package of the IEEE library is used.

(Otherwise the arithmetic operators can only be used with INTEGER, SIGNED and UNSIGNED data types).

NUMERIC_STD

The **NUMERIC_STD** package introduces two new types: that of the **UNSIGNED** and **SIGNED** arrays of **std_logic**. Recall that **std_logic_vector** is merely a pattern with no positional meaning given to the bits. Both the UNSIGNED and SIGNED data types do assume a positional meaning with the least significant position (slice 0) representing the 1's place.

VHDL Code & Simulation

CODE (Unsigned Library)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity Addition_2Num is
    Port ( NUM1,NUM2 : in  STD_LOGIC_VECTOR (4 downto 0):= "00000";
           sum : out  STD_LOGIC_VECTOR (4 downto 0));
end Addition_2Num;
architecture Behavioral of Addition_2Num is
begin
 Sum <= NUM1 + NUM2;
end Behavioral;
```

CHAPTER FOUR
VHDL EXPERIEMENTS

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Addition_2Num_test IS
END Addition_2Num_test;
ARCHITECTURE behavior OF Addition_2Num_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Addition_2Num
 PORT(
      NUM1 : IN  std_logic_vector(4 downto 0);
      NUM2 : IN  std_logic_vector(4 downto 0);
      sum : OUT  std_logic_vector(4 downto 0)
     );
 END COMPONENT;
--Inputs
signal NUM1 : std_logic_vector(4 downto 0) := (others => '0');
signal NUM2 : std_logic_vector(4 downto 0) := (others => '0');
--Outputs
signal sum : std_logic_vector(4 downto 0);
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Addition_2Num PORT MAP (
       NUM1 => NUM1,
       NUM2 => NUM2,
       sum => sum
     );
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
   NUM1 <= "00001";
   NUM2 <= "00001";
   wait for 75 ns;
   NUM1 <= "00011";
   NUM2 <= "00011";
   wait for 75 ns;
   NUM1 <= "00100";
   NUM2 <= "00101";
   wait for 75 ns;
   NUM1 <= "00010";
   NUM2 <= "00011";
   wait for 75 ns;
   NUM1 <= "10000";
   NUM2 <= "00000";
   wait;
 end process;
END;
```

In figure 4.54 that shown the Signal and schematic of addition two numbers.

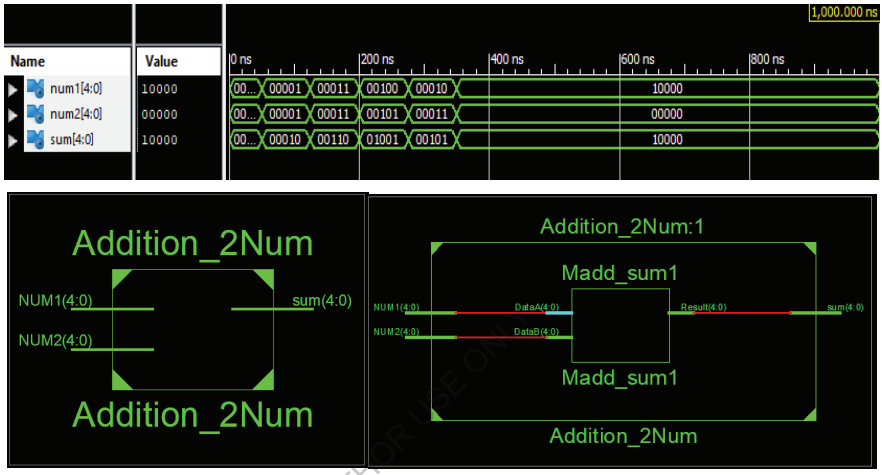


Figure 4.54: Signal and Schematic of Addition Two Numbers

CODE (Numeric Library)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Addition_Numeric is
    Port ( A,B : in  STD_LOGIC_VECTOR (7 downto 0);
           result1,result2,result3 : out  STD_LOGIC_VECTOR (7 downto 0));
end Addition_Numeric;
architecture Behavioral of Addition_Numeric is
begin
  -- this is allowed
    result1 <= std_logic_vector(unsigned(a) + unsigned(b));

    -- this is allowed
    result2 <= std_logic_vector(signed(a) + signed(b));

    -- this is NOT ALLOWED
    result3 <= std_logic_vector(unsigned(a) + signed(b));
end Behavioral;
```

Discussion

1- Write and simulate the addition for 6 bits and by signed library.

CHAPTER FOUR
VHDL EXPERIEMENTS

Half Subtract

Aim of Experiment

In this experiment, we will learn the half subtract in VHDL.

Theory

Designing of Half Subtractor

A **Subtractor** is a digital circuit which performs subtraction operation.

Half Subtractor

It is a combinational circuit that performs subtraction of two binary bits. It has two inputs (minuend and subtrahend) and two outputs Difference (**D**) and Borrows (**Bout**). We use half-subtractor to subtract the **LSB** of the subtrahend to the **LSB** of the minuend when one binary number is subtracted from another. **Subtraction** is done according to the rule of binary subtraction and the operations can be summarized in a truth table as table 4.17,

Table 4.17: Half Subtractor Truth Table

A	B	Difference (D)	Borrow (B_{out})
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

K-Map Simplification: We use **K-Map** to obtain the expression for Difference and Borrow bit which is as shown in figure 4.55.

CHAPTER FOUR
VHDL EXPERIEMENTS

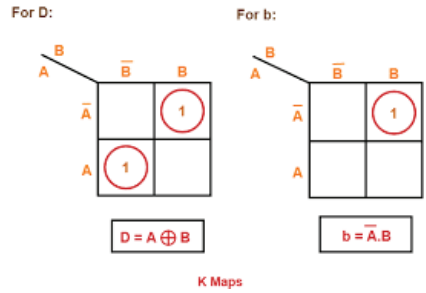


Figure 4.55: K-map Half Subtractor

On solving the **K-Map** and getting the simplified Boolean Expressions we can observe that Boolean expression for Difference (**D**) is the same as the XOR operation and which is the same as what we get as the expression of Sum in **Half Adder** and Boolean expression for Borrow (**Bout**) is A.B

Hence, Logic circuit diagram for Half-Adder can be drawn as shown in figure 4.56.

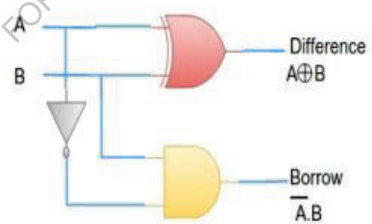


Figure 4.56: Half Subtractor Logic Circuit

Process Statement

We now turn our attention to a the VHDL process statement. The process is the key structure in behavioral VHDL modeling. A process is the only means by which the executable functionality of a component is defined. In fact, for a model to be capable of

being simulated, all components in the model must be defined using one or more processes.

Statements within a process are executed sequentially (although care needs to be used in signal assignment statements since they do not take effect immediately; this was covered in the VHDL Basics module when the VHDL timing model was discussed). Variables are used as internal place holders which take on their assigned values immediately.

All processes in a VHDL description are executed concurrently. That is, although statements within a process are evaluated and executed sequentially, all processes within the model begin executing concurrently.

Simplified Syntax

[process_label:] **process** [(sensitivity_list)] [**is**] (*optional)
process_declarations
begin
sequential_statements
end process [process_label] ;

If Statement

The **if statement** is a statement that depending on the value of one or more corresponding conditions, selects for execution one or none of the enclosed sequences of statements.

Simplified Syntax

if condition then
sequential_statements
end if;
if condition then
sequential_statements
else

sequential_statements

end if;

if condition then

sequential_statements

elsif condition then

sequential_statements

else

sequential_statements

end if;

Description

The **if statement** controls conditional execution of other **sequential statements**. It contains at least one Boolean condition (specified after the if keyword). The remaining conditions are specified with the **elsif clause**. The else clause is treated as elsif true then. Conditions are evaluated one by one until any of them turns to be true or there are no more conditions to be checked for. When a condition is true then the sequence of statements specified after the then clause is executed. If no condition is met then the control is passed to the next statement after the if statement.

VHDL Code & Simulation

CODE (dataflow Module)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Half_Sub_dataflow is
    Port ( A,B : in  STD_LOGIC;
           diff,borrow : out  STD_LOGIC);
end Half_Sub_dataflow;
architecture Behavioral of Half_Sub_dataflow is
begin
diff <= A xor B;
borrow <= (not A) and B;
end Behavioral;
```

CHAPTER FOUR
VHDL EXPERIEMENTS

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Half_Sub_dataflow_test IS
END Half_Sub_dataflow_test;
ARCHITECTURE behavior OF Half_Sub_dataflow_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Half_Sub_dataflow
 PORT(
      A : IN  std_logic;
      B : IN  std_logic;
      diff : OUT  std_logic;
      borrow : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal A : std_logic := '0';
signal B : std_logic := '0';
--Outputs
signal diff : std_logic;
signal borrow : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Half_Sub_dataflow PORT MAP (
       A => A,
       B => B,
       diff => diff,
       borrow => borrow
     );

-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
   A <= '0';
   B <= '0';
   wait for 75 ns;
   A <= '0';
   B <= '1';
   wait for 75 ns;
   A <= '1';
   B <= '0';
   wait for 75 ns;
   A <= '1';
   B <= '1';
   wait;
end process;
END;
```

In figure 4.57 that shown the Signal of half subtractor by using dataflow.

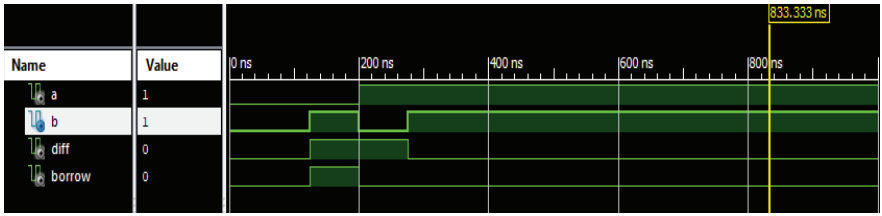


Figure 4.57: Signal of half subtractor by using dataflow

CODE (Behavioral Module)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Half_Sub_if is
    Port ( A,B : in  STD_LOGIC;
           diff,borrow : out  STD_LOGIC);
end Half_Sub_if;
architecture Behavioral of Half_Sub_if is
begin
process (A,B)
begin
if (A = '0' AND B = '0') OR (A = '1' AND B = '1')  then
diff <= '0'; borrow <='0';
elsif (A = '0' and B = '1')then
diff <= '1'; borrow <='1';
else diff <= '1'; borrow <='0';
end if;
end process;
end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Half_Sub_if_test IS
END Half_Sub_if_test;
ARCHITECTURE behavior OF Half_Sub_if_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Half_Sub_if
 PORT(
      A : IN  std_logic;
      B : IN  std_logic;
      diff : OUT  std_logic;
      borrow : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal A : std_logic := '0';
signal B : std_logic := '0';
--Outputs
signal diff : std_logic;
signal borrow : std_logic;
BEGIN
 -- Instantiate the Unit Under Test (UUT)
 uut: Half_Sub_if PORT MAP (
       A => A,
       B => B,
       diff => diff,
       borrow => borrow
     );
 -- Stimulus process
 stim_proc: process
 begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    -- insert stimulus here
    A<= '0';
    B<= '0';
    wait for 75 ns;
    A<= '0';
    B<= '1';
    wait for 75 ns;
    A<= '1';
    B<= '0';
    wait for 75 ns;
    A<= '1';
    B<= '1';
    wait;
 end process;
 END;
```


In figure 4.58 that shown the Signal and schematic of half subtractor by using behavioral.

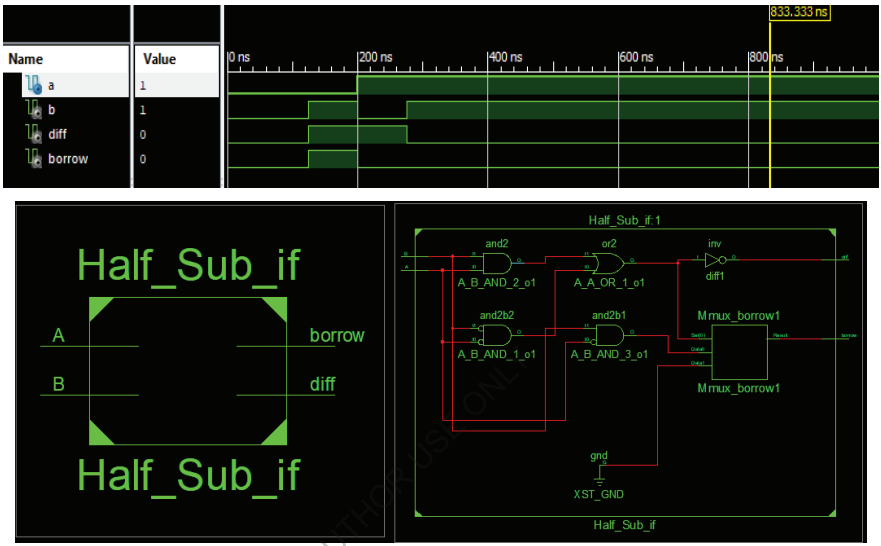


Figure 4.58: Signal and Schematic of half subtractor by using behavioral

Discussion

1- Write and simulate the half subtractor for vector 4 bits.

CHAPTER FOUR
VHDL EXPERIEMENTS

Full Subtract

Aim of Experiment

In this experiment, we will learn the full subtract in VHDL.

Theory

Designing of full Subtractor

Full Subtractor also belongs to the class of a combinational circuit and is used to perform subtraction of two binary bits. The **half-subtractor** can only be used for subtraction of **LSB** bits, but if there occurs a case of borrow during subtraction of **LSB** bits, then it can have affected over subtraction in higher columns. Thus, we have a total of three inputs (two original bits and third is considering the borrow (**Bin**) of the previous stage) and two outputs Difference (**D**) and Borrow (**Bout**) respectively. The Truth Table of Full Subtractor can be written as table 4.18.

Table 4.18: Full Subtractor Truth Table

A	B	B_{in}	Difference (D)	Borrow (B_{out})
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

K-Map Simplification: We use K-Map to obtain the expression for Difference and Borrow bit which is as shown in figure 2.59.

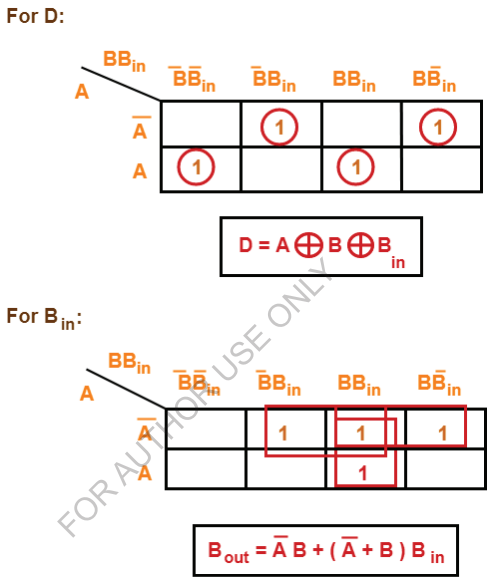


Figure 4.59: Full Subtractor K-map

After solving K-Map, simplified Boolean Expressions for Difference is **A ⊕ B ⊕ Bin** and for Borrow it is **$\bar{A}$.B +$\bar{A}$.Bin + B. Bin**, thus, logic circuit diagram for full-subtractor can be drawn as shown in figure 4.60.

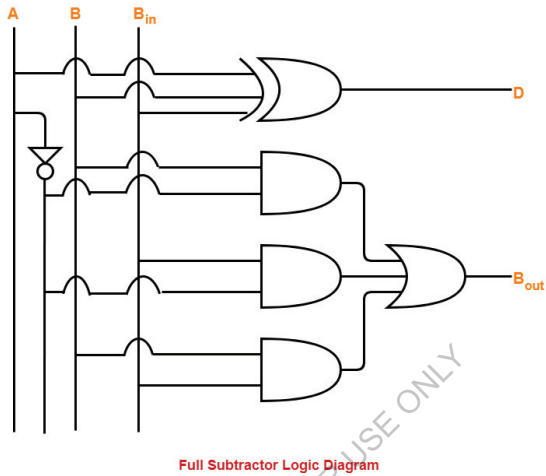


Figure 4.60: Full Subtractor Logic Diagram

Designing of Full Subtractor using Half-Subtractors

A Full-Subtractor can also be implemented using two half-subtractors and one OR gate. The circuit diagram for this can be drawn as shown in figure 4.61.

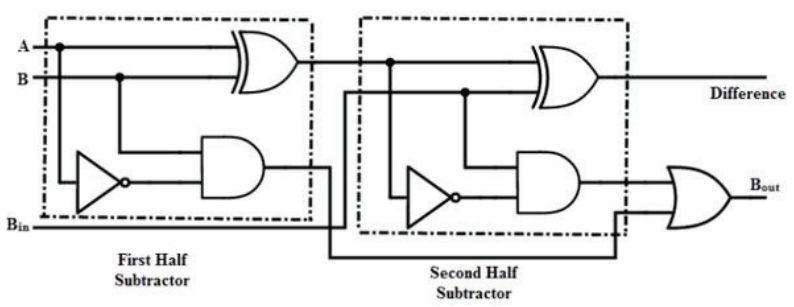


Figure 4.61: Full Subtractor by Two Half Subtractor

Case Statement

Formal Definition

The case statement selects for execution one of several alternative sequences of statements; the alternative is chosen based on the value of the associated expression.

The VHDL **Case Statement** works exactly the way that a switch statement in C works. Given an input, the statement looks at each possible condition to find one that the input signal satisfies. They are useful to check one input signal against many combinations.

Just like in C, the VHDL designer should always specify a default condition provided that none of the case statements are chosen. This is done via the "when others =>" statement.

One annoyance with case statements is that VHDL does not allow the use of less than or greater than relational operators in the "when" condition. Only values that are equal to the signal in the case test can be used.

A note about synthesis: When case statements are synthesized by the tools, they generate optimized decode logic to quickly select which case statement is valid. They are more efficient than using **if/elsif** statements because **if/elsif** can generate long carry-chains of logic that can cause difficulties meeting timing.

Simplified Syntax

```
case expression is
    when choice => sequential_statements
    when choice => sequential_statements
    . . .
    end case;
```

Important Notes

- The **case expression** must be of a discrete type or of a **one-dimensional array** type, whose element type is a **character type**.

- Every possible value of the case expression must be covered by the specified alternatives; moreover, every value may appear only once (no duplicates or overlapping of ranges is allowed).
- The **When others** clause may appear only once and only as the very last choice.

VHDL Code & Simulation

CODE (dataflow Module)

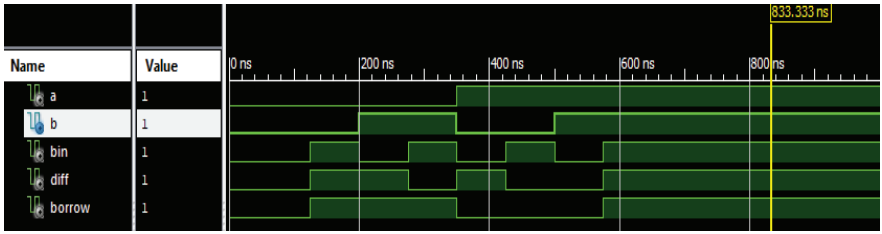
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Full_Sub_dataflow is
    Port ( A,B,Bin : in  STD_LOGIC;
           diff,borrow : out  STD_LOGIC);
end Full_Sub_dataflow;
architecture Behavioral of Full_Sub_dataflow is
begin
diff <= A xor B xor Bin;
borrow <= ((not A) and B) or ((not A or B) and Bin);
end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Full_Sub_dataflow_test IS
END Full_Sub_dataflow_test;
ARCHITECTURE behavior OF Full_Sub_dataflow_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Full_Sub_dataflow
 PORT(
      A : IN  std_logic;
      B : IN  std_logic;
      Bin : IN  std_logic;
      diff : OUT  std_logic;
      borrow : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal A : std_logic := '0';
signal B : std_logic := '0';
signal Bin : std_logic := '0';
--Outputs
signal diff : std_logic;
signal borrow : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Full_Sub_dataflow PORT MAP (
       A => A,
       B => B,
       Bin => Bin,
```

```
        diff => diff,
        borrow => borrow
      );
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
A <= '0';
B <= '0';
Bin <= '0';
wait for 75 ns;
A <= '0';
B <= '0';
Bin <= '1';
wait for 75 ns;
A <= '0';
B <= '1';
Bin <= '0';
wait for 75 ns;
A <= '0';
B <= '1';
Bin <= '1';
wait for 75 ns;
A <= '1';
B <= '0';
Bin <= '0';
wait for 75 ns;
A <= '1';
B <= '0';
Bin <= '1';
wait for 75 ns;
A <= '1';
B <= '1';
Bin <= '0';
wait for 75 ns;
A <= '1';
B <= '1';
Bin <= '1';
   wait;
end process;
END;
```


In figure 4.62 that shown the Signal and schematic of full subtractor by using dataflow.



CHAPTER FOUR
VHDL EXPERIEMENTS

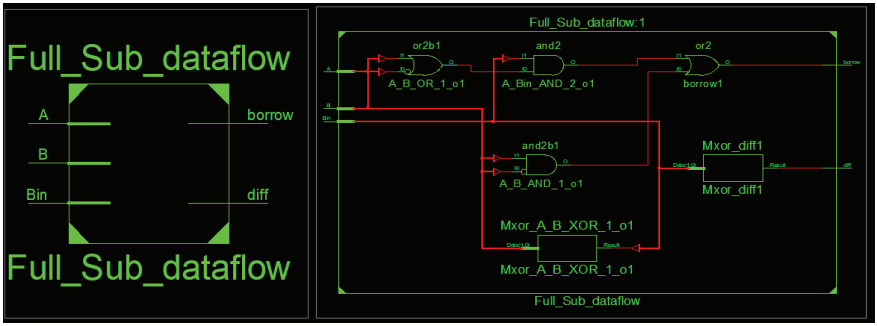


Figure 4.62: Signal and Schematic of full subtractor by using dataflow

CODE (Behavioral Module)


```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Full_Sub_case is
    Port ( A,B,Bin : in  STD_LOGIC;
           diff,borrow : out  STD_LOGIC);
end Full_Sub_case;
architecture Behavioral of Full_Sub_case is
signal X1: STD_LOGIC_VECTOR(2 DOWNTO 0);
begin
X1 <= (A,B,Bin);
p: process (X1)
begin
case X1 is
   when "000" => diff <= '0'; borrow <= '0';
   when "001" => diff <= '1'; borrow <= '1';
   when "010" => diff <= '1'; borrow <= '1';
   when "011" => diff <= '0'; borrow <= '1';
   when "100" => diff <= '1'; borrow <= '0';
   when "101" => diff <= '0'; borrow <= '0';
   when "110" => diff <= '0'; borrow <= '0';
   when "111" => diff <= '1'; borrow <= '1';
   when others => null;
   end case;
end process;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Full_Sub_case_test IS
END Full_Sub_case_test;
ARCHITECTURE behavior OF Full_Sub_case_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Full_Sub_case
 PORT(
      A : IN  std_logic;
      B : IN  std_logic;
      Bin : IN  std_logic;
      diff : OUT  std_logic;
      borrow : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal A : std_logic := '0';
signal B : std_logic := '0';
signal Bin : std_logic := '0';
--Outputs
signal diff : std_logic;
signal borrow : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Full_Sub_case PORT MAP (
       A => A,
       B => B,
       Bin => Bin,
       diff => diff,
       borrow => borrow
     );
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
A <= '0';
B <= '0';
Bin <= '0';
wait for 80 ns;
A <= '0';
B <= '0';
Bin <= '1';
wait for 80 ns;
A <= '0';
B <= '1';
Bin <= '0';
wait for 80 ns;
A <= '0';
B <= '1';
Bin <= '1';
wait for 80 ns;
A <= '1';
B <= '0';
Bin <= '0';
wait for 80 ns;
A <= '1';
B <= '0';
Bin <= '1';
wait for 80 ns;
A <= '1';
B <= '1';
Bin <= '0';
wait for 80 ns;
A <= '1';
B <= '1';
Bin <= '1';
   wait;
end process;
END;
```

In figure 4.63 that shown the Signal and schematic of full subtractor by using case statement.

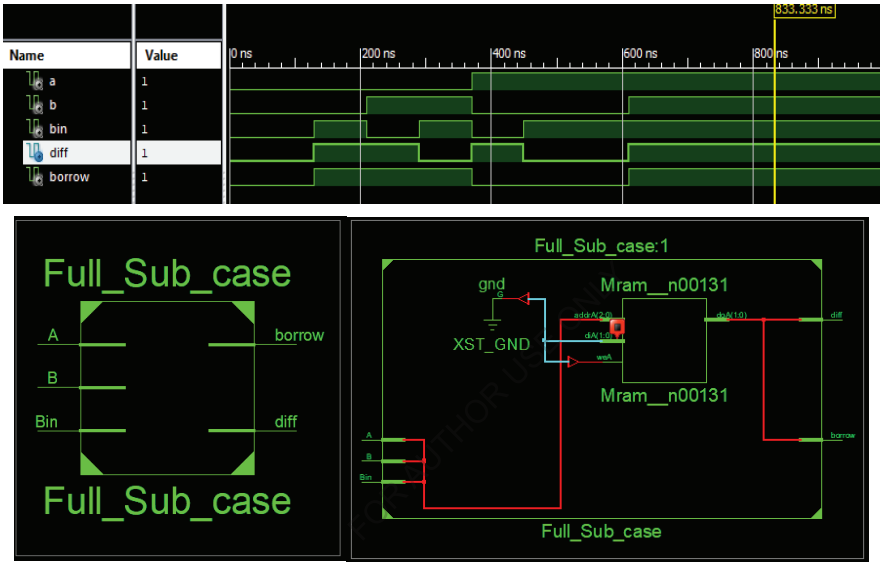


Figure 4.63: Signal and Schematic of full subtractor by using case statement

- Rather than declaring a bus in the entity for signal A as shown in the code, individual signals A, B and Bin can be declared and then grouped together to form a bus using an expression called an **aggregate**. An aggregate is a collection or group of elements. Forming a bus with an aggregate is a handy concept to know. The placement of the elements (or the order of the elements) in the aggregate is very important. The left-most element in the list is the **MSB**, while the right-most element in the list is the **LSB**. An aggregate can also be used to form a larger bus using the individual elements of the bus to form the aggregate. Do not use a vector in an aggregate; however, you may use the individual elements to form a larger bus with a larger range. Suppose we

wanted to form the bus A(4:0). If some of the values of the individual elements of the bus [say, A(4) and A(3)] must be added to A(2 downto 0) to form the larger bus then the correct way to write the aggregate is A <= (A(4), A(3), A(2), A(1), A(0)). Writing the aggregate as A <= (A(4), A(3), A(2 downto 0)) would be a syntax error. If you want to form a larger bus with vectors you may use the concatenation operator "&" as follows: A <= (A(4) & A(3) & A(2 downto 0)). Just like the aggregate, the order in which the elements are placed in the list is very important—that is, the left-most element in the list is the MSB and the right-most element in the list is the LSB.

- Signal statement that is discuss in the next experiment.

Discussion

1- Write and simulate the full subtractor for vector 4 bits.

2- Write and simulate the full subtractor by using if statement.

CHAPTER FOUR
VHDL EXPERIEMENTS

Internal Signal

Aim of Experiment

In this experiment, we will learn the internal signal in VHDL.

Theory

Internal Signal

Signals represent wires or outputs of gates, FFs, etc. Ports (ins, outs, inouts) in the entity are **signals**. **Internal signals** are often needed in complex models and are declared in the architecture description as follows:

```
architecture architecture_name of entity_name is
signal signal_name: type;
:
:
signal signal_name: type;
begin
:
end architecture architecture_name;
```


The signal type can be **bit**, **bit_vector**, **std_logic**, or **std_logic_vector**.

Signals can be initialized to a beginning value at the declaration BUT this is meaningless to synthesis tools since no hardware mechanism exists to produce this “power-up” init value. In figure 4.64 that shown the example of internal signal.

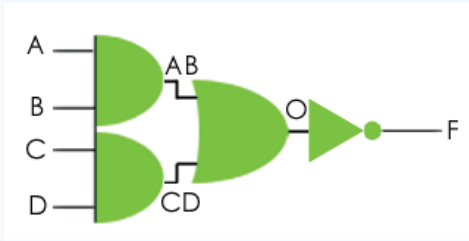


Figure 4.64: Internal Signal Example

The internal signal of the figure 4.64:

```
AB = A AND B;
CD = C AND D;
O = AB OR CD;
```


Signals

The architecture contains three signals **AB**, **CD** and **O**, used internally within the architecture. A signal is declared before the begin, of an architecture, and has its own data type (eg. **STD_LOGIC**). Technically, ports are signals, so signals and ports are read and assigned in the same way.

Assignments

The assignments within the architecture are concurrent signal assignments. Such assignments execute whenever a signal on the right-hand side of the assignment changes value. Because of this, the order in which concurrent assignments are written has no effect on their execution. The assignments are concurrent because potentially two assignments could execute at the same time (if two inputs changed simultaneously). The style of description that uses only concurrent assignments is sometimes termed dataflow.

Delays

Each of the concurrent signal assignments has a delay. The expression on the right-hand side is evaluated whenever a signal on the right-hand side changes value, and the signal

on the left-hand side of the assignment is updated with the new value after the given delay. In this case, a change on the port **A** would propagate through the circuit entity to the port **F**, with a total delay of **5 NS**.

VHDL Code & Simulation

CODE (figure 4.64)

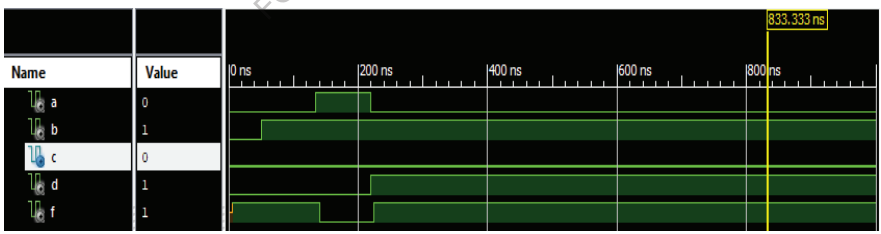
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Internal_Signal is
    Port ( A,B,C,D : in  STD_LOGIC;
           F : out  STD_LOGIC);
end Internal_Signal;
architecture Behavioral of Internal_Signal is
signal AB, CD, O: STD_LOGIC;
begin
 AB <= A and B after 2 NS;
  CD <= C and D after 2 NS;
  O <= AB or CD after 2 NS;
  F <= not O after 1 NS;
end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Internal_Signal_test IS
END Internal_Signal_test;
ARCHITECTURE behavior OF Internal_Signal_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Internal_Signal
 PORT(
      A : IN  std_logic;
      B : IN  std_logic;
      C : IN  std_logic;
      D : IN  std_logic;
      F : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal A : std_logic := '0';
signal B : std_logic := '0';
signal C : std_logic := '0';
signal D : std_logic := '0';
--Outputs
signal F : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Internal_Signal PORT MAP (
       A => A,
       B => B,
       C => C,
```

```
        D => D,
        F => F
      );
 -- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
A <= '0';
B <= '1';
C <= '0';
D <= '0';
wait for 85 ns;
A <= '1';
B <= '1';
C <= '0';
D <= '0';
wait for 85 ns;
A <= '0';
B <= '1';
C <= '0';
D <= '1';
   wait;
end process;
END;
```


In figure 4.65 that shown the Signal and schematic of internal signal.



CHAPTER FOUR
VHDL EXPERIEMENTS

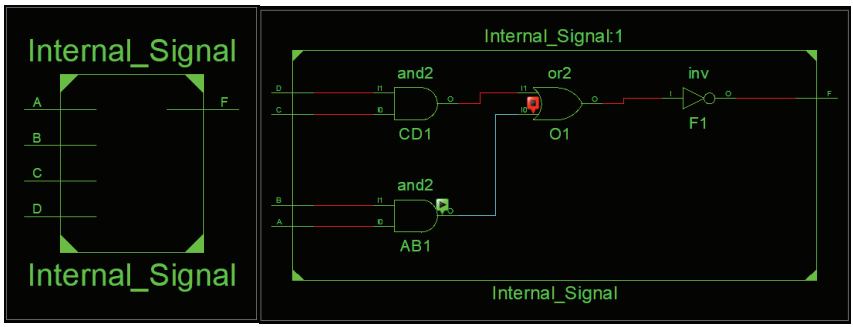


Figure 4.65: Signal and Schematic of Internal Signal

Discussion

1- Write and simulate the logic circuit in figure 4.66.

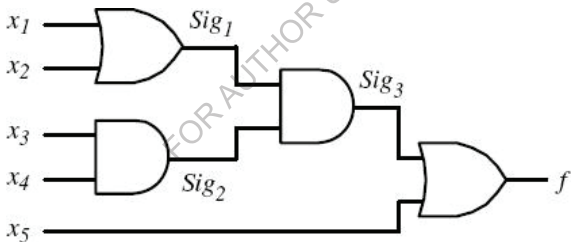


Figure 4.66: Logic Circuit

CHAPTER FOUR
VHDL EXPERIEMENTS

Encoders

Aim of Experiment

In this experiment, we will learn the encoder in VHDL.

Theory

Binary code of N digits can be used to store 2^N distinct elements of coded information. This is what encoders and decoders are used for. Encoders convert 2^N lines of input into a code of N bits and Decoders decode the N bits into 2^N lines.

Encoder

An **encoder** is a combinational circuit that converts binary information in the form of a 2^N input lines into N output lines, which represent N bit code for the input. For simple encoders, it is assumed that only one input line is active at a time.

As an example, let's consider Octal to Binary encoder. As shown in the following figure 4.67, an octal-to-binary encoder takes 8 input lines and generates 3 output lines.

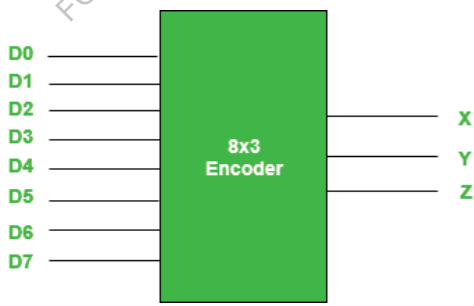


Figure 4.67: 8x3 Encoder

Truth Table

In table 4.19 that shown the truth table of encoder.

Table 4.19: 8x3 Encoder Truth Table

D7	D6	D5	D4	D3	D2	D1	D0	X	Y	Z
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

As seen from the truth table 4.19, the output is 000 when D0 is active; 001 when D1 is active; 010 when D2 is active and so on.

Hence, the encoder can be realized with OR gates as follows:

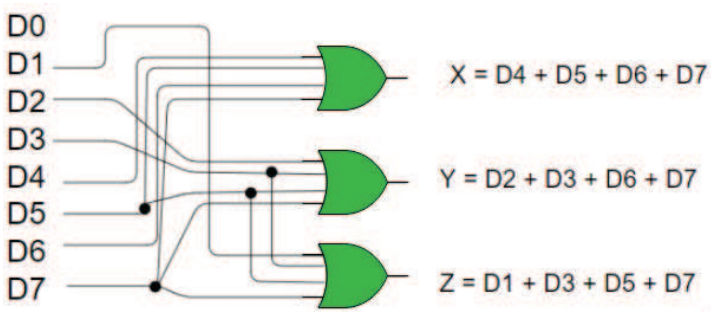


Figure 4.68: Encoder Logic Circuit with OR Gate

One limitation of this encoder is that only one input can be active at any given time. If more than one inputs are active, then the output is undefined. For example, if D6 and D3 are both active, then, our output would be 111 which is the output for D7. To overcome this, we use Priority Encoders.

Another ambiguity arises when all inputs are 0. In this case, encoder outputs 000 which actually is the output for D0 active. In order to avoid this, an extra bit can be added to the output, called the valid bit which is 0 when all inputs are 0 and 1 otherwise.

4x2 Encoder

Truth Table

In figure 4.69 that shown the 4x2 encoder.

INPUTS				OUTPUTS	
Y3	Y2	Y1	Y0	A1	A0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Figure 4.69: 4x2 Encoder

Logical expression for A1 and A0:

A1 = Y3 + Y2

A0 = Y3 + Y1

The above two Boolean functions A1 and A0 can be implemented using two input OR gates, as shown in figure 4.70:

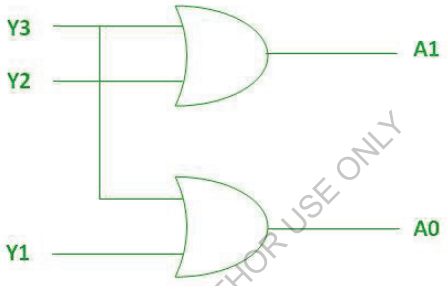


Figure 4.70: 4x2 Encoder OR Gate

VHDL Code & Simulation

CODE (4x2 Encoder)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Encoder4x2 is
    Port ( Y3,Y2,Y1 : in  STD_LOGIC;
           A1,A0 : out  STD_LOGIC);
end Encoder4x2;
architecture Behavioral of Encoder4x2 is
begin
A1 <= Y3 OR Y2;
A0 <= Y3 OR Y1;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Encoder4x2_test IS
END Encoder4x2_test;
ARCHITECTURE behavior OF Encoder4x2_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Encoder4x2
 PORT(
      Y3 : IN  std_logic;
      Y2 : IN  std_logic;
      Y1 : IN  std_logic;
      Y0 : IN  std_logic;
      A1 : OUT  std_logic;
      A0 : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal Y3 : std_logic := '0';
signal Y2 : std_logic := '0';
signal Y1 : std_logic := '0';
signal Y0 : std_logic := '0';
--Outputs
signal A1 : std_logic;
signal A0 : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Encoder4x2 PORT MAP (
       Y3 => Y3,
        Y2 => Y2,
        Y1 => Y1,
        Y0 => Y0,
        A1 => A1,
        A0 => A0
      );
 -- Stimulus process
 stim_proc: process
 begin
    -- hold reset state for 100 ns.
    wait for 50 ns;
    -- insert stimulus here
    Y3 <= '0';
    Y2 <= '0';
    Y1 <= '0';
    Y0 <= '1';
    wait for 80 ns;
    Y3 <= '0';
    Y2 <= '0';
    Y1 <= '1';
    Y0 <= '0';
    wait for 80 ns;
    Y3 <= '0';
    Y2 <= '1';
    Y1 <= '0';
    Y0 <= '0';
    wait for 80 ns;
    Y3 <= '1';
                                        Y2 <= '0';
                                        Y1 <= '0';
                                        Y0 <= '0';
                                        wait;
                                    end process;
                                    END;
```

In figure 4.71 that shown the Signal and schematic of encoder 4x2.

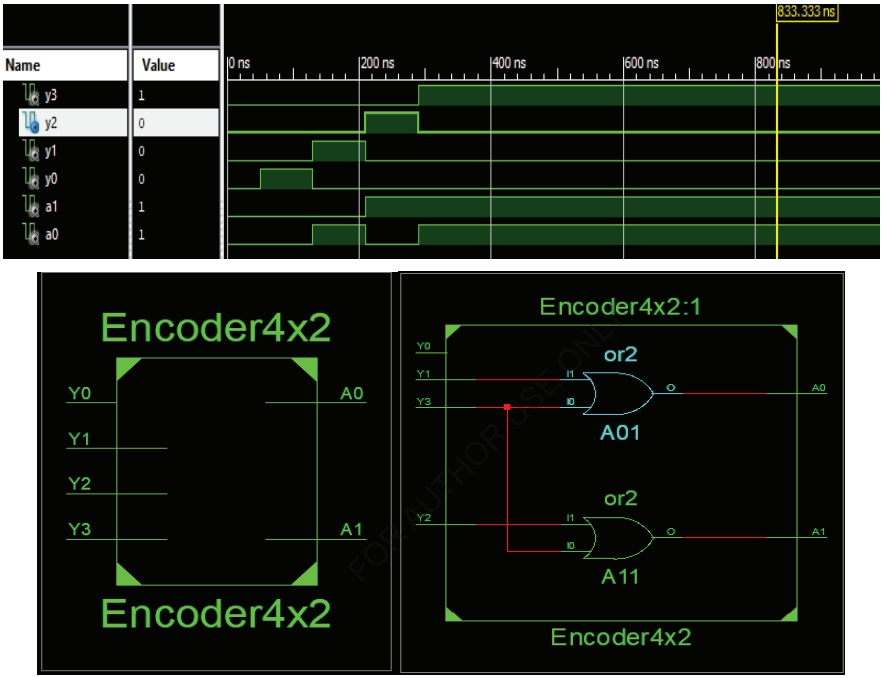


Figure 4.71: Signal and Schematic of Encoder 4x2

Discussion

1- Write and simulate the encoder 8x3 circuit.

CHAPTER FOUR
VHDL EXPERIEMENTS

Decoders

Aim of Experiment

In this experiment, we will learn the decoder in VHDL.

Theory

Decoder

A decoder does the opposite job of an encoder. It is a combinational circuit that converts n lines of input into 2n lines of output.

Let's take an example of 3-to-8-line decoder.

Truth Table

In table 4.20 that shown the truth table of 3x8 decoder.

Table 4.20: 8x3 Decoder Truth Table

X	Y	Z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Implementation

D0 is high when X = 0, Y = 0 and Z = 0. Hence,

D0 = X' Y' Z'

Similarly,

D1 = X' Y' Z

D2 = X' Y Z'

D3 = X' Y Z

D4 = X Y' Z'

D5 = X Y' Z

D6 = X Y Z'

D7 = X Y Z

In figure 4.72 that shown the 3x8 logic circuit decoder.

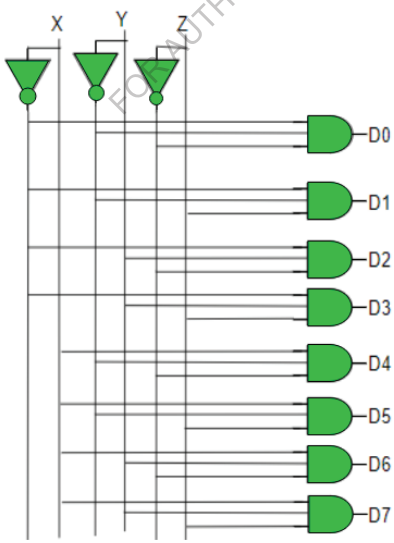


Figure 4.72: 3x8 Logic Circuit Decoder

CHAPTER FOUR
VHDL EXPERIEMENTS

2x4 Decoder

In figure 4.72 that shown the 2x4 decoder.

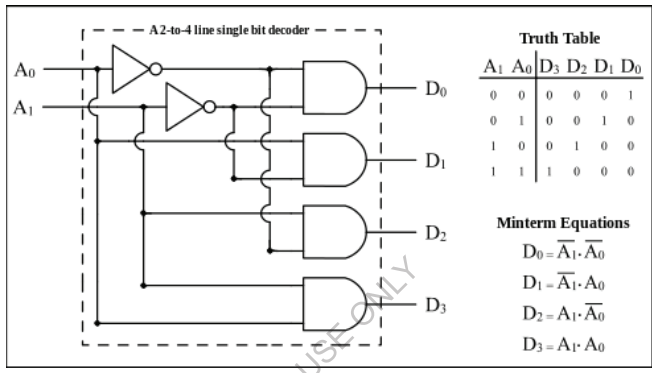


Figure 4.72: 2x4 Decoder

VHDL Code & Simulation

CODE (2x4 Decoder dataflow)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Decoder2x4_dataflow is
    Port ( A1,A0 : in  STD_LOGIC;
           D3,D2,D1,D0 : out  STD_LOGIC);
end Decoder2x4_dataflow;
architecture Behavioral of Decoder2x4_dataflow is
begin
D0 <= NOT A1 AND NOT A0;
D1 <= NOT A1 AND A0;
D2 <=  A1 AND NOT A0;
D3 <= A1 AND A0;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Decoder2x4_test IS
END Decoder2x4_test;
ARCHITECTURE behavior OF Decoder2x4_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Decoder2x4_dataflow
 PORT(
      A1 : IN  std_logic;
      A0 : IN  std_logic;
      D3 : OUT  std_logic;
      D2 : OUT  std_logic;
      D1 : OUT  std_logic;
      D0 : OUT  std_logic
     );
 END COMPONENT;
--Inputs
 signal A1 : std_logic := '0';
 signal A0 : std_logic := '0';
 --Outputs
 signal D3 : std_logic;
 signal D2 : std_logic;
 signal D1 : std_logic;
 signal D0 : std_logic;
 BEGIN
 -- Instantiate the Unit Under Test (UUT)
 uut: Decoder2x4_dataflow PORT MAP (
       A1 => A1,
        A0 => A0,
        D3 => D3,
        D2 => D2,
        D1 => D1,
        D0 => D0
      );
 -- Stimulus process
 stim_proc: process
 begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    -- insert stimulus here
 A1 <= '0';
 A0 <= '0';
 wait for 90 ns;
 A1 <= '0';
 A0 <= '1';
 wait for 90 ns;
 A1 <= '1';
 A0 <= '0';
 wait for 90 ns;
 A1 <= '1';
 A0 <= '1';
    wait;
 end process;
 END;
```

In figure 4.73 that shown the Signal and schematic of decoder 2x4.

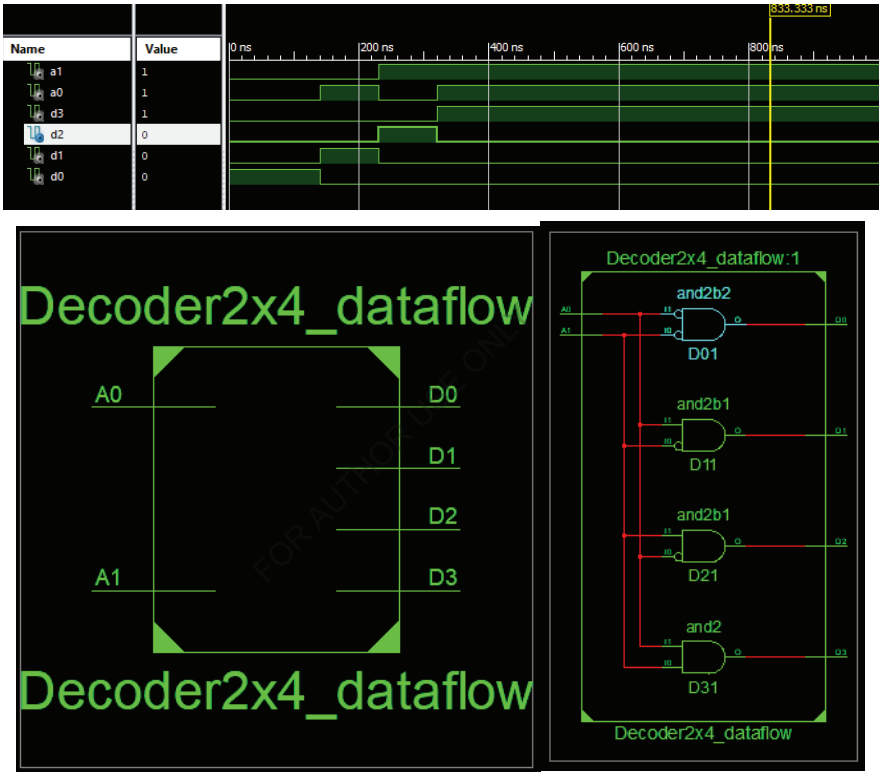


Figure 4.73: Signal and Schematic of Decoder 2x4

CODE (2x4 Decoder dataflow)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Decoder2x4_case is
    Port ( A1,A0 : in  STD_LOGIC;
           D : out  STD_LOGIC_VECTOR(0 TO 3));
end Decoder2x4_case;
architecture Behavioral of Decoder2x4_case is
signal A: std_logic_vector(1 downto 0);
begin
A <= (A1,A0);
process (A)
begin
   case A is
      when "00" => D <="1000";
      when "01" => D <="0100";
      when "10" => D <="0010";
      when "11" => D <="0001";
      when others => null;
   end case;
end process;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Decoder2x4_case_test IS
END Decoder2x4_case_test;
ARCHITECTURE behavior OF Decoder2x4_case_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Decoder2x4_case
 PORT(
      A1 : IN  std_logic;
      A0 : IN  std_logic;
      D : OUT  std_logic_vector(0 to 3)
     );
 END COMPONENT;
--Inputs
signal A1 : std_logic := '0';
signal A0 : std_logic := '0';
--Outputs
signal D : std_logic_vector(0 to 3);
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Decoder2x4_case PORT MAP (
       A1 => A1,
       A0 => A0,
       D => D
     );
-- Stimulus process
stim_proc: process
begin
```

```
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
A1<= '0';
A0 <= '0';
wait for 90 ns;
A1<= '0';
A0 <= '1';
wait for 90 ns;
A1<= '1';
A0 <= '0';
wait for 90 ns;
A1<= '1';
A0 <= '1';
   wait;
end process;
END;
```

In figure 4.74 that shown the Signal and schematic of decoder 2x4 by using case statement.

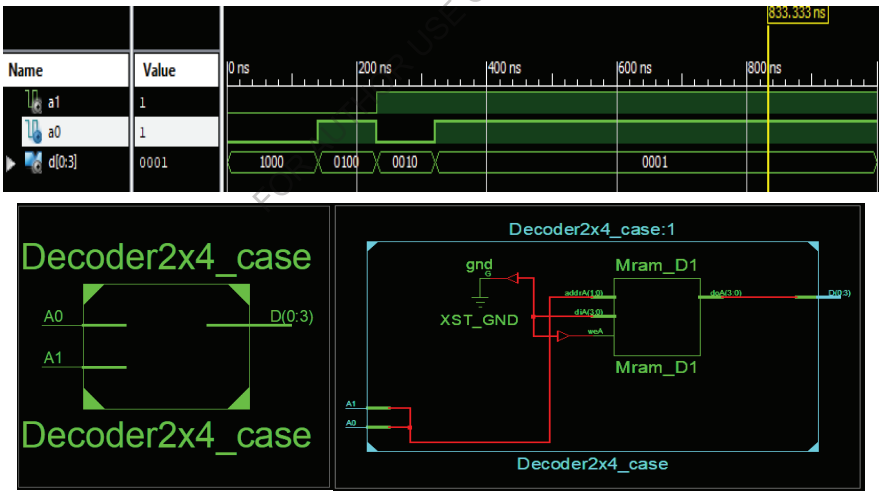


Figure 4.74: Signal and Schematic of Decoder 2x4 by Using Case Statement

Discussion

1- Write and simulate the decoder 3x8 circuit by using:

a- Dataflow (equation). b- Case Statement.

CHAPTER FOUR
VHDL EXPERIEMENTS

Multiplexer

Aim of Experiment

In this experiment, we will learn the multiplexer in VHDL.

Theory

Multiplexer

The **multiplexer** is a combinational logic circuit designed to switch one of several input lines to a single common output line. In figure 4.75 that shown the multiplexer symbol.

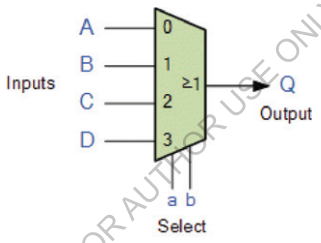


Figure 4.75: Multiplexer Symbol

Multiplexing is the generic term used to describe the operation of sending one or more analogue or digital signals over a common transmission line at different times or speeds and as such, the device we use to do just that is called a **Multiplexer**.

The **multiplexer**, shortened to "**MUX**" or "**MPX**", is a combinational logic circuit designed to switch one of several input lines through to a single common output line by the application of a control signal. **Multiplexers** operate like very fast acting multiple position rotary switches connecting or controlling multiple input lines called "**channels**" one at a time to the output.

Multiplexers, or **MUX's**, can be either digital circuits made from high speed logic gates used to switch digital or binary data or they can be analogue types using transistors,

MOSFET's or **relays** to switch one of the voltage or current inputs through to a single output.

Basic Multiplexing Switch

The most basic type of multiplexer device is that of a **one-way rotary switch** as shown in figure 4.76.

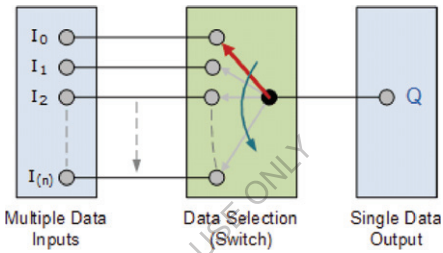


Figure 4.76: One-Way Rotary Switch

The **rotary switch**, also called a **wafer switch** as each layer of the switch is known as a wafer, is a mechanical device whose input is selected by rotating a shaft. In other words, the rotary switch is a manual switch that you can use to select individual data or signal lines simply by turning its inputs "ON" or "OFF". So how can we select each data input automatically using a digital device.

In digital electronics, multiplexers are also known as **data selectors** because they can "**select**" each input line, are constructed from individual **Analogue Switches** encased in a single IC package as opposed to the "mechanical" type selectors such as normal conventional switches and relays.

They are used as one method of reducing the number of logic gates required in a circuit design or when a single data line or data bus is required to carry two or more different digital signals. For example, a single 8-channel multiplexer.

Generally, the selection of each input line in a multiplexer is controlled by an additional set of inputs called control lines and according to the binary condition of these control

inputs, either “HIGH” or “LOW” the appropriate data input is connected directly to the output. Normally, a multiplexer has an even number of 2n data input lines and a number of “control” inputs that correspond with the number of data inputs.

Note that multiplexers are different in operation to Encoders. Encoders are able to switch an n-bit input pattern to multiple output lines that represent the binary coded (BCD) output equivalent of the active input.

Multiplexer Input Line Selection

In figure 4.77 that shown the multiplexer input line selection.

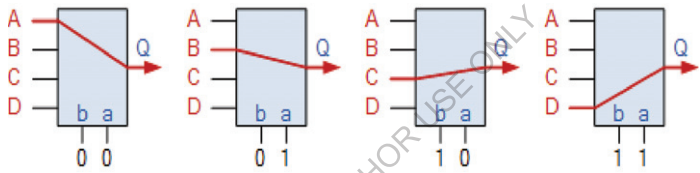


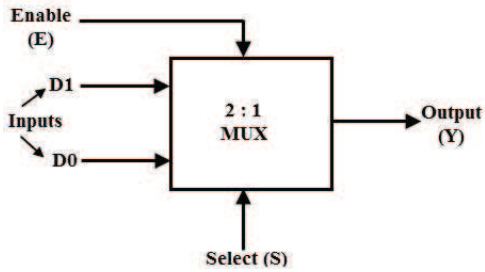
Figure 4.77: Multiplexer Input Line Selection

Adding more control address lines, (n) will allow the multiplexer to control more inputs as it can switch 2^n inputs but each control line configuration will connect only ONE input to the output.

Then the implementation of the Boolean expression above using individual logic gates would require the use of seven individual gates consisting of AND, OR and NOT gates as shown.

2-input Multiplexer Design

We can build a simple 2-line to 1-line (2-to-1) multiplexer from basic logic AND gates as shown in figure 4.78.



$$Y = D_0 \bar{S} + D_1 S$$

Select	Inputs		Output
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	1

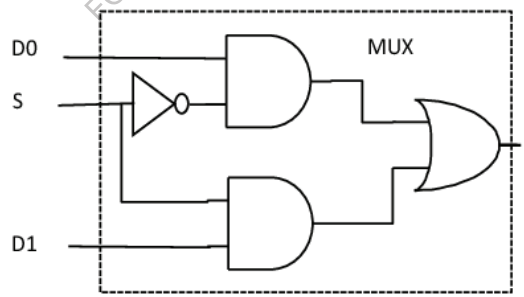
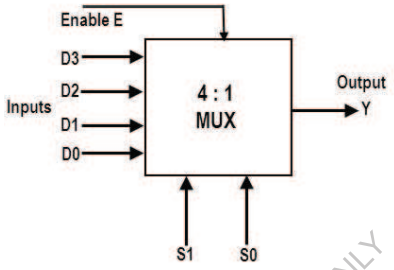


Figure 4.78: Multiplexer 2-to-1

CHAPTER FOUR
VHDL EXPERIEMENTS

4 Channel Multiplexer using Logic Gates

In figure 4.79 that shown the 4:1 MUX.



Select Data Inputs		Output
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

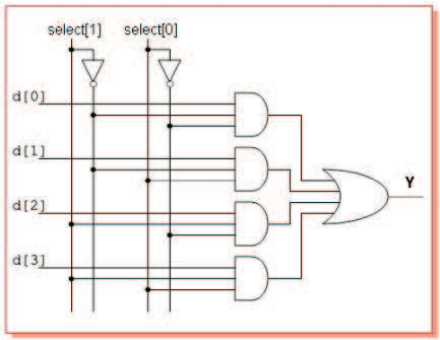


Figure 4.79: Multiplexer 4-to-1

CHAPTER FOUR
VHDL EXPERIEMENTS

VHDL Code & Simulation

CODE (MUX 2:1 dataflow)

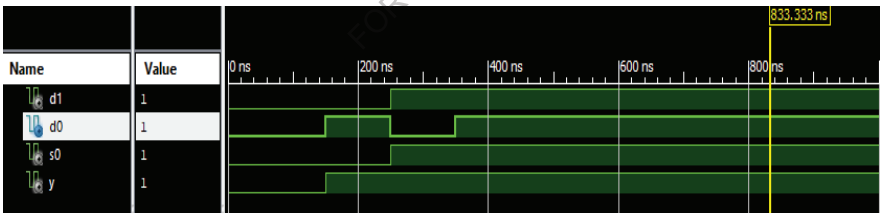
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MUX2_1 is
    Port ( D1,D0,S0 : in  STD_LOGIC;
           Y : out  STD_LOGIC);
end MUX2_1;
architecture Behavioral of MUX2_1 is
begin
Y <= (D0 AND NOT S0) OR (D1 AND S0);
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY MUX2_1_test IS
END MUX2_1_test;
ARCHITECTURE behavior OF MUX2_1_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT MUX2_1
 PORT(
      D1 : IN  std_logic;
      D0 : IN  std_logic;
      S0 : IN  std_logic;
      Y : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal D1 : std_logic := '0';
signal D0 : std_logic := '0';
signal S0 : std_logic := '0';
--Outputs
signal Y : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: MUX2_1 PORT MAP (
       D1 => D1,
       D0 => D0,
       S0 => S0,
       Y => Y
     );
```

```
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
S0 <= '0';
D0 <= '0';
D1 <= '0';
wait for 100 ns;
S0 <= '0';
D0 <= '1';
D1 <= '0';
wait for 100 ns;
S0 <= '1';
D0 <= '0';
D1 <= '1';
wait for 100 ns;
S0 <= '1';
D0 <= '1';
D1 <= '1';
   wait;
end process;
END;
```


In figure 4.80 that shown the Signal and schematic of multiplexer 2:1.



CHAPTER FOUR
VHDL EXPERIEMENTS

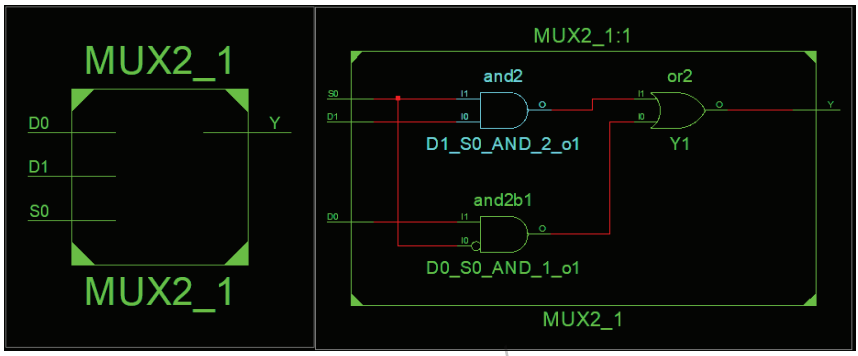


Figure 4.80: Signal and Schematic of Multiplexer 2:1

CODE (MUX 4:1 dataflow)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MUX4_1 is
    Port ( D3,D2,D1,D0,S1,S0 : in  STD_LOGIC;
           Y : out  STD_LOGIC);
end MUX4_1;
architecture Behavioral of MUX4_1 is
begin
Y <= (D0 AND NOT S1 AND NOT S0)OR (D1 AND NOT S1 AND S0) OR
(D2 AND S1 AND NOT S0) OR (D3 AND S1 AND S0);
end Behavioral;
```

Test Bench

CHAPTER FOUR
VHDL EXPERIEMENTS

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY MUX4_1_test IS
END MUX4_1_test;
ARCHITECTURE behavior OF MUX4_1_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT MUX4_1
 PORT(
      D3 : IN  std_logic;
      D2 : IN  std_logic;
      D1 : IN  std_logic;
      D0 : IN  std_logic;
      S1 : IN  std_logic;
      S0 : IN  std_logic;
      Y : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal D3 : std_logic := '0';
signal D2 : std_logic := '0';
signal D1 : std_logic := '0';
signal D0 : std_logic := '0';
signal S1 : std_logic := '0';
signal S0 : std_logic := '0';
--Outputs
signal Y : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
  uut: MUX4_1 PORT MAP (
          D3 => D3,
          D2 => D2,
          D1 => D1,
          D0 => D0,
          S1 => S1,
          S0 => S0,
          Y => Y
        );
  -- Stimulus process
  stim_proc: process
  begin
     -- hold reset state for 50 ns.
     wait for 50 ns;
     -- insert stimulus here
  S1 <= '0';
  S0 <= '0';
  D3 <= '0';
  D2 <= '0';
  D1 <= '0';
  D0 <= '1';
  wait for 70 ns;
  S1 <= '0';
  S0 <= '1';
  D3 <= '0';
  D2 <= '0';
  D1 <= '1';
  D0 <= '0';
  wait for 70 ns;
  S1 <= '1';
  S0 <= '0';
  D3 <= '0';
  D2 <= '1';
  D1 <= '0';
  D0 <= '0';
  wait for 70 ns;
  S1 <= '1';
  S0 <= '1';
  D3 <= '1';
  D2 <= '0';
  D1 <= '0';
  D0 <= '0';
     wait;
  end process;
  END;
```

In figure 4.81 that shown the Signal and schematic of multiplexer 4:1.

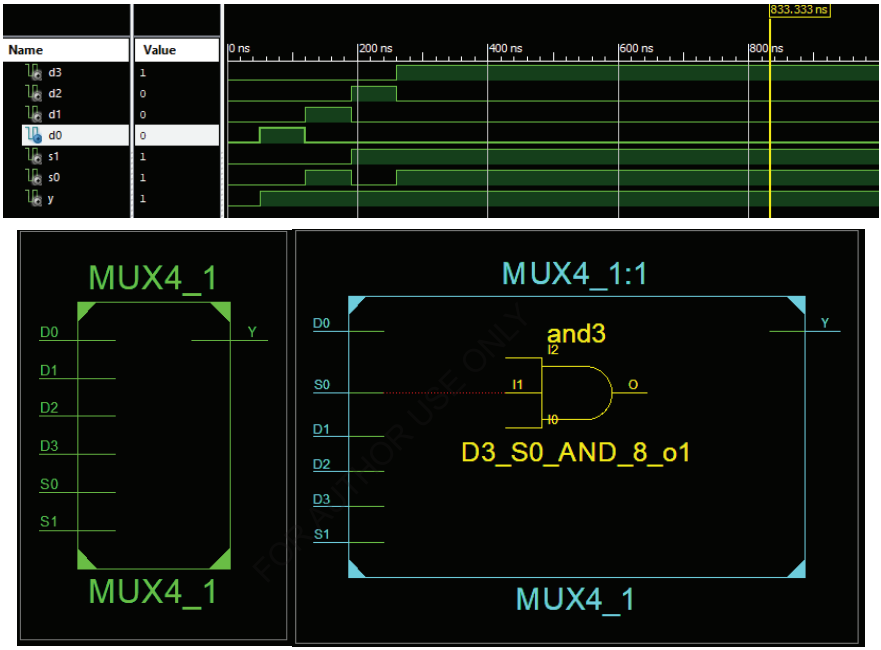


Figure 4.81: Signal and Schematic of Multiplexer 4:1

Discussion

1- Write and simulate the MUX 8:1 circuit.

2- Write and simulate the MUX 4:1 circuit by using CSA method.

CHAPTER FOUR
VHDL EXPERIEMENTS

Demultiplexer

Aim of Experiment

In this experiment, we will learn the demultiplexer in VHDL.

Theory

Demultiplexer

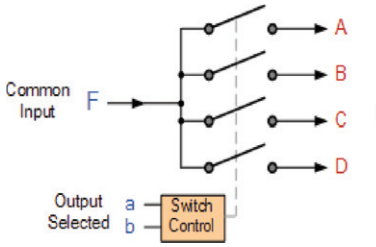
The **demultiplexer** is a combinational logic circuit designed to switch one common input line to one of several separate output lines.

The **data distributor**, known more commonly as a **Demultiplexer** or "**Demux**" for short, is the exact opposite of the Multiplexer.

The **demultiplexer** takes one single input data line and then switches it to any one of a number of individual output lines one at a time. The demultiplexer converts a serial data signal at the input to a parallel data at its output lines as shown below.

1-to-4 Channel De-multiplexer

In figure 4.82 that shown the 1:4 Demultiplexer.



Data Outputs

Output Select		Data Output Selected
a	b	
0	0	A
0	1	B
1	0	C
1	1	D

Figure 4.82: 1:4 Demultiplexer

The Boolean expression for this **1-to-4 Demultiplexer** above with outputs A to D and data select lines a, b is given as:

$$F = \overline{a}\overline{b}A + \overline{a}bB + a\overline{b}C + abD$$

The function of the Demultiplexer is to switch one common data input line to any one of the 4 output data lines A to D in our example above. As with the multiplexer the individual solid-state switches are selected by the binary input address code on the output select pins "a" and "b" as shown in figure 4.83.

Demultiplexer Output Line Selection

In figure 4.83 that shown the Demultiplexer Output Line Selection.

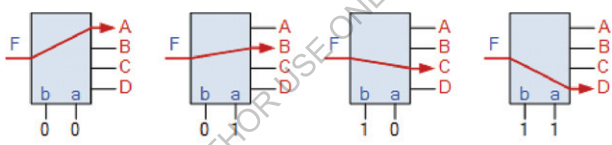


Figure 4.82: 1:4 Demultiplexer Output Line Selection

As with the previous multiplexer circuit, adding more address line inputs it is possible to switch more outputs giving a 1-to-2n data line outputs.

Some standard demultiplexer IC´s also have an additional "enable output" pin which disables or prevents the input from being passed to the selected output. Also, some have latches built into their outputs to maintain the output logic level after the address inputs have been changed.

However, in standard decoder type circuits the address input will determine which single data output will have the same value as the data input with all other data outputs having the value of logic "0".

The implementation of the Boolean expression above using individual logic gates would require the use of six individual gates consisting of AND and NOT gates as shown.

4 Channel Demultiplexer using Logic Gates

CHAPTER FOUR
VHDL EXPERIEMENTS

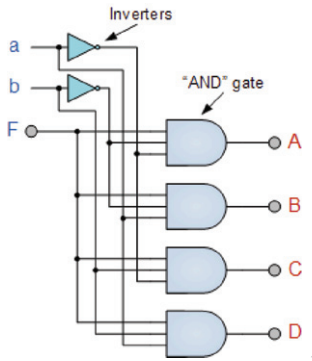


Figure 4.83: 1:4 Demultiplexer Logic Circuit

The symbol used in logic diagrams to identify a demultiplexer is as shown in figure 4.84.

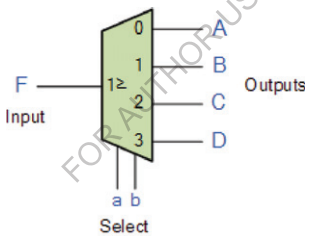


Figure 4.84: 1:4 Demultiplexer Logic Symbol

VHDL Code & Simulation

CODE (Demux 4:1 dataflow)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Demux1_4 is
    Port ( Y : in  STD_LOGIC;
           SEL : in  STD_LOGIC_VECTOR (1 downto 0);
           D : out  STD_LOGIC_VECTOR (3 downto 0));
end Demux1_4;
architecture Behavioral of Demux1_4 is
begin
D(0) <= (not SEL(0)) and (not SEL(1));
D(1) <= SEL(0) and (not SEL(1));
D(2) <= (not SEL(0)) and SEL(1);
D(3) <= SEL(0) and SEL(1);
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Demux1_4_test IS
END Demux1_4_test;
ARCHITECTURE behavior OF Demux1_4_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Demux1_4
 PORT(
      Y : IN  std_logic;
      SEL : IN  std_logic_vector(1 downto 0);
      D : OUT  std_logic_vector(3 downto 0)
     );
 END COMPONENT;
--Inputs
signal Y : std_logic := '0';
signal SEL : std_logic_vector(1 downto 0) := (others => '0');
--Outputs
signal D : std_logic_vector(3 downto 0);
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Demux1_4 PORT MAP (
       Y => Y,
       SEL => SEL,
       D => D
     );
-- Stimulus process
stim_proc: process
begin
```

```
    -- hold reset state for 50 ns.
    wait for 50 ns;
    -- insert stimulus here
  SEL(0) <= '0';
  SEL(1) <= '0';
  wait for 80 ns;
  SEL(0) <= '1';
  SEL(1) <= '0';
  wait for 80 ns;
  SEL(0) <= '0';
  SEL(1) <= '1';
  wait for 80 ns;
  SEL(0) <= '1';
  SEL(1) <= '1';
    wait;
 end process;
 END;
```

In figure 4.85 that shown the Signal and schematic of demultiplexer 4:1.

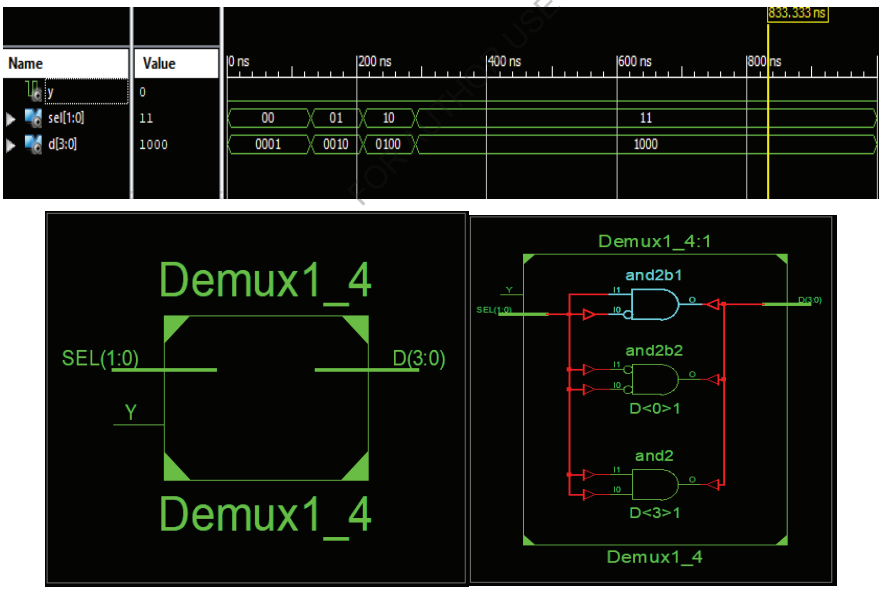


Figure 4.85: Signal and Schematic of Demultiplexer 4:1

CHAPTER FOUR
VHDL EXPERIEMENTS

Discussion

1- Write and simulate the Demux 1:8 circuit.

2- Write and simulate the Demux 1:4 circuit by using CSA method.

CHAPTER FOUR
VHDL EXPERIEMENTS

Latches

Aim of Experiment

In this experiment, we will learn the Latches in VHDL.

Theory

There are two types of memory elements based on the type of triggering that is suitable to operate it.

- Latches.
- Flip-flops.

In this experiment we learn the Latch circuit by using NOR and NAND gates.

Latches

A **latch** is an electronic logic circuit that has two inputs and one output. One of the inputs is called the **SET input**; the other is called the **RESET** input.

Latch circuits can be either **active-high** or **active-low**. The difference is determined by whether the operation of the latch circuit is triggered by **HIGH** or **LOW** signals on the inputs.

Active-high circuit: Both inputs are normally tied to ground (LOW), and the latch is triggered by a momentary HIGH signal on either of the inputs.

Active-low circuit: Both inputs are normally HIGH, and the latch is triggered by a momentary LOW signal on either input.

In an **active-high latch**, both the **SET** and **RESET** inputs are connected to ground. When the SET input goes HIGH, the output also goes HIGH. When the SET input returns to LOW, however, the output remains HIGH. The output of the active-high latch stays HIGH until the RESET input goes HIGH. Then, the output returns to LOW and will go HIGH again only when the SET input is triggered once more.

In other words, the latch remembers that the SET input has been activated. If the SET input goes HIGH for even a moment, the output goes HIGH and stays HIGH, even after the SET input returns to LOW. The output returns to LOW only when the **RESET** input goes HIGH.

On the other hand, in an **active-low latch** the inputs are normally held at **HIGH**. When the SET input momentarily goes LOW, the output goes HIGH. The output then stays HIGH until the RESET input momentarily goes LOW.

Note that most latch circuits actually have a second output that is simply the first output inverted. In other words, whenever the first output is HIGH, the second output is LOW, and vice versa. These outputs are usually referred to as **Q** and **Q-bar** with the latter notated as follows: $\overline{Q}$.

The notation is usually pronounced either "**bar Q**" or "**Q bar**," though some people pronounce it "**not Q**." The horizontal bar symbol over a label is a common logical shorthand for inversion. That is, **Q-bar** is the inverse of **Q**. If Q is HIGH, Q-bar is LOW, and if Q is LOW, Q-bar is HIGH.

You can easily create an **active-high latch** from a pair of **NOR gates**. (The output of a NOR gate is HIGH if both inputs are LOW; otherwise, the output is LOW.) In this circuit, the **SET input** is connected to one of the inputs of the first NOR gate, and the **RESET input** is connected to one of the inputs of the second NOR gate.

The trick of the latch circuit is that the output of the **NOR** gates are cross-connected to the remaining NOR gate inputs. In other words, the output from the first NOR gate is connected to one of the inputs of the second NOR gate, and the output from the second NOR gate is connected to one of the inputs of the first NOR gate, as shown in figure 4.86.

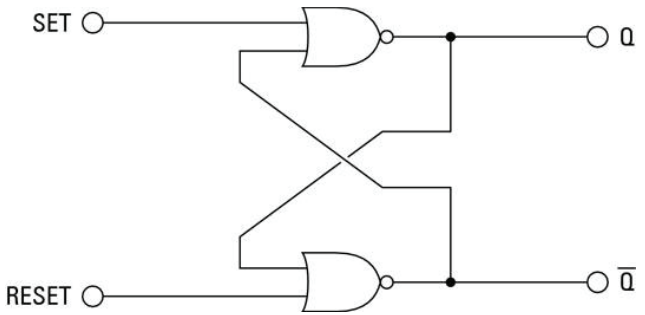


Figure 4.86: SR NOR Latch

The next schematic is for an **active-low latch**. The only difference between this schematic and the one shown previously is that the active-low latch uses **NAND gates** instead of **NOR gates**. Notice also in this diagram in figure 4.87 that the inputs are referred to as **SET-bar** and **RESET-bar** rather than **SET** and **RESET**, which indicates that the inputs are **active-low**.

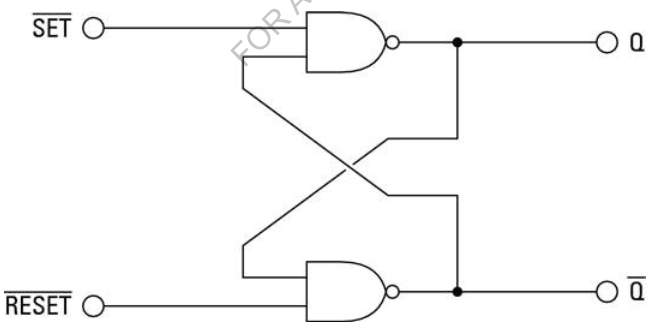
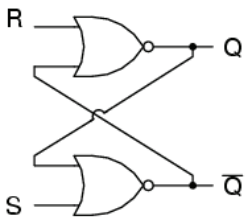


Figure 4.87: SR NAND Latch

In figure 4.88 that shown the truth table for SR latch by using NOR and NAND gates.

CHAPTER FOUR
VHDL EXPERIEMENTS



S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

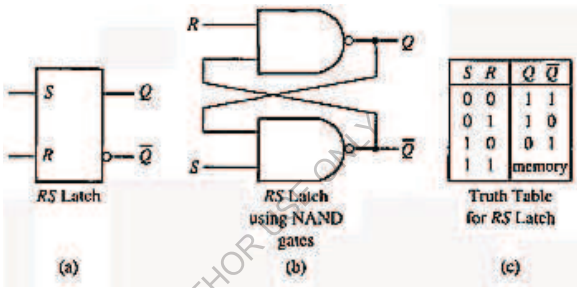


Figure 4.88: Truth Table for SR NOR and NAND Latch

In figure 4.89 that shown another way for SR NOR latch circuit and truth table.

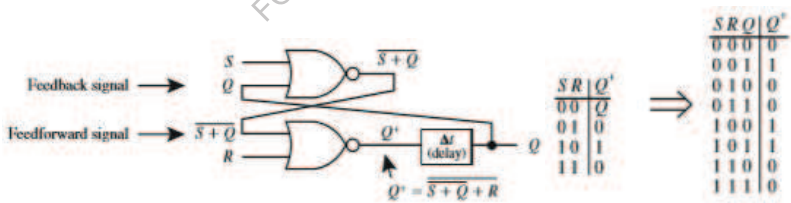


Figure 4.89: Truth Table for SR NOR Latch

$$Q^+ = \overline{\overline{S+Q}+R}$$

Or we can rewrite the characteristic equation in SOP form (AND/OR form) as follows:

$$Q^+ = \overline{\overline{Q+S+R}} = \overline{\overline{Q+S}\cdot\overline{R}} = (Q+S)\overline{R} = Q\overline{R} + S\overline{R}$$

In figure 4.90 that shown another way for SR NOR latch circuit and truth table.

CHAPTER FOUR
VHDL EXPERIEMENTS

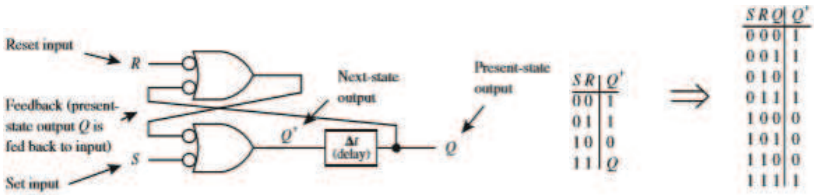


Figure 4.90: Truth Table for SR NAND Latch

$$Q+ = \overline{\overline{R} + \overline{Q}} + \overline{S}$$

VHDL Code & Simulation

CODE (SR NOR Latch)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SR_NOR_Latch is
    Port ( S,R : in  STD_LOGIC;
           Q : inout  STD_LOGIC);
end SR_NOR_Latch;
architecture Behavioral of SR_NOR_Latch is
begin
Q <= (Q and not R)or (S and not R);
end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY SR_NOR_Latch_test IS
END SR_NOR_Latch_test;
ARCHITECTURE behavior OF SR_NOR_Latch_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT SR_NOR_Latch
 PORT(
      S : IN  std_logic;
      R : IN  std_logic;
      Q : INOUT  std_logic
     );
 END COMPONENT;
--Inputs
signal S : std_logic := '0';
signal R : std_logic := '0';
--BiDirs
signal Q : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: SR_NOR_Latch PORT MAP (
       S => S,
       R => R,
       Q => Q
     );
-- Stimulus process
stim_proc: process
begin
```

```
    -- hold reset state for 50 ns.
    wait for 50 ns;
    -- insert stimulus here
S <= '0';
R <='0';
wait for 100 ns;
S <= '0';
R <='1';
wait for 100 ns;
S <= '1';
R <='0';
wait for 100 ns;
S <= '1';
R <='1';
    wait;
end process;
END;
```

In figure 4.91 that shown the Signal and schematic of SR NOR latch.

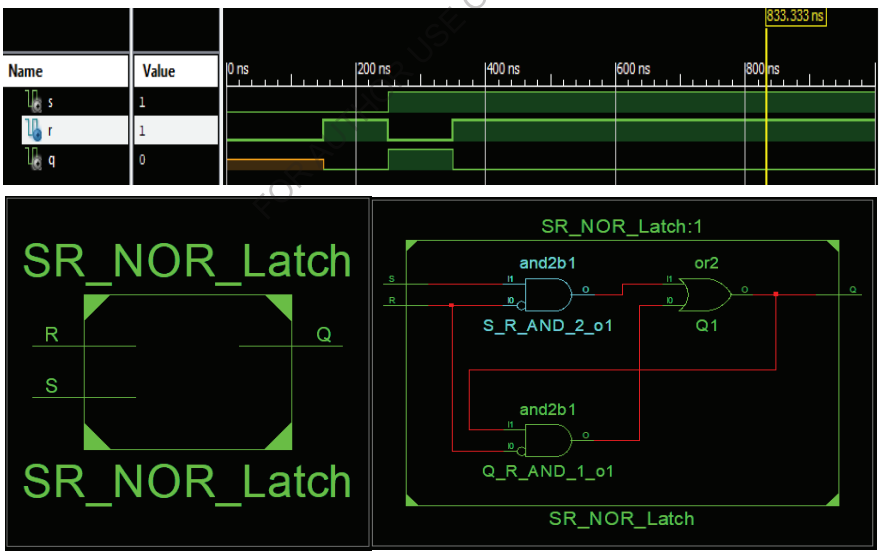


Figure 4.91: Signal and Schematic of SR NOR Latch

CODE (SR NAND Latch)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SR_NAND_Latch is
    Port ( S,R : in  STD_LOGIC;
           Q : inout  STD_LOGIC);
end SR_NAND_Latch;
architecture Behavioral of SR_NAND_Latch is
begin
Q <= (R and Q)or not S;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY SR_NAND_Latch_test IS
END SR_NAND_Latch_test;
ARCHITECTURE behavior OF SR_NAND_Latch_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT SR_NAND_Latch
 PORT(
      S : IN  std_logic;
      R : IN  std_logic;
      Q : INOUT  std_logic
     );
 END COMPONENT;
--Inputs
signal S : std_logic := '0';
signal R : std_logic := '0';
--BiDirs
signal Q : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: SR_NAND_Latch PORT MAP (
       S => S,
       R => R,
       Q => Q
     );
-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    -- insert stimulus here
 S <= '0';
 R <= '0';
 wait for 100 ns;
 S <= '0';
 R <= '1';
 wait for 100 ns;
 S <= '1';
 R <= '0';
 wait for 100 ns;
 S <= '1';
 R <= '1';
    wait;
 end process;
 END;
```

In figure 4.92 that shown the Signal and schematic of SR NAND latch.

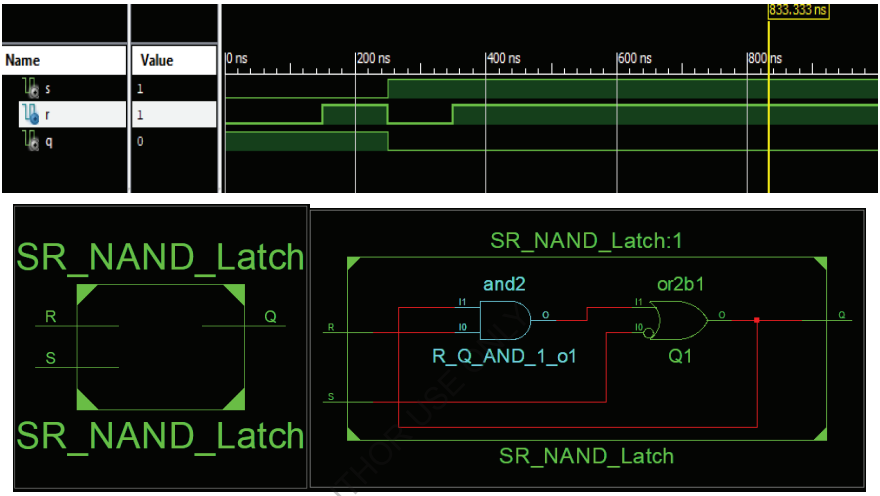


Figure 4.92: Signal and Schematic of SR NAND Latch

Discussion

1- Write and simulate the SR NOR circuit by using another way.

2- Write complete VHDL code for the S-R NAND latch shown in Figure 4.93. Use two Boolean equations— that is, one Boolean equation for signal E and another Boolean equation for output Q.

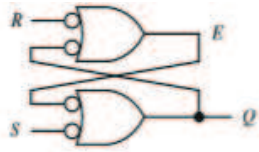


Figure 4.93: SR NAND Latch

CHAPTER FOUR
VHDL EXPERIEMENTS

Gated Latches

Aim of Experiment

In this experiment, we will learn the Gated Latches in VHDL.

Theory

Gated Latch

In the field of electronics, a gated latch is a latch that has a third input that must be active in order for the **SET** and **RESET** inputs to take effect. This third input is sometimes called **ENABLE** because it enables the operation of the SET and RESET inputs (or called **CONTROL**).

The **ENABLE** input can be connected to a simple switch. Then, when the switch is closed, the SET and RESET inputs are enabled; when the switch is open, any changes in the SET and RESET inputs are ignored.

Alternatively, the **ENABLE** input can be connected to a **clock pulse**. For example, you could connect the output of a **555-timer** circuit to the **ENABLE** input. Then, the latch inputs will be operational only when the 555 timer's output is HIGH. Note that the ENABLE input is often called the **CLOCK** input.

You can easily add an **ENABLE** input to a latch by adding a pair of **NAND** gates. Here, the **SET** and **RESET** inputs (SR latch) are connected to one input of each of the two **NAND** gates. The **ENABLE** input is connected to the other input of each **NAND** gate. Then, the output from these gates are used as the inputs to the basic latch circuit, as shown in figure 4.94.

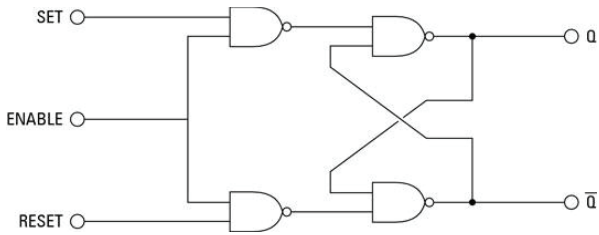


Figure 4.94: SR NAND Gated Latch

In figure 4.95 that shown the SR NOR gated latch.

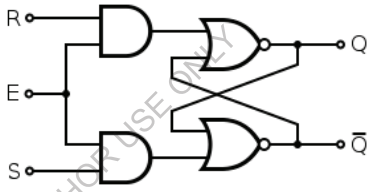


Figure 4.95: SR NOR Gated Latch

Another common type of gated latch is called a **gated D latch**, which has just two inputs: **DATA** and **ENABLE** (or called **CONTROL**). When a HIGH is received at the **ENABLE** input, the **DATA** input is copied to the output. Even if the **ENABLE** input then goes low, the output remains unchanged. The output cannot be changed until the **ENABLE** input goes high.

To create a **gated D latch** from a **gated SR latch**, you simply connect the **SET** and **RESET** inputs together through an inverter. Thus, the SET and RESET inputs will always be opposite of one another. When the DATA input is HIGH, the SET input is HIGH and the RESET input is LOW. When the DATA input is LOW, the SET input is LOW and the RESET input is HIGH, as shown in figure 4.96.

CHAPTER FOUR
VHDL EXPERIEMENTS

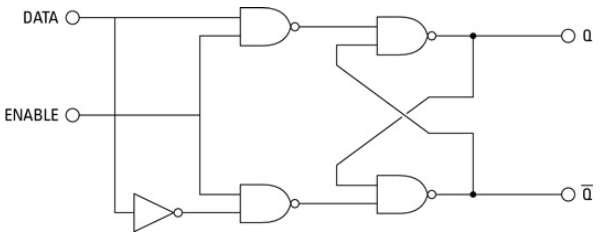
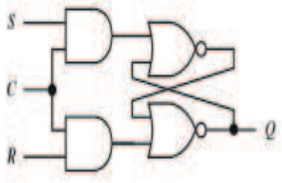
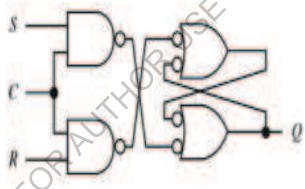


Figure 4.96: D Gated Latch

In figure 4.96 that shown the truth table for SR gated latch.





Gated S-R latch (using an S-R NOR latch)

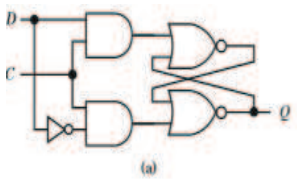
C	S	R	Q^+
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	0

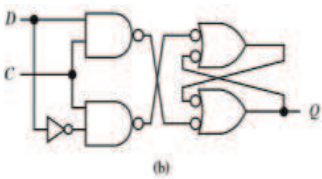
Gated S-R latch (using an S-R NAND latch)

C	S	R	Q^+
0	0	0	Q
0	0	1	Q
0	1	0	Q
0	1	1	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	1

Figure 4.97: SR Gated Latch

In figure 4.98 that shown the truth table for D gated latch.





C	D	Q^+
0	0	Q
0	1	Q
1	0	0
1	1	1

⇒

C	D	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 4.98: D Gated Latch

In figure 4.99 that shown K-map and reduce AND/OR circuit for D gated latch.

CHAPTER FOUR
VHDL EXPERIEMENTS

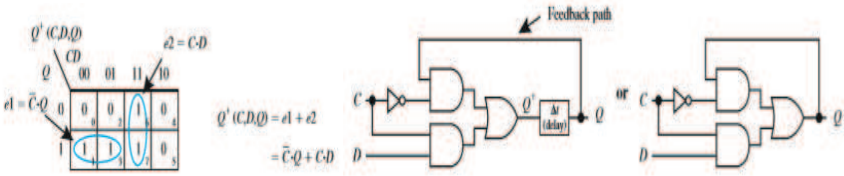


Figure 4.99: K-map and reduce AND/OR circuit D Gated Latch

In this experiment we make the D gated latch in VHDL.

VHDL Code & Simulation

CODE (D Latch)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity D_Latch_CSA is
    Port ( C,D : in  STD_LOGIC;
           Q : inout  STD_LOGIC);
end D_Latch_CSA;
architecture Behavioral of D_Latch_CSA is
begin
Q <= '1' when ((not C and Q) or (C and D)) = '1' else '0';
-- Q <= (not C and Q) or (C and D) --alternate way
end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY D_Latch_CSA_test IS
END D_Latch_CSA_test;
ARCHITECTURE behavior OF D_Latch_CSA_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT D_Latch_CSA
 PORT(
      C : IN  std_logic;
      D : IN  std_logic;
      Q : INOUT  std_logic
     );
 END COMPONENT;
--Inputs
signal C : std_logic := '0';
signal D : std_logic := '0';
--BiDirs
signal Q : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: D_Latch_CSA PORT MAP (
       C => C,
       D => D,
       Q => Q
     );
-- Stimulus process
stim_proc: process
begin
```

```
   -- hold reset state for 50 ns.
   wait for 50 ns;
   -- insert stimulus here
   C <= '0';
   D <= '1';
   wait for 100 ns;
   C <= '1';
   D <= '1';
   wait for 50 ns;
   C <= '1';
   D <= '0';
   wait for 100 ns;
   C <= '1';
   D <= '1';
   wait;
end process;
END;
```

In figure 4.100 that shown the Signal and schematic of D latch.

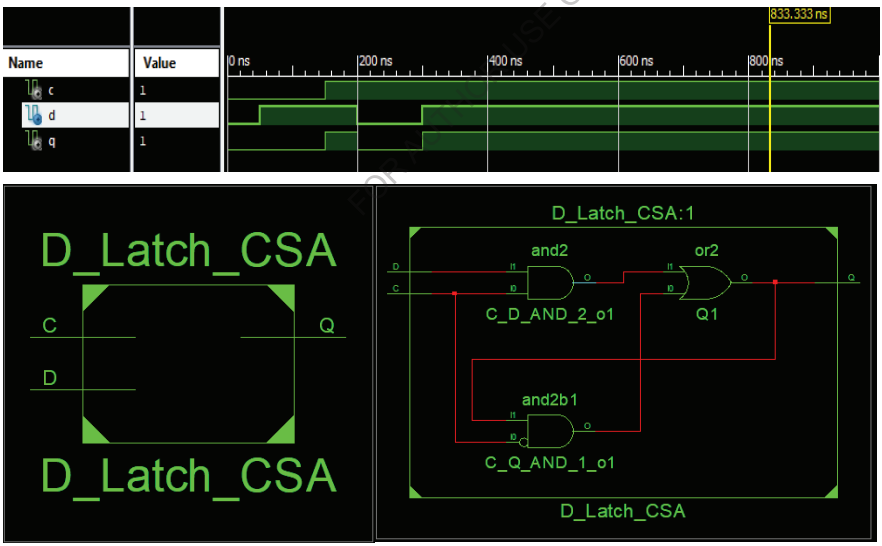


Figure 4.100: Signal and Schematic of D Latch

CHAPTER FOUR
VHDL EXPERIEMENTS

Discussion

1- Write complete VHDL code for the D Latch circuit shown in Figure 101. Use five Boolean equations— that is, one Boolean equation for signal E, and one Boolean equation for each NAND gate outputs.

Figure 4.101: D Latch NAND

CHAPTER FOUR
VHDL EXPERIEMENTS

Designing A Simple Clock

Aim of Experiment

In this experiment, we will learn the designing a simple clock in VHDL.

Theory

In this experiment, we present the design of a simple circuit called a **clock**. A **clock** provides a sequence of pulses at its output. **Clocks** are used in the design of synchronous sequential logic systems. For example, your PC (personal computer) uses a clock to operate the digital logic circuits inside the computer in a synchronous manner—that is, synchronized with the clock. The output of a clock is used in the operation of **D latches** and **flip-flops**. The control input to a **D Latch** and a **D flip-flop** is driven by a system clock.

A **state diagram** is a graphical way of describing a sequential logic circuit—that is, a logic circuit with **feedback**. A circle or oval is used to represent each state of the circuit. A **state transition line** (a directed line segment) is used to represent the flow of the circuit when it changes from one state to the next state. The four major parts of a state diagram are the state bubbles, state-transition lines, state variables, and states or state values. The state diagram for a simple clock is shown in Figure 4.102.

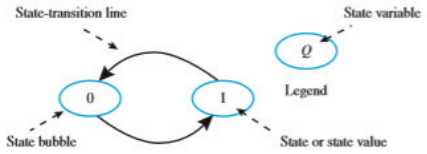


Figure 4.102: State Diagram for a Clock

If the clock is in the reset state (**Q = 0**) it goes to the set state (**Q = 1**) after a brief delay time. It then goes back to the reset state and continues in this manner as long as power is

supplied to the circuit. Because the clock switches from 0 to 1 to 0, a clock is sometimes called an **oscillator**. Notice that there are no external inputs. If there were external input signals, they would be placed beside the **state-transition lines**. What does a clock circuit look like? First, we need to obtain the equation for the clock to see how to design it.

It is common in digital design to describe a logic element by its state diagram or its **PS/NS table**. The state diagram, or PS/NS table, can then be used to obtain the equation or equations for the logic element. By observing the state diagram, we can write the PS/NS table for the clock as shown in Table 4.21.

Table 4.21: PS/NS table for the clock (oscillator)

Q	Q^+
0	1
1	0

From the PS/NS table, we can write the characteristic equations for the clock as $Q^+=\sum m(0) = m0 = \overline{Q}$.

Figure 4.103 shows the gate-level circuit design for the clock using the 1s of the function Q^+—that is, $Q^+ = \overline{Q}$.

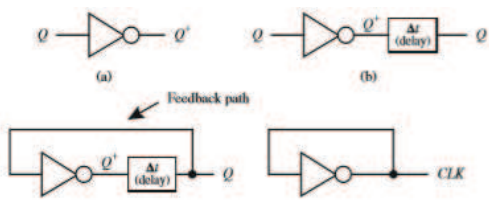


Figure 4.103: Gate-level circuit Design for the Clock

The output signal of a clock circuit is usually called **CLK**. The circuit delay, 1tp, is the propagation delay time for the NOT gate used in the circuit. The period is **T = 2tp**, the frequency of oscillation is **f = 1/T = 1/(2tp),** and the duty cycle is **DC = 50%**, as shown in the timing diagram in figure 4.104.

CHAPTER FOUR
VHDL EXPERIEMENTS

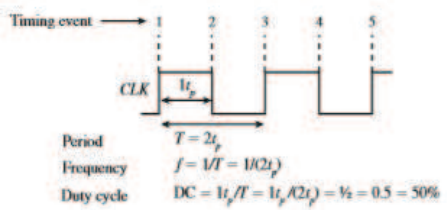


Figure 4.104: Ideal Timing Diagram for the Clock Circuit

The duty cycle of a periodic square waveform is defined as the on time (waveform is 1 or high) divided by "the period of the waveform." For the timing diagram of the clock circuit shown in figure 4.104, the duty cycle is 50% because the on time is one-half the period of the waveform.

The propagation delay time **tp** of the **NOT** gate in the clock circuit determines the period and thus the frequency of the circuit. Providing a longer delay time increases the period and thus provides a **lower frequency**. Adding buffers in cascade with the NOT gate will provide a longer delay time. Using an odd number of NOT gates will also provide a longer delay time to ensure that the clock circuit will oscillate at a lower frequency, when constructed with hardware.

VHDL Code & Simulation

We make a complete VHDL design for a clock circuit with a **DISABLE** input signal. When **DISABLE** is **1**, the output of the clock circuit **CLK** is **0**, and when **DISABLE** is **0**, the output of the clock circuit CLK oscillates at a very high unknown frequency. The circuit for this clock is simply a NOR gate with feedback. In practice, you cannot create a clock with just a NOR gate as we are doing here, because the frequency is too high and is not predictable or controllable.

CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity CLOCK is
    Port ( DISABLE : in  STD_LOGIC;
           CLK : inout  STD_LOGIC);
end CLOCK;
architecture Behavioral of CLOCK is
begin
-- Clock Generation
   CLK <= CLK nor DISABLE;
end Behavioral;
```

In figure 4.105 that shown the signal of clock generation.

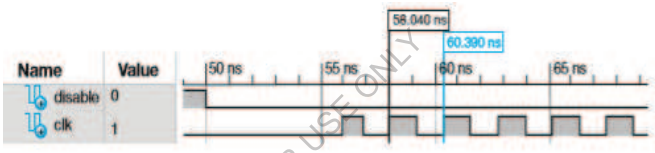


Figure 4.105: Signal of Clock Generation

Discussion

1- Write the PS/NS table for a simple clock.

2- Write an expression for the definition of the term duty cycle (DC). If the on time is 10 ns and the period of the Waveform is 30 ns, calculate the duty cycle as a percentage.

CHAPTER FOUR
VHDL EXPERIEMENTS

D Flip Flops

Aim of Experiment

In this experiment, we will learn the designing D flip flops in VHDL.

Theory

In the previous experiments, we discuss about Latches. Those are the basic building blocks of flip-flops. The flip-flops types are:

- SR Flip-Flop.
- D Flip-Flop.
- JK Flip-Flop.
- T Flip-Flop.

In this experiment we discuss the D flip-flop.

D Flip-Flop

D flip-flop operates with only **positive clock** transitions or **negative clock** transitions. Whereas, **D latch** operates with enable signal. That means, the output of D flip-flop is insensitive to the changes in the input, D except for active transition of the clock signal. The circuit diagram of D flip-flop is shown in the following figure 4.106.

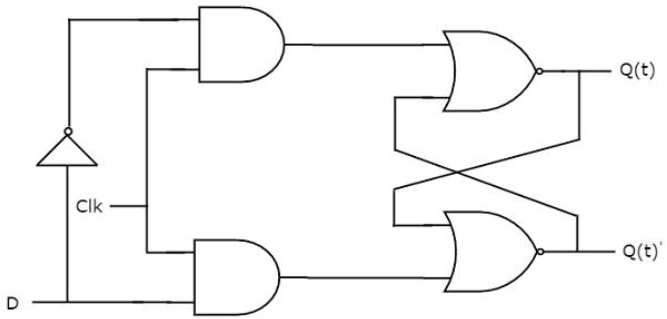


Figure 4.106: D Flip-Flop

This circuit has single input **D** and two outputs **Qt** & **Qt'**. The operation of D flip-flop is similar to D Latch. But this flip-flop affects the outputs only when positive transition of the clock signal is applied instead of active enable.

The following table 4.22 shows the state table of D flip-flop.

Table 4.22: state table of D flip-flop

D	Qt + 1t + 1
0	0
1	1

Therefore, D flip-flop always Hold the information, which is available on data input, D of earlier positive transition of clock signal. From the above state table, we can directly write the next state equation as

$$Q\ t+1\ = D$$

Next state of D flip-flop is always equal to data input, D for every positive transition of the clock signal. Hence, D flip-flops can be used in registers, shift registers and some of the counters.

VHDL Code & Simulation

CODE (CSA method)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity DFF is
    Port ( rst,clk,d : in  STD_LOGIC;
           q : inout  STD_LOGIC);
end DFF;
architecture Behavioral of DFF is
begin
   q <= '0' when rst ='1' else
   d when rising_edge(clk) else
   q;
      --note: "else q" is inferred (so it can be removed)
end Behavioral;
```

CHAPTER FOUR
VHDL EXPERIEMENTS

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY DFF_test IS
END DFF_test;
ARCHITECTURE behavior OF DFF_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT DFF
 PORT(
      rst : IN  std_logic;
      clk : IN  std_logic;
      d : IN  std_logic;
      q : INOUT  std_logic
     );
 END COMPONENT;
--Inputs
signal rst : std_logic := '0';
signal clk : std_logic := '0';
signal d : std_logic := '0';
--BiDirs
signal q : std_logic;
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: DFF PORT MAP (
       rst => rst,
       clk => clk,
       d => d,

        q => q
      );
 -- Clock process definitions
 clk_process :process
 begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
 end process;
 -- Stimulus process
 stim_proc: process
 begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    wait for clk_period*10;
    -- insert stimulus here
 rst <= '1';
 d <= '1';
 wait for 25 ns;
 rst <= '1';
 d <= '0';
 wait for 20 ns;
 rst <= '0';
 d <= '1';
 wait for 30 ns;
 rst <= '0';
 d <= '0';
 wait for 25 ns;
 rst <= '0';
 d <= '0';
 wait for 30 ns;
 rst <= '0';
 d <= '1';
 wait for 30 ns;
 rst <= '1';
 d <= '1';
 wait;
 end process;
 END;
```

In figure 4.107 that shown the signal and simulation of D flip-flop.

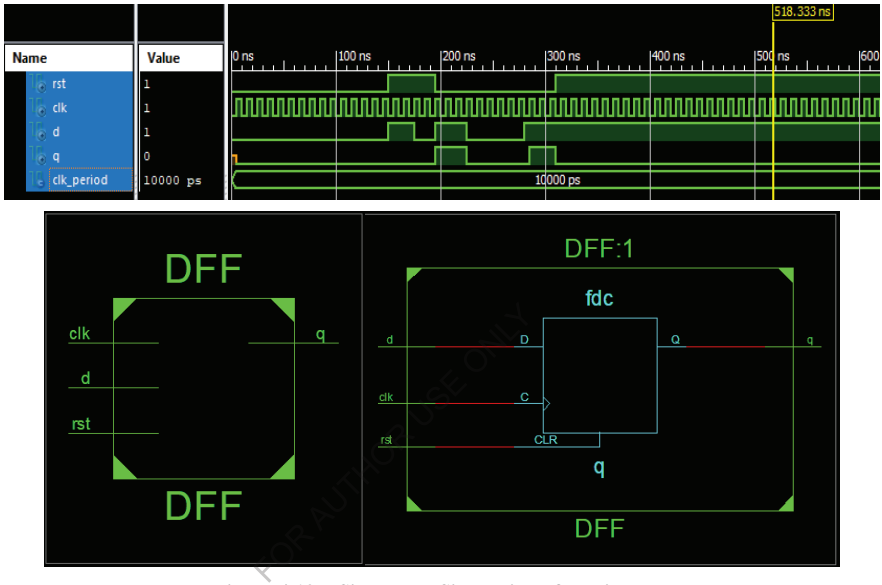


Figure 4.107: Signal and Simulation of D Flip-Flop

Discussion

1- Write and simulate the VHDL code for D flip-flop by using alternate way.

CHAPTER FOUR
VHDL EXPERIEMENTS

SR Flip Flops

Aim of Experiment

In this experiment, we will learn the designing SR flip flops in VHDL.

Theory

SR Flip-Flop

SR flip-flop operates with only **positive clock** transitions or **negative clock** transitions. Whereas, **SR latch** operates with enable signal. The circuit diagram of SR flip-flop is shown in the following figure 4.108.

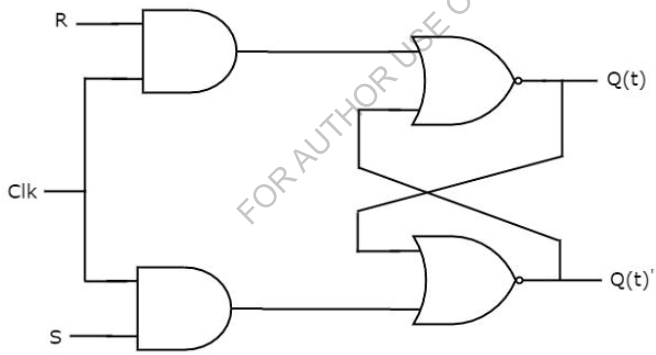


Figure 4.108: Circuit Diagram of SR Flip-Flop

This circuit has two inputs S & R and two outputs **Qt** & **Qt'**. The operation of SR flipflop is similar to SR Latch. But this flip-flop affects the outputs only when positive transition of the clock signal is applied instead of active enable.

The following table 4.23 shows the state table of SR flip-flop.

Table 4.23: State Table of SR flip-flop

S	R	Q $t+1$
0	0	Q t
0	1	0
1	0	1
1	1	-

Here, **Qt** & **Qt+1** is present state & next state respectively. So, **SR flip-flop** can be used for one of these three functions such as **Hold**, **Reset** & **Set** based on the input conditions, when **positive transition** of clock signal is applied. The following table 4.24 shows the characteristic table of SR flip-flop.

Table 4.24: Characteristic Table of SR flip-flop

Present Inputs		Present State	Next State
S	R	Q t	Q $t+1$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

By using three variable **K-Map**, we can get the simplified expression for next state, Q^{t+1}. The three variable K-Map for next state, Q^{t+1} is shown in the following figure 4.109.

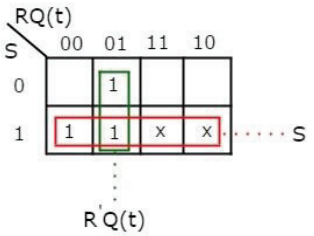


Figure 4.109: K-map for Next State

The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the simplified expression for next state Q t+1 is

$$Q(t+1) = S + R'Q(t)$$

VHDL Code & Simulation

CODE (if Statement)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SRFF is
    Port ( s,r,clk,rst : in  STD_LOGIC;
           q,qb : out  STD_LOGIC);
end SRFF;
architecture Behavioral of SRFF is
begin
process(s,r,clk,rst)
begin
if(rst='1') then
 q<='0';
 qb<='0';
elsif(rising_edge(clk)) then
 if(s/=r) then
  q<=s;
  qb<=r;
 elsif (s='1' and r='1') then
  q<='Z';
  qb<='Z';
 end if;
end if;
end process;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY SRFF_test IS
END SRFF_test;
ARCHITECTURE behavior OF SRFF_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT SRFF
 PORT(
      s : IN  std_logic;
      r : IN  std_logic;
      clk : IN  std_logic;
      rst : IN  std_logic;
      q : OUT  std_logic;
      qb : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal s : std_logic := '0';
signal r : std_logic := '0';
signal clk : std_logic := '0';
signal rst : std_logic := '0';
--Outputs
signal q : std_logic;
signal qb : std_logic;
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
 uut: SRFF PORT MAP (
        s => s,
        r => r,
        clk => clk,
        rst => rst,
        q => q,
        qb => qb
      );
 -- Clock process definitions
 clk_process :process
 begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
 end process;
 -- Stimulus process
 stim_proc: process
 begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    wait for clk_period*10;
    -- insert stimulus here
    rst <= '1';
wait for 50 ns;
rst <= '0';
s <= '0';
r <= '1';
wait for 50 ns;
rst <= '0';
s <= '1';
r <= '0';
wait for 50 ns;
rst <= '0';
s <= '0';
r <= '0';
wait for 50 ns;
rst <= '0';
s <= '1';
r <= '1';
wait;
 end process;
 END;
```

In figure 4.110 that shown the signal and simulation of SR flip-flop.

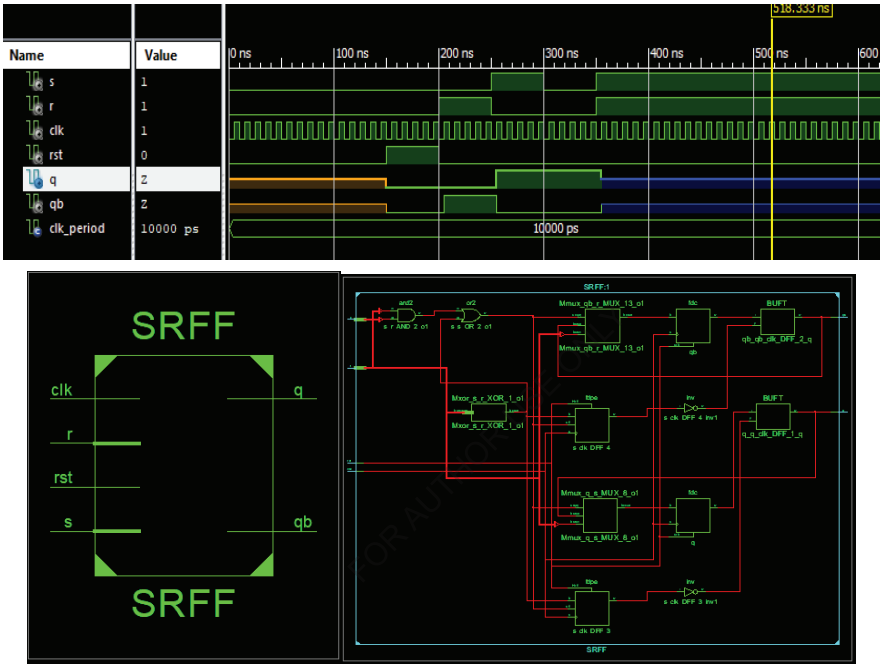


Figure 4.110: Signal and Simulation of SR Flip-Flop

Discussion

1- Write and simulate the VHDL code for SR flip-flop by using alternate way.

CHAPTER FOUR
VHDL EXPERIEMENTS

JK Flip Flops

Aim of Experiment

In this experiment, we will learn the designing JK flip flops in VHDL.

Theory

JK Flip-Flop

JK flip-flop is the modified version of **SR flip-flop**. It operates with only positive clock transitions or negative clock transitions. The circuit diagram of **JK flip-flop** is shown in the following figure 4.111.

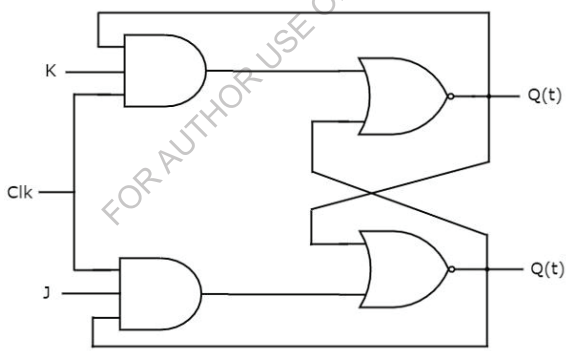


Figure 4.111: Circuit Diagram of JK Flip-Flop

This circuit has two inputs **J** & **K** and two outputs **Qt** & **Qt'**. The operation of **JK flip-flop** is similar to **SR flip-flop**. Here, we considered the inputs of **SR flip-flop** as **S = J Qt'** and **R = K Qt** in order to utilize the modified **SR flip-flop** for 4 combinations of inputs. The following table 4.25 shows the state table of **JK flip-flop**.

Table 4.25: State Table of JK flip-flop

J	K	Q $t+1$
0	0	Q t
0	1	0
1	0	1
1	1	Q t '

Here, **Qt** & **Qt+1** are present state & next state respectively. So, JK flip-flop can be used for one of these four functions such as **Hold**, **Reset**, **Set** & **Complement** of present state based on the input conditions, when positive transition of clock signal is applied. The following table 4.26 shows the characteristic table of JK flip-flop.

Table 4.26: Characteristic Table of JK flip-flop

Present Inputs		Present State	Next State
J	K	Q t	Q $t+1$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

By using three variable **K-Map**, we can get the simplified expression for next state, **Qt+1**. Three variable **K-Map** for next state, **Qt+1** is shown in the following figure 4.112.

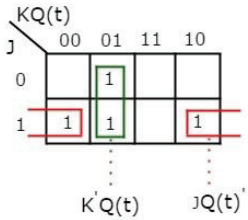


Figure 4.112: K-map JK Flip-Flop

The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the simplified expression for next state **Qt+1** is

$$Q(t+1) = JQ(t)' + K'Q(t)$$

VHDL Code & Simulation

CODE (if Statement)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity JKFF is
    Port ( j,k,clk,rst : in  STD_LOGIC;
           q,qb : out  STD_LOGIC);
end JKFF;
architecture Behavioral of JKFF is
begin
process(j,k,clk,rst)
begin
if(rst='1') then
 q <= '0';
 qb <= '0';
elsif (rising_edge(clk)) then
 if(j/=k) then
  q <= j;
  qb <= not j;
 elsif(j='1' and k='1') then
  q <= not j;
  qb <= j;
 end if;
end if;
end process;
end Behavioral;
```

CHAPTER FOUR
VHDL EXPERIEMENTS

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY JKFF_test IS
END JKFF_test;
ARCHITECTURE behavior OF JKFF_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT JKFF
 PORT(
      j : IN  std_logic;
      k : IN  std_logic;
      clk : IN  std_logic;
      rst : IN  std_logic;
      q : OUT  std_logic;
      qb : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal j : std_logic := '0';
signal k : std_logic := '0';
signal clk : std_logic := '0';
signal rst : std_logic := '0';
--Outputs
signal q : std_logic;
signal qb : std_logic;
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
 uut: JKFF PORT MAP (
        j => j,
        k => k,
        clk => clk,
        rst => rst,
        q => q,
        qb => qb
      );

 -- Clock process definitions
 clk_process :process
 begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
 end process;
 -- Stimulus process
 stim_proc: process
 begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    wait for clk_period*10;
    -- insert stimulus here
 rst <= '1';
wait for 50 ns;
rst <= '0';
j <= '0';
k <= '1';
wait for 50 ns;
rst <= '0';
j <= '1';
k <= '0';
wait for 50 ns;
rst <= '0';
j <= '1';
k <= '1';
wait for 50 ns;
rst <= '0';
j <= '0';
k <= '0';
    wait;
 end process;
 END;
```

In figure 4.113 that shown the signal and simulation of JK flip-flop.

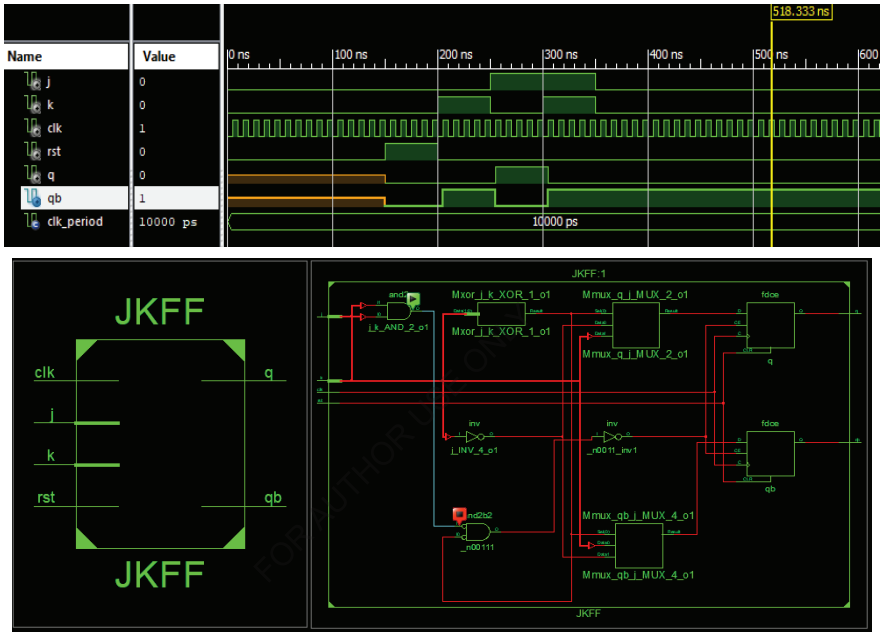


Figure 4.113: Signal and Simulation of JK Flip-Flop

Discussion

1- Write and simulate the VHDL code for JK flip-flop by using alternate way.

CHAPTER FOUR
VHDL EXPERIEMENTS

T Flip Flops

Aim of Experiment

In this experiment, we will learn the designing T flip flops in VHDL.

Theory

T Flip-Flop

T flip-flop is the simplified version of **JK flip-flop**. It is obtained by connecting the same input '**T**' to both inputs of **JK flip-flop**. It operates with only positive clock transitions or negative clock transitions. The circuit diagram of T flip-flop is shown in the following figure 4.114.

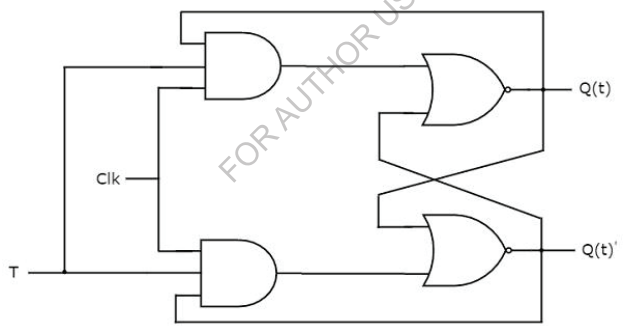


Figure 4.114: Circuit Diagram of T Flip-Flop

This circuit has single input **T** and two outputs **Qt** & **Qt'**. The operation of **T flip-flop** is same as that of **JK flip-flop**. Here, we considered the inputs of **JK flip-flop** as **J = T** and **K = T** in order to utilize the modified **JK flip-flop** for 2 combinations of inputs. So, we eliminated the other two combinations of **J** & **K**, for which those two values are complement to each other in T flip-flop.

The following table 4.27 shows the state table of T flip-flop.

Table 4.27: State Table of T flip-flop

D	Q $t+1$
0	Q t
1	Q t '

Here, **Qt** & **Qt+1** are present state & next state respectively. So, T flip-flop can be used for one of these two functions such as Hold, & Complement of present state based on the input conditions, when positive transition of clock signal is applied. The following table 4.28 shows the characteristic table of T flip-flop.

Table 4.28: Characteristic Table of T flip-flop

Inputs	Present State	Next State
T	Q t	Q $t+1$
0	0	0
0	1	1
1	0	1
1	1	0

From the above characteristic table, we can directly write the next state equation as

$$Q(t+1) = T'Q(t) + TQ(t)'$$

$$\Rightarrow Q(t+1) = T \oplus Q(t)$$

The output of **T flip-flop** always **toggles** for every positive transition of the clock signal, when input T remains at logic High 1. Hence, T flip-flop can be used in **counters**.

In this chapter, we implemented various flip-flops by providing the cross coupling between **NOR gates**. Similarly, you can implement these flip-flops by using **NAND gates**.

VHDL Code & Simulation

CODE (if Statement)

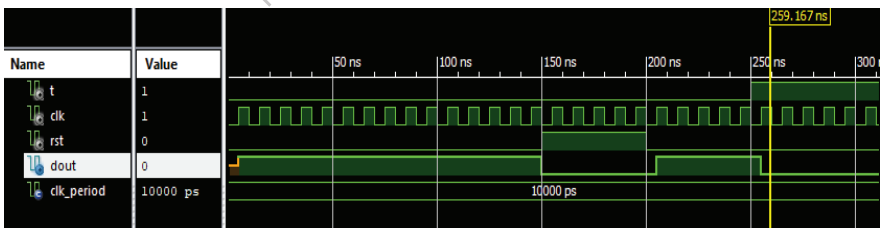
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TFF is
    Port ( t,clk,rst : in  STD_LOGIC;
           dout : out  STD_LOGIC);
end TFF;
architecture Behavioral of TFF is
begin
process(rst,clk,t)
begin
if (rst='1') then
dout<='0';
elsif(rising_edge(clk)) then
dout<=not t;
end if;
end process;
end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY TFF_test IS
END TFF_test;
ARCHITECTURE behavior OF TFF_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT TFF
 PORT(
      t : IN  std_logic;
      clk : IN  std_logic;
      rst : IN  std_logic;
      dout : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal t : std_logic := '0';
signal clk : std_logic := '0';
signal rst : std_logic := '0';
--Outputs
signal dout : std_logic;
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: TFF PORT MAP (
       t => t,
       clk => clk,
       rst => rst,
```

```
        dout => dout
      );
 -- Clock process definitions
 clk_process :process
 begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
 end process;
 -- Stimulus process
 stim_proc: process
 begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    wait for clk_period*10;
    -- insert stimulus here
 rst <= '1';
wait for 50 ns;
rst <= '0';
t <= '0';
wait for 50 ns;
rst <= '0';
t <= '1';
    wait;
 end process;
 END;
```


In figure 4.115 that shown the signal and simulation of T flip-flop.



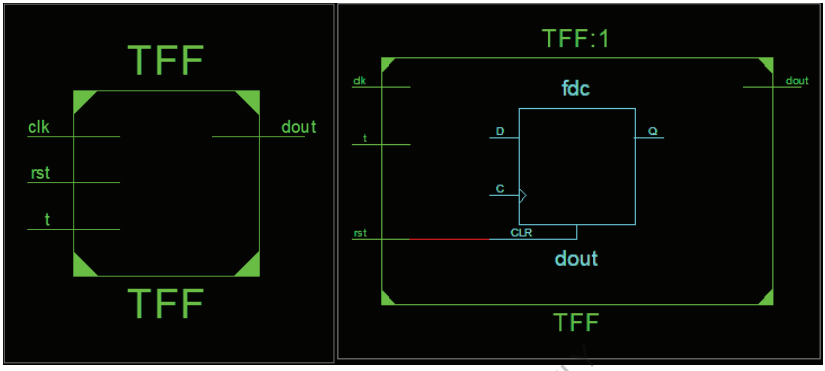


Figure 4.115: Signal and Simulation of T Flip-Flop

Discussion

1- Write and simulate the VHDL code for T flip-flop by using alternate way.

CHAPTER FOUR
VHDL EXPERIEMENTS

Conventional Counters

Aim of Experiment

In this experiment, we will learn the designing conventional Counters in VHDL.

Theory

Counters

Simple counters have a fixed counting sequence and therefore do not have external inputs to change the counting sequence. **Complex counters** have external inputs that allow the counting sequence to be changed. **Simple counters** are often named by the sequence as well as the direction in which they count. The sequence can be a straight binary sequence, a Gray code sequence, or any other sequence in addition to counting forward (or up) or counting in reverse (or down). The sequence can be a **one-hot sequence** such that all bits are off except one that is on; hence, the term one-hot. A **one-cold sequence** can be used as well. Counters by definition are state machines, or finite state machines, because they contain only a finite number of states in their counting sequence.

A simple counter is a simple state machine (**SSM**) or simple finite state machine (**SFSM**). Figure 4.116a shows a **counting sequence diagram** (or **state sequence diagram**) for a **binary up** counter (**2 bits**) with a clock independent reset input RST for design entity Counter1. An equivalent state diagram for the counter is shown in Figure 4.116b. When the asynchronous input **RST** is asserted or equal to 1, the counter goes to state 00. When the asynchronous input **RST** is not asserted or is 0, the counter responds to the clock. Each time the clock ticks, the counter changes from its present-state value to its next-state value—that is, 00 to 01, then 01 to 10, then 10 to 11, then 11 back to 00—following the state-transition lines. The clock signal is not shown in the counting sequence diagram or in a state diagram; that is, it is implied.

Figure 4.116 shows an **asynchronous** reset signal **RST**, which resets the counter to 00 independent of the clock. From this information, we can draw the clouds-of-logic circuit for the binary up counter as shown in Figure 4.117. Each state represents the output of a D flip-flop, so two flipflops are drawn with the outputs **Q0** and **Q1**. D flip-flop Q0 has a D input signal labeled D0, and D flip-flop Q1 has a D input signal labeled D1. The D inputs provide the next-state values for the D flip-flops.

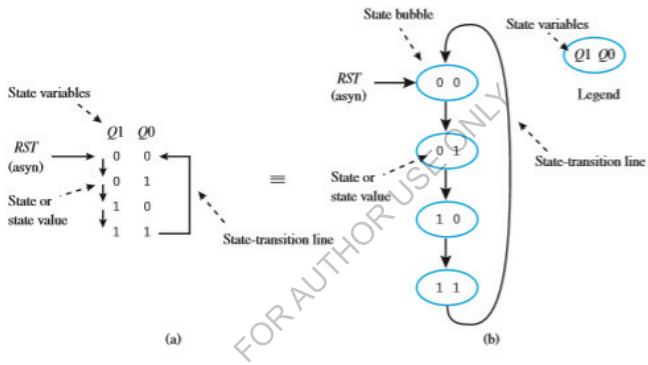


Figure 4.116: Binary up counter (2 bits) for design entity Counter1: (a) counting sequence diagram; (b) equivalent state diagram

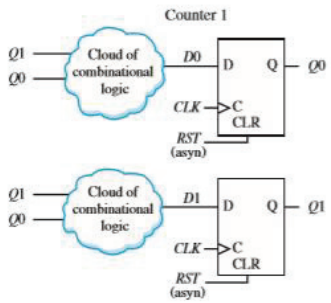


Figure 4.117: Binary up counter (2 bits) with clouds of combinational logic to be determined

The excitation equation for a single D flip-flop depends on the output transitions made by the flip-flop from its present-state output Q(PS) to its next-state output Q(NS), or Q(PS) S Q(NS). Table 4.29 summarizes all possible output transitions for a single D flip-flop. Table 4.29 that the D column is the equivalent to the next-state value, or D = Q1 = Q(NS).

Table 4.29: Flip-flop output transitions and required D input

Q(PS)	*Q*(NS)	Comment	*D*
0	0	Hold 0 transition	0
0	1	Set transition	1
1	0	Clear transition	0
1	1	Hold 1 transition	1

Figure 4.118a shows the procedure for using the **Set OR Hold 1** method via a counting sequence diagram. Figure 4.118b shows the classical procedure for using a **PS/NS** (present-state/ next-state) table to obtain the same D input excitation equations for the **binary up counter**, which we will refer to as the conventional method.

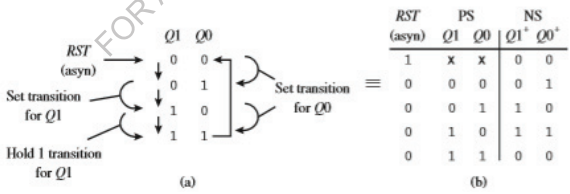


Figure 4.118: (a) Counting sequence diagram showing all set and hold 1 transition for the binary up counter; (b) equivalent PS/NS table

$D1 = Q1^{+} = \overline{Q1}.Q0 + Q1.\overline{Q0}$(observe that D1 is an XOR function).

$D0 = Q01 = \overline{Q1}.\overline{Q0} + Q1.\overline{Q0} = Q0.$

The excitation equations for the D inputs are used to fill in the clouds of combinational logic for the binary up counter as shown in the Figure 4.119.

CHAPTER FOUR
VHDL EXPERIEMENTS

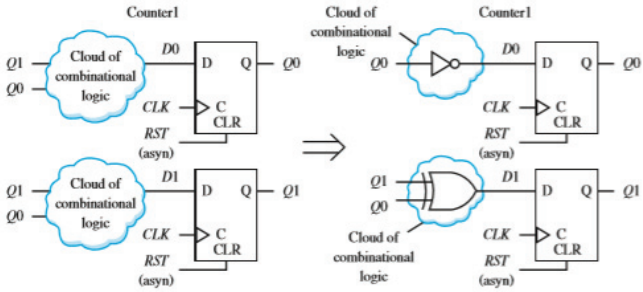


Figure 4.119: Schematic for a binary up counter (2 bits) with wire connections implied by signal name association

VHDL Code & Simulation

CODE (2 Bits Counter)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Counter1 is
    Port ( rst,clk : in  STD_LOGIC;
           q1,q0 : inout  STD_LOGIC);
end Counter1;
architecture Behavioral of Counter1 is
signal d1, d0: std_logic;
begin
d1 <= q1 xor q0;
d0 <= not q0;
q1_output: process (clk, rst)
   begin
    if rst = '1' then q1 <= '0';
    elsif rising_edge (clk) then q1 <= d1;
                            --or q1 <= q1 xor q0
   end if;
   end process;
q0_output: process (clk, rst)
            begin
             if rst = '1' then q0 <= '0';
    elsif rising_edge (clk) then q0 <= d0;
                            --or q1 <= not q0

   end if;
   end process;
end Behavioral;
```

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Counter1_test IS
END Counter1_test;
ARCHITECTURE behavior OF Counter1_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Counter1
 PORT(
      rst : IN  std_logic;
      clk : IN  std_logic;
      q1 : INOUT  std_logic;
      q0 : INOUT  std_logic
     );
 END COMPONENT;
--Inputs
signal rst : std_logic := '0';
signal clk : std_logic := '0';
--BiDirs
signal q1 : std_logic;
signal q0 : std_logic;
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Counter1 PORT MAP (
       rst => rst,
       clk => clk,
       q1 => q1,
       q0 => q0
     );
-- Clock process definitions
clk_process :process
begin
   clk <= '0';
   wait for clk_period/2;
   clk <= '1';
   wait for clk_period/2;
end process;
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   wait for clk_period*10;
   -- insert stimulus here
   rst <= '1';
   wait for 50 ns;
   rst <= '0';
   wait for 100 ns;
   rst <= '1';
   wait;
end process;
END;
```

In figure 4.120 that shown the signal and simulation of Conventional counter up 2 bits.

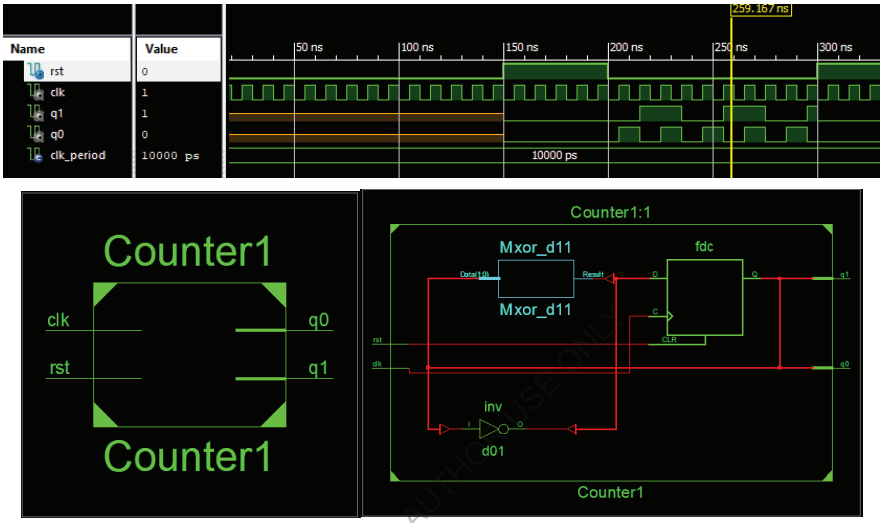


Figure 4.120: Signal and Simulation of Conventional up 2 bits counter

Figure 4.121 shows an asynchronous signal INIT (INITialize), which initializes the counter to 0001 (one of its one-hot states) independent of the clock. Using the Set OR Hold 1 equation, we can write the excitation equations for the D inputs for each of the four flip-flops Q3, Q2, Q1, and Q0 that are required for the design.

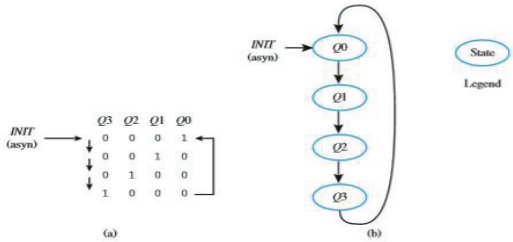


Figure 4.121: One-hot up counter (4 bits): (a) counting sequence diagram; (b) equivalent state diagram

After we obtain the excitation equations for the D inputs, we can fill in the clouds of combinational logic for the D flip-flops and write the proper VHDL for the design. From Figure 6.7a or 7b, we can write the excitation equations for the D inputs by inspection using the Set OR Hold 1 equation as shown here:

$$D3 = \overline{Q3}\cdot Q2\cdot\overline{Q1}\cdot\overline{Q0} = Q2$$

$$D2 = \overline{Q3}\cdot\overline{Q2}\cdot Q1\cdot\overline{Q0} = Q1$$

$$D1 = \overline{Q3}\cdot\overline{Q2}\cdot\overline{Q1}\cdot Q0 = Q0$$

$$D0 = Q3\cdot\overline{Q2}\cdot\overline{Q1}\cdot\overline{Q0} = Q3$$

The D excitation equations allow us to draw the schematic shown in Figure 4.122.

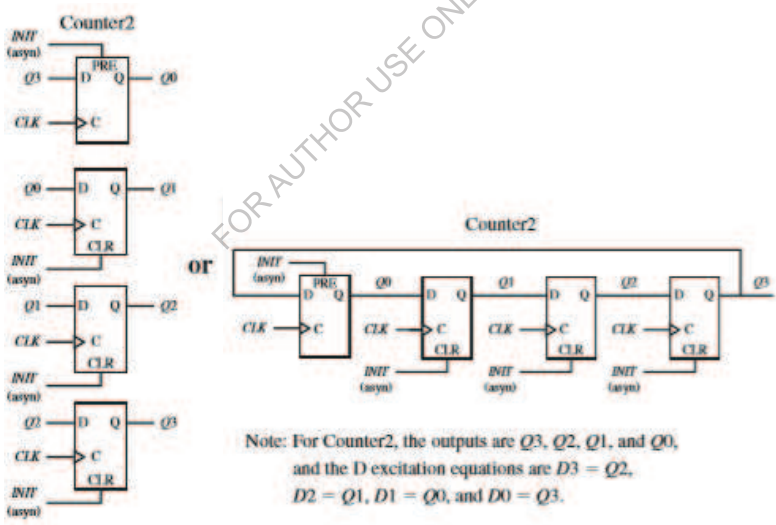


Figure 4.122: Schematic for a one-hot up counter (4 bits) for design entity Counter2

VHDL Code & Simulation

CODE (4 Bits Counter)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Counter2 is
    Port ( init, clk : in  STD_LOGIC;
           q3,q2,q1,q0 : inout  STD_LOGIC);
end Counter2;
architecture Behavioral of Counter2 is
begin
process (init,clk)
begin
if init ='1' then q3 <= '0'; q2 <= '0'; q1 <= '0'; q0 <= '1';
elsif rising_edge(clk) then q3 <= q2; q2 <= q1; q1 <= q0; q0 <= q3;
end if;
end process;
end Behavioral;
```

Test Bench

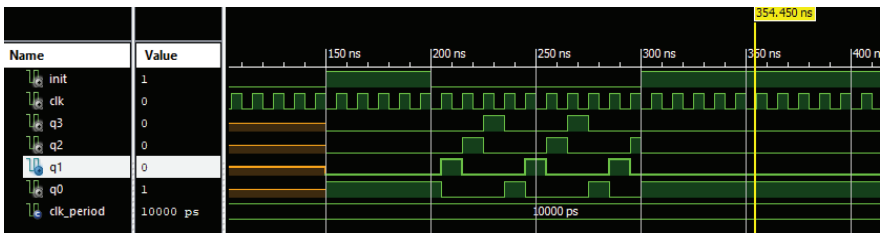
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Counter2_test IS
END Counter2_test;
ARCHITECTURE behavior OF Counter2_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Counter2
 PORT(
      init : IN  std_logic;
      clk : IN  std_logic;
      q3 : INOUT  std_logic;
      q2 : INOUT  std_logic;
      q1 : INOUT  std_logic;
      q0 : INOUT  std_logic
     );
 END COMPONENT;
--Inputs
signal init : std_logic := '0';
signal clk : std_logic := '0';
--BiDirs
signal q3 : std_logic;
signal q2 : std_logic;
signal q1 : std_logic;
signal q0 : std_logic;
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
```

```
uut: Counter2 PORT MAP (
       init => init,
       clk => clk,
       q3 => q3,
       q2 => q2,
       q1 => q1,
       q0 => q0
     );

-- Clock process definitions
clk_process :process
begin
   clk <= '0';
   wait for clk_period/2;
   clk <= '1';
   wait for clk_period/2;
end process;
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   wait for clk_period*10;
   -- insert stimulus here
init <= '0';
wait for 50 ns;
init <= '1';
wait for 100 ns;
```

```
init <= '0';
   wait;
end process;
END;
```


In figure 4.123 that shown the signal and simulation of Ring counter up 4 bits.



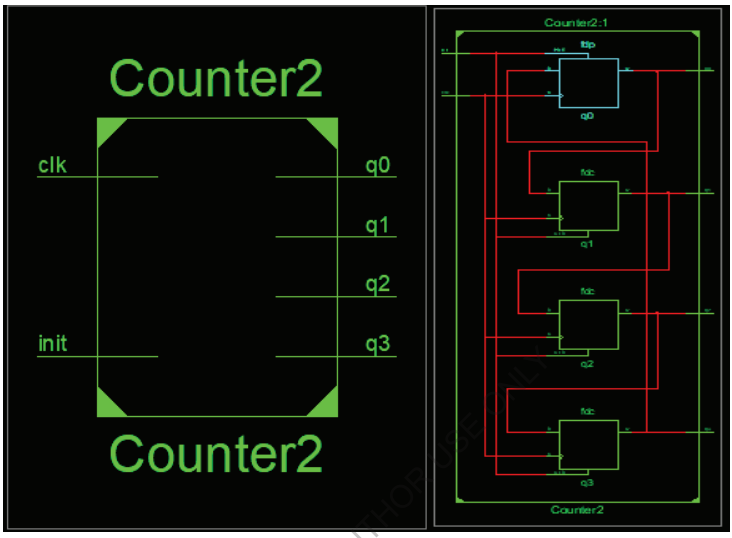


Figure 4.123: Signal and Simulation of Ring counter up 4 bits

Discussion

1- Write and simulate the VHDL code for Conventional down 2 bits counter.

2- Write and simulate the VHDL code for Ring 5 bits counter.

CHAPTER FOUR
VHDL EXPERIEMENTS

Nonconventional Counters

Aim of Experiment

In this experiment, we will learn the designing nonconventional Counters in VHDL.

Theory

Nonconventional Counters

Most simple counters are **conventional counters**, because they do not have repeating states. Each state in a conventional counter has a different state value. The simple counters we discussed in the previous experiment were **conventional counters**. A simple counter that has repeating states (states with the same state value) in its counting sequence is a **nonconventional (NC)** counter. It is necessary to add additional **flip-flops** to a nonconventional counter to differentiate between the repeating states; that is, the additional flip-flops are used to convert a nonconventional counter to a conventional counter. Only one additional flip-flop is required for two repeating states. For additional repeating states, you must determine the number of additional flip-flops to add so that each state has a different state value—that is, remove all repeating states values so there are no repeating states.

Figure 4.124a shows a nonconventional counter (**2 bits**) with the repeating state value **10**. Adding state variable **Q2** (an additional D flip-flop Q2) removes the repeating state value 10 as shown in Figure 4.124b. The states in Figure 4.124b are now 001, 110, and 010. Figure 4.124c shows Figure 4.124b written in the format of a **PS/NS** table. Figures 4.124b and 4.124c do not have repeating states because **Q2** has been added to remove the repeating states in Figure 4.124a. In other words, an additional D flip-flop **Q2** was added to the counting sequence diagram to differentiate between the repeating states of **Q1 Q0**.

CHAPTER FOUR
VHDL EXPERIEMENTS

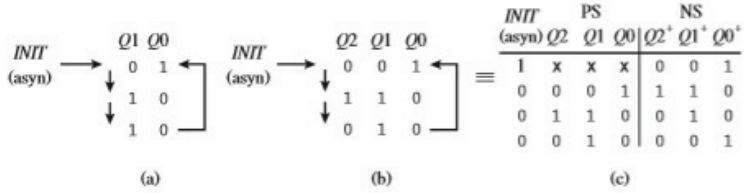


Figure 4.124: Nonconventional counter: (a) counting sequence diagram for NC counter (2 bits); (b) additional D flip-flop Q2 to differentiate between repeating states of Q1Q0; (c) equivalent PS/NS table

When the asynchronous input **INIT** is asserted or pulled high or to a 1, the counter goes to state **001**. When the asynchronous input **INIT** is not asserted the counter responds to the **clock**. Each time the clock ticks, the counter changes from its present-state value to its next state value—that is, 001 to 110, then 110 to 010, then 010 back to 001, and so on. The **counting sequence diagram** and the **PS/NS** table clearly show these transitions. Using the Set OR Hold 1 equation, we can write the **D2**, **D1**, and **D0** excitation equations from Figure 4.124b as follows:

$$D2 = \overline{Q2}\cdot\overline{Q1}\cdot Q0$$
$$D1 = \overline{Q2}\cdot\overline{Q1}\cdot Q0 + Q2\cdot Q1\cdot\overline{Q0}$$
$$D0 = \overline{Q2}\cdot Q1\cdot\overline{Q0}$$

To simplify drawings, it is sometimes convenient to represent each combinational logic circuit as a rectangular box as shown in Figure 4.125b. The D excitation equation for each rectangular box may be listed separately or inside each box as shown in Figure 4.125b (**NOTE: ! is NOT, * is AND, and 1 is OR**).

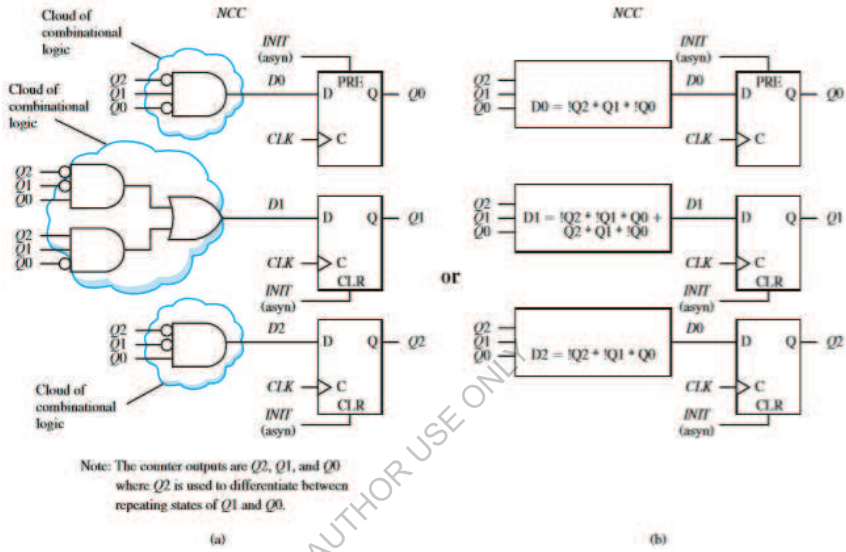


Figure 4.125: Schematic for nonconventional counter: (a) with gates and DFFs; (b) with rectangular boxes and DFFs

VHDL Code & Simulation

CODE (2 Bits NNC Counter)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity NNC is
    Port ( init,clk : in  STD_LOGIC;
           q : inout  STD_LOGIC_VECTOR (2 downto 0));
end NNC;
architecture Behavioral of NNC is
signal d2,d1,d0: std_logic;
begin
d2 <= (not q(2) and not q(1) and q(0));
d1 <= (not q(2) and not q(1) and q(0)) or
      (q(2) and q(1) and not q(0));
d0 <= (not q(2) and q(1) and not q(0));
q <= "001" when init ='1' else
      (d2,d1,d0) when rising_edge (clk);
end Behavioral;
```

CHAPTER FOUR
VHDL EXPERIEMENTS

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY NCC_test IS
END NCC_test;
ARCHITECTURE behavior OF NCC_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT NNC
 PORT(
      init : IN  std_logic;
      clk : IN  std_logic;
      q : INOUT  std_logic_vector(2 downto 0)
     );
 END COMPONENT;
--Inputs
signal init : std_logic := '0';
signal clk : std_logic := '0';
--BiDirs
signal q : std_logic_vector(2 downto 0);
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: NNC PORT MAP (
       init => init,
       clk => clk,
       q => q
     );
-- Clock process definitions

clk_process :process
begin
   clk <= '0';
   wait for clk_period/2;
   clk <= '1';
   wait for clk_period/2;
end process;
-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 50 ns.
   wait for 50 ns;
   wait for clk_period*10;
   -- insert stimulus here
init <= '1';
wait for 50 ns;
init <= '0';
wait for 100 ns;
init <= '0';
   wait;
end process;
END;
```

In figure 4.126 that shown the signal and simulation of 2 bits NNC counter.

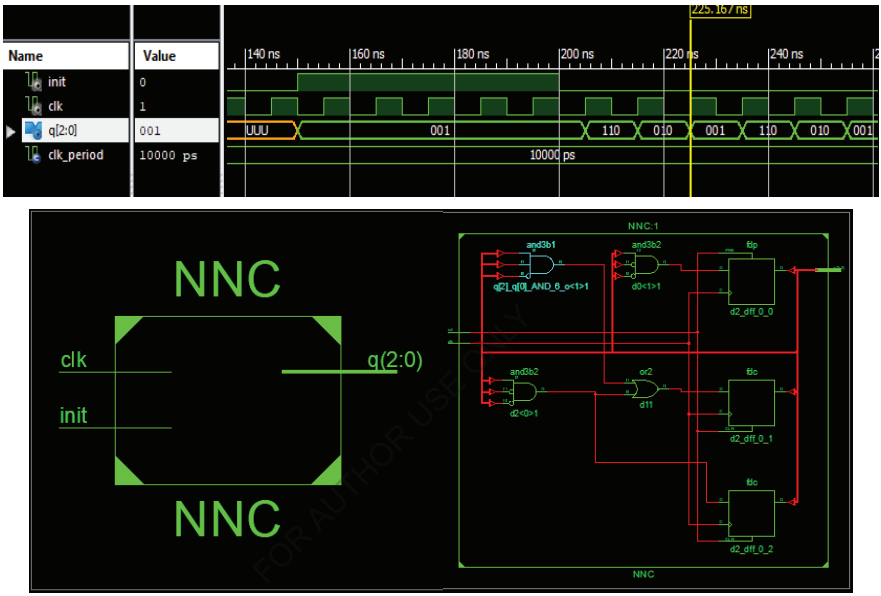


Figure 4.126: Signal and Simulation of 2 bits NNC Counter

We can make the 2 bits NCC counter by using another way called Arithmetic Method.

CODE (2 Bits NNC Counter AM)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity Counter1_AM is
    Port ( rst,clk : in  STD_LOGIC;
           q : inout  STD_LOGIC_VECTOR (1 downto 0));
end Counter1_AM;
architecture Behavioral of Counter1_AM is
begin
process (clk, rst)
begin
   if rst = '1' then q <="00";
   elsif rising_edge (clk) then q <= q+1;
   end if;
end process;
end Behavioral;
```

CHAPTER FOUR
VHDL EXPERIEMENTS

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Counter1_AM_test IS
END Counter1_AM_test;
ARCHITECTURE behavior OF Counter1_AM_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Counter1_AM
 PORT(
      rst : IN  std_logic;
      clk : IN  std_logic;
      q : INOUT  std_logic_vector(1 downto 0)
     );
 END COMPONENT;
--Inputs
signal rst : std_logic := '0';
signal clk : std_logic := '0';
--BiDirs
signal q : std_logic_vector(1 downto 0);
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Counter1_AM PORT MAP (
       rst => rst,
       clk => clk,
       q => q
     );
-- Clock process definitions

  clk_process :process
  begin
     clk <= '0';
     wait for clk_period/2;
     clk <= '1';
     wait for clk_period/2;
  end process;
  -- Stimulus process
  stim_proc: process
  begin
     -- hold reset state for 50 ns.
     wait for 50 ns;
     wait for clk_period*10;
     -- insert stimulus here
  rst <='1';
  wait for 50 ns;
  rst <='0';
  wait for 100 ns;
  rst <='1';
     wait;
  end process;
  END;
```

In figure 4.127 that shown the signal and simulation of 2 bits NNC counter by using arithmetic method.

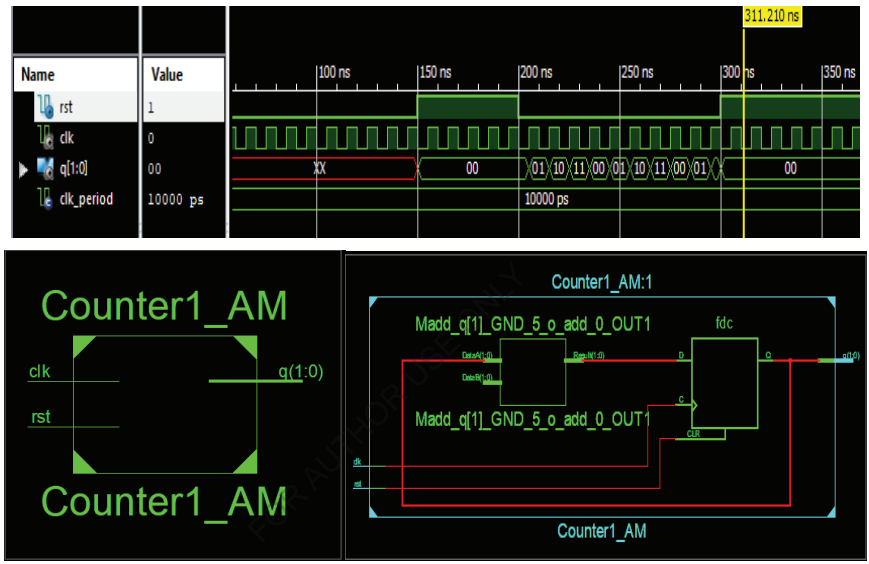


Figure 4.127: Signal and Simulation of 2 bits NNC Counter Arithmetic Method

Discussion

1- Write and simulate the VHDL code for Conventional down 2 bits NCC counter by using PS/NS tabular method.

2- A design specific cation for a simple nonconventional counter is shown in the counting sequence diagram in Figure 4.128. Add the required number of FFs to the counting sequence diagram so that the counter does not have repeating state values.

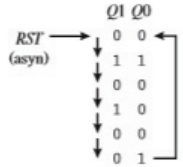


Figure 4.128: NCC Counter

CHAPTER FOUR
VHDL EXPERIEMENTS

Gray Code Counter

Aim of Experiment

In this experiment, we will learn the designing gray code counter in VHDL.

Theory

Gray Code Counters

This tutorial is about designing an N-bit gray counter in VHDL. **A gray counter** changes **1-bit** only during one state to another state transition. The counter is same like the normal incremental counter. The only difference is in binary representation. Take a look at the below figure 4.129.

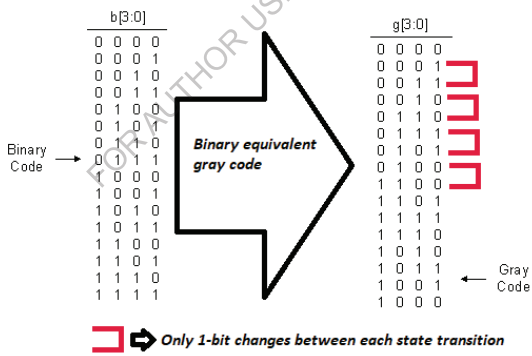


Figure 4.129: Binary to Gray Code State Transition

In the above figure 4.129 on the left side binary representation of the whole numbers starting from **0** and ending on **9** is shown. On the right-hand side, the equivalent of the binary to **gray code** is shown. Notice the difference only **1-bit** changes between the gray code binary values.

CHAPTER FOUR
VHDL EXPERIEMENTS

VHDL Code & Simulation

CODE (0-7 Gray Counter)

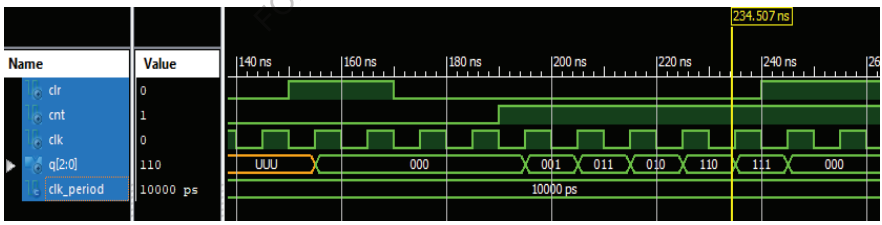
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Gray_Counter is
    Port ( clr,cnt,clk : in  STD_LOGIC;
           q : out  STD_LOGIC_VECTOR (2 downto 0));
end Gray_Counter;
architecture Behavioral of Gray_Counter is
signal reg :STD_LOGIC_VECTOR (2 downto 0);
begin
process (clk)
   begin
      if clk ' event and clk ='1' then
         if clr ='1' then reg <= "000";
         elsif cnt = '1' then
          case reg is
          when "000" => reg <= "001";
          when "001" => reg <= "011";
          when "011" => reg <= "010";
          when "010" => reg <= "110";
          when "110" => reg <= "111";
          when "111" => reg <= "101";
          when "101" => reg <= "100";
          when "100" => reg <= "000";
          when others => reg <= "000";
          end case;
      end if;
   end if;
end process;
q <= reg;
end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Gray_Counter_test IS
END Gray_Counter_test;
ARCHITECTURE behavior OF Gray_Counter_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Gray_Counter
 PORT(
      clr : IN  std_logic;
      cnt : IN  std_logic;
      clk : IN  std_logic;
      q : OUT  std_logic_vector(2 downto 0)
     );
 END COMPONENT;
--Inputs
signal clr : std_logic := '0';
signal cnt : std_logic := '0';
signal clk : std_logic := '0';
--Outputs
signal q : std_logic_vector(2 downto 0);
-- Clock period definitions
constant clk_period : time := 10 ns;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Gray_Counter PORT MAP (
       clr => clr,
       cnt => cnt,
       clk => clk,
```

```
        q => q
      );
-- Clock process definitions
clk_process :process
begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
end process;
-- Stimulus process
stim_proc: process
begin
    -- hold reset state for 50 ns.
    wait for 50 ns;
    wait for clk_period*10;
    -- insert stimulus here
    clr <= '1';
    wait for 20 ns;
    clr <= '0';
    cnt <= '0';
    wait for 20 ns;
    cnt <= '1';
    wait for 50 ns;
    clr <= '1';
    wait;
end process;
END;
```


In figure 4.130 that shown the signal and simulation of 0-7 Gray counter.



CHAPTER FOUR
VHDL EXPERIEMENTS

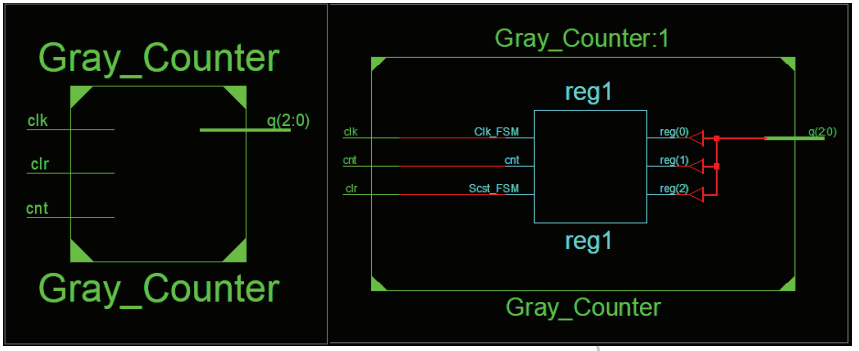


Figure 4.130: Signal and Simulation of 0-7 Gray Counter

N-bit gray counter

It's easy to design a fixed sized gray counter using case statement. But what if the size of the gray counter increases? With size of gray counter, the VHDL code length increases which in any case is not feasible. So, we have to come up with some cleaver and optimized methods to reduce VHDL code size and achieve greater speeds. An example of four-bit gray counter with VHDL case statement is given Previously. Look at the size of code. What if it we increase it to 6-bit?

The main **gray counter** entity VHDL code is given below. The counter is designed on **current** and **next state** logic. I assume that you know what is meant by current state and next state logic. I also assume that you are well aware of **FSM** (finite state machine). If you don't know about finite state machine and how it works? Then before moving any future I will suggest you to first take some tutorials on fine state machine and how it works?

In the main entity first a generic **N: integer: =6** is defined. The generic variable is used to declare the size of the gray counter. In my case I want a **6-bit gray counter** so I initialized it with **6**. You can change it according to your need. **Clock**, **reset** and **enable** signals are

input to the entity and the only output is our gray code. Notice that the output port size is dependent on the generic variable **N**. clock, reset and enable are 1-bit inputs.

In the architecture part few signals are defined. These signals are used to hold the values manipulated for gray counter by finite state machine. The reset is **synchronous** which means that reset execute on positive edge of the clock. In the absence of clock, you can't reset the system.

After reset the current state value is **000000**. We then xor this current state with the **5** lsb (least significant bits) of current state while concatenating the 0 at msb(most significant bit) of current state. & is concatenating operator. The result is placed in hold variable **hold <= Currstate XOR ('0' & hold(N-1 DOWNTO 1));**.

In the next statement we increment the hold by 1. The result is placed in next_hold variable next_hold <= std_logic_vector(unsigned(hold) + 1);.

In the third statement we perform the same concatenation and logical xor but this time on next_hold variable. The result is placed in Next state variable Next state <= next_hold XOR ('0' & next_hold(N-1 DOWNTO 1)); .

The current state is assigned to the output and next state is calculated. This next state is transferred to current state on next rising edge of clock.

VHDL Code & Simulation

CODE (N Bits Gray Counter)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
entity Gray_Counter_N is
     GENERIC (N: integer := 6);
    Port ( Clk,Rst,En : in  STD_LOGIC;
           output : out  STD_LOGIC_vector (N-1 DOWNTO 0));
end Gray_Counter_N;
architecture Behavioral of Gray_Counter_N is
SIGNAL Currstate, Nextstate, hold, next_hold: std_logic_vector (N-1 DOWNTO 0);
begin
StateReg: PROCESS (Clk)
  BEGIN
    IF (Clk = '1' AND Clk'EVENT) THEN
      IF (Rst = '1') THEN
        Currstate <= (OTHERS =>'0');
      ELSIF (En = '1') THEN
        Currstate <= Nextstate;
      END IF;
    END IF;
  END PROCESS;
  hold <= Currstate XOR ('0' & hold(N-1 DOWNTO 1));
  next_hold <= std_logic_vector(unsigned(hold) + 1);
  Nextstate <= next_hold XOR ('0' & next_hold(N-1 DOWNTO 1));
  output <= Currstate;
end Behavioral;
```


Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Gray_Counter_N_test  IS
END Gray_Counter_N_test;

ARCHITECTURE behavior OF Gray_Counter_N_test  IS
  COMPONENT Gray_Counter_N IS
    GENERIC (N: integer := 6);
    PORT (Clk, Rst, En: IN std_logic;
          output: OUT std_logic_vector (N-1 DOWNTO 0));
  END COMPONENT;
  SIGNAL Clk_s, Rst_s, En_s: std_logic;
  SIGNAL output_s: std_logic_vector(5 DOWNTO 0);
BEGIN
  CompToTest: Gray_Counter_N GENERIC MAP (6) PORT MAP (Clk_s, Rst_s, En_s, output_s);
  Clk_proc: PROCESS
  BEGIN
    Clk_s <= '1';
    WAIT FOR 10 ns;
    Clk_s <= '0';
    WAIT FOR 10 ns;
  END PROCESS clk_proc;
  Vector_proc: PROCESS
  BEGIN
    Rst_s <= '1';
    WAIT UNTIL Clk_s='1' AND Clk_s'EVENT;
    WAIT FOR 5 NS;
```

```
    Rst_s <= '0';
    En_s <= '1';
    FOR index IN 0 To 3 LOOP
      WAIT UNTIL Clk_s='1' AND Clk_s'EVENT;
    END LOOP;
    WAIT FOR 5 NS;
    WAIT;
  END PROCESS Vector_proc;
END;
```

In figure 4.131 that shown the signal of N bits Gray counter.

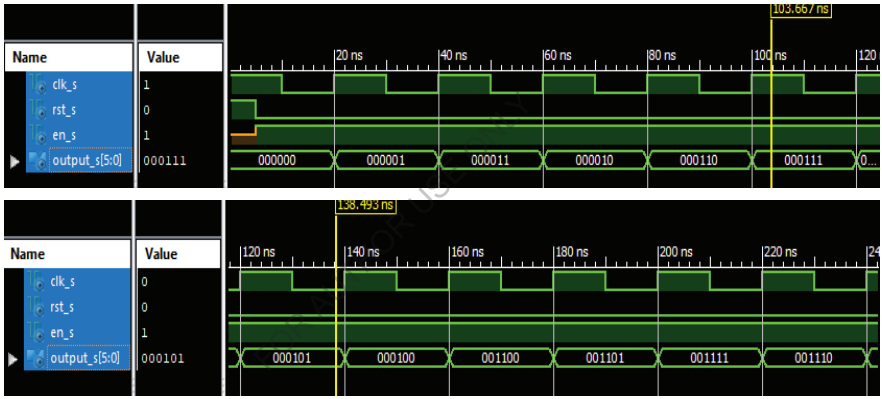
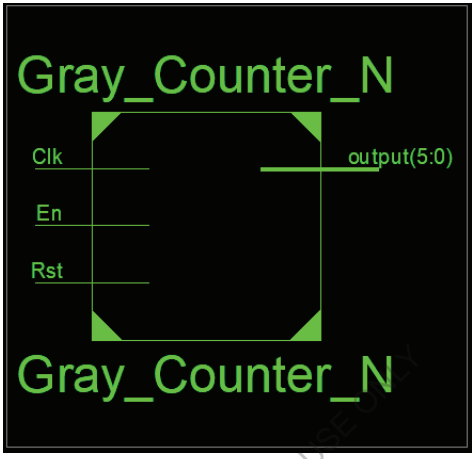


Figure 4.131: Signal of N Bits Gray Counter

In figure 4.132 that shown the simulation of N bits Gray counter.



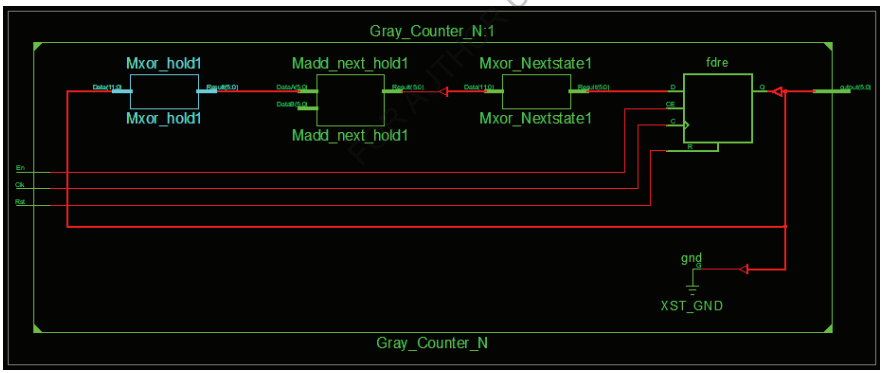


Figure 4.132: Simulation of N Bits Gray Counter

Discussion

1- Write and simulate the VHDL code for (8 4 2 1 or BCD code) counter from 0-15. Then write and simulate the same counter but from 15-0 (down).

CHAPTER FOUR
VHDL EXPERIEMENTS

Three State Outputs and Disconnected State

Aim of Experiment

In this experiment, we will learn the designing three-state outputs and the disconnected state in VHDL.

Theory

Three State Outputs and Disconnected State

Each of the integrated circuits (**ICs**) shown in Figure 4.133 has an output enable (**OE**) input that allows the ICs output signal to be **tri-stated**. This third-state output condition is a high impedance state in which the output acts as though it is disconnected. Circuits with three-state output are combinational logic circuits and do not have storage capability. In the positive logic system, a high output is logic 1, and a low output is logic 0. The disconnected or high impedance state is represented by the symbol **Z**.

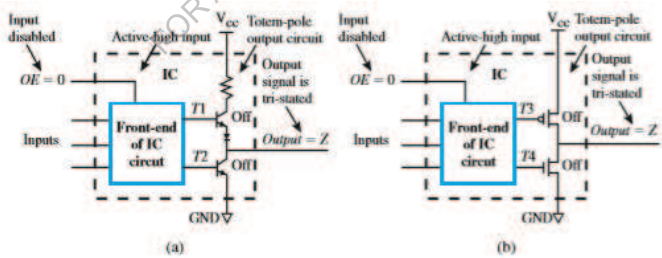


Figure 4.133: Condition for obtaining a tristate output signal with an active-high output enable input

Circuits with this type of control input are designed to act normally—that is, as **2-state output** devices when the control input is active; however, when the control input is **not active** (inactive or disabled), the output (or outputs), becomes disconnected or floating. A device with a 3-state output is created by simply turning off both transistors in the totem-

pole output as shown in Figure 4.133. Both transistors are turned off in a totem-pole output by disabling the **OE** input by pulling it low or to **0**. The term **Z** is borrowed from circuit theory, where it is defined as impedance. When the **Z** value occurs, the output signal is said to be **tri-stated** or in its **high impedance state**. A device with a 3-state output has the output values high, low, and Z.

In Figure 4.133a when **OE** = 0, **T1** = **T2** = 0 and this shuts off both transistors and provides a **Z** output. In Figure 4.133b when **OE** = 0, **T3** = 1 and **T4** = 0 and this shuts off both transistors and provides a **Z** output. In both Figure 4.133a and b when **OE** = 1, the T signals only allow a single transistor to turn on and provide a high or low output. Figure 4.134 shows an example of a **3-state buffer** with an **OE** input and a **3-state output**.

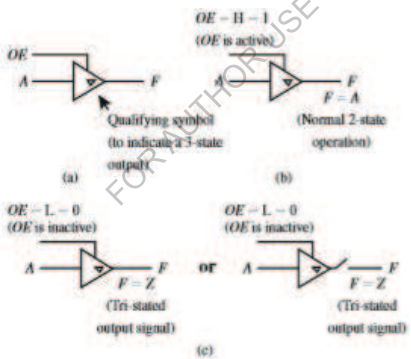


Figure 4.134: 3-state buffer with an OE input: (a) logic symbol; (b) normal 2-state operation; (c) output in disconnected state or open switch representation of tri-stated output signal

Table 4.29 shows a compressed truth table for the 3-state buffer in Figure 4.134.

Table 4.29: compressed truth table for the 3-state buffer

OE	*F*
0	Z
1	*A*

CHAPTER FOUR
VHDL EXPERIEMENTS

VHDL Code & Simulation

CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Tri_State is
    Port ( oe,a : in  STD_LOGIC;
           f : out  STD_LOGIC);
end Tri_State;
architecture Behavioral of Tri_State is
begin
f <= 'Z' when oe ='0' else
      a;
end Behavioral;
```

- Uppercase Z is a value—that is, it is not a signal. Signals can be made upper- or lowercase in VHDL, but values cannot be made lowercase.

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Tri_State_test IS
END Tri_State_test;
ARCHITECTURE behavior OF Tri_State_test IS
 -- Component Declaration for the Unit Under Test (UUT)
 COMPONENT Tri_State
 PORT(
      oe : IN  std_logic;
      a : IN  std_logic;
      f : OUT  std_logic
     );
 END COMPONENT;
--Inputs
signal oe : std_logic := '0';
signal a : std_logic := '0';
--Outputs
signal f : std_logic;
BEGIN
-- Instantiate the Unit Under Test (UUT)
uut: Tri_State PORT MAP (
       oe => oe,
       a => a,
       f => f
     );
-- Stimulus process
stim_proc: process
begin
```

```
    -- hold reset state for 50 ns.
    wait for 50 ns;
    -- insert stimulus here
oe <= '0';
a <= '1';
wait for 50 ns;
oe <= '1';
a <= '1';
wait for 50 ns;
oe <= '1';
a <= '0';
wait for 50 ns;
oe <= '0';
a <= '0';
    wait;
end process;
END;
```

In figure 4.135 that shown the signal and simulation of Tri_State.

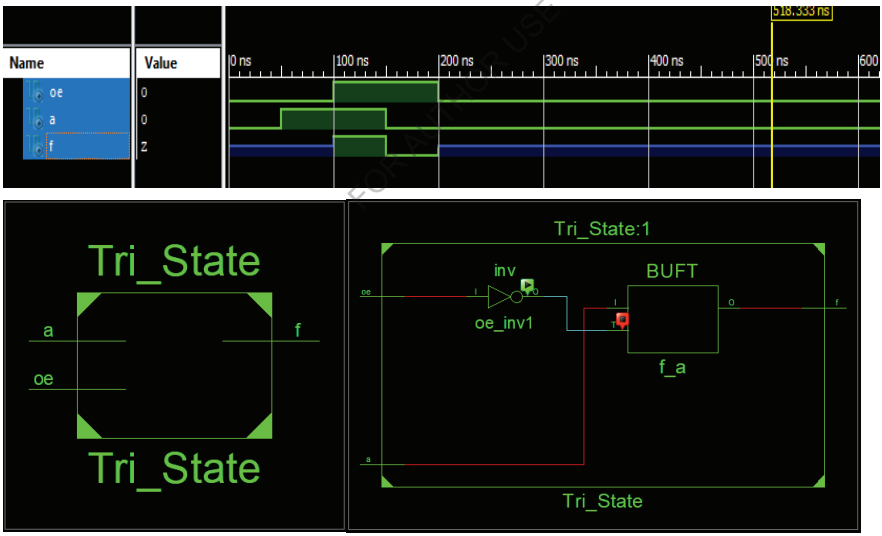


Figure 4.135: Signal and Simulation of Tri_State

CHAPTER FOUR
VHDL EXPERIEMENTS

Discussion

1- Write and simulate the VHDL code for four 3-state buffers with 3-state outputs connected together so the signals A, B, C, and D can share a common data line.

CHAPTER FOUR
VHDL EXPERIEMENTS

Linear-Feedback Shift Register (LFSR)

Aim of Experiment

In this experiment, we will learn the designing Linear-feedback shift register (**LFSR**)in VHDL.

Theory

Linear-Feedback Shift Register (LFSR)

LFSR stands for linear feedback shift register. Although they are widely used in random electronics projects but they are quiet often neglected by the engineer's community. **LFSR** is comprised of a series of **D-flip flops**, depending on the size of the **LFSR**. Some of the states and especially the last one is feedback to the system by going through logical **XOR**. In figure 4.136 that shown the LFSR circuit diagram.

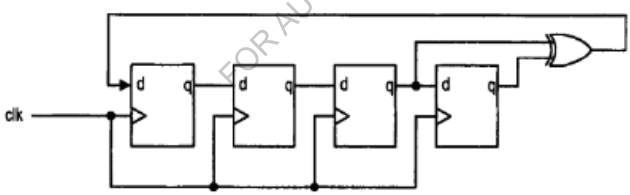


Figure 4.136: LFSR circuit diagram

what's the purpose of LFSR? and how it works?

The soul purpose of lfsr is to generate random numbers/logic. However, the big advantage is next state can be know if the inputs are known.

This control means everything to hardware engineer. By knowing the states lfsr can be utilized to generate test patterns for a given circuit. lfsr can be molded to act as a counter or event generator. Ends of **LFSR** can be brought together to form a cascaded loop.

So, a linear feed-back shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. This is a rotating register, in which one of the Flip-Flops has a XOR as its input, an XOR among two or more outputs of the remaining Flip-Flops. The outputs connected to the XOR Gate are called TAP. There are two TAPs in the below figure 4.137.

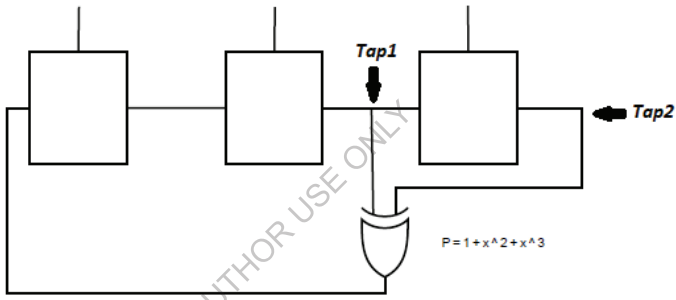


Figure 4.137: Linear feedback register with two taps

The initial value of the LFSR is called "seed", and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits which appears random and which has a very long cycle.

The circuit can be initialized with a different seed from Null vector. In the upper Figure two of the three Flip-Flops are connected to the XOR, which is an input to the other Flip-Flop. Let's suppose that all bits are initialized to '1' after the reset. At each clock cycle the rotation continues and runs a sequence of pseudo random bits on the Flip Flop's

outputs, which will be repeated at a given frequency. In this case the sequence will have a length of 7 as shown in the table 2.30.

Table 4.30: LFSR Output Truth Table

Q1	Q2	Q3
1	1	1
0	1	1
0	0	1
1	0	0
0	1	0
1	0	1
1	1	0
1	1	1

It can be demonstrated that the length of sequence is 2n − 1. The sequence is often associated to a polynomial where the terms different from zero are those with a position corresponding to the TAP. In this case P = 1 + x2 + x3.

Applications of LFSRs include generating pseudo-random numbers, pseudo-noise sequences, fast digital counters, and whitening sequences. Both hardware and software implementations of LFSRs are common. In figure 4.138 LFSR with Polynomial function.

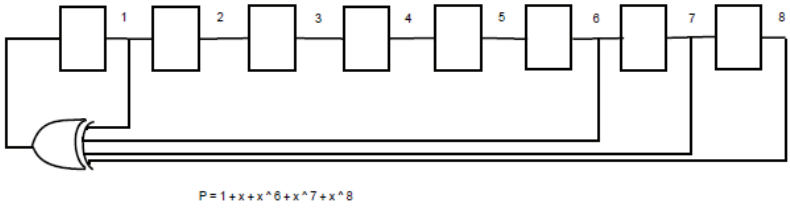


Figure 4.138: LFSR with Polynomial function

The input to lfsr is states coming back as input to XOR. The states which are taped for XOR are 0,2,3 and 4. The lfsr diagram is shown below in figure 4.139.

CHAPTER FOUR
VHDL EXPERIEMENTS

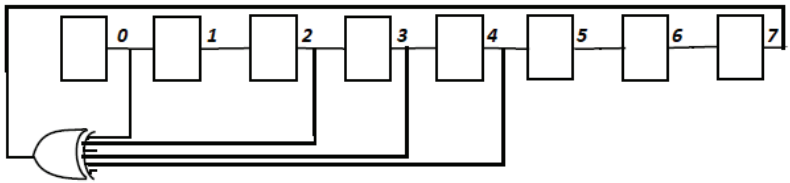


Figure 4.139: 8-bit lfsr in VHDL

VHDL Code & Simulation

CODE (LFSR)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity LFSR8 is
    Port ( Clk,Rst : in  STD_LOGIC;
           output : out  STD_LOGIC_VECTOR (7 DOWNTO 0));
end LFSR8;
architecture LFSR8_beh of LFSR8 is
 SIGNAL Currstate, Nextstate: std_logic_vector (7 DOWNTO 0);
  SIGNAL feedback: std_logic;
begin
 StateReg: PROCESS (Clk,Rst)
  BEGIN
    IF (Rst = '1') THEN
      Currstate <= (0 => '1', OTHERS =>'0');
    ELSIF (Clk = '1' AND Clk'EVENT) THEN
      Currstate <= Nextstate;
    END IF;
  END PROCESS;
  feedback <= Currstate(4) XOR Currstate(3) XOR Currstate(2) XOR Currstate(0);
  Nextstate <= feedback & Currstate(7 DOWNTO 1);
  output <= Currstate;
end LFSR8_beh;
```


First the necessary VHDL libraries are included in the project. After libraries top level lfsr entity is defined in the code. In the code top-level entity name is **LFSR8**. After entity input output ports declaration, it's time to define the internal architecture of the linear feedback register.

In architecture i first defined the two signals (**currstate and nextstate**) of 8-bit length. These signals are playing a vital role in lfsr working. Their purpose is to hold the current

state and next state. In the process part first the reset port is defined. If reset is '**1**' high then current state is initialized "**00000001**". Else if it's a rising edge of clock then next state is assigned to current state. Note that the reset is asynchronous and process is sensitive to input clock and reset.

After the process block the **XOR** between the tap states or bits is done. The result of **XOR** is saved in feedback signal. On the bases of the feedback signal nextstate is calculated. The statement **Nextstate <= feedback & Currstate(7 downto 1)** is concatenating the new lsb (least significant bit) which is feedback with the rest of the number. The & operator in VHDL is used for concatenation purposes. The new state is assigned to the next state. While the current state is assigned to output in the next statement.

Test Bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Testbench IS
END Testbench;
ARCHITECTURE TBarch OF Testbench IS
  COMPONENT LFSR8 IS
    PORT (Clk, Rst: IN std_logic;
          output: OUT std_logic_vector (7 DOWNTO 0));
  END COMPONENT;
  SIGNAL Clk_s, Rst_s: std_logic;
  SIGNAL output_s: std_logic_vector(7 DOWNTO 0);
BEGIN
  CompToTest: LFSR8 PORT MAP (Clk_s, Rst_s, output_s);
  Clk_proc: PROCESS
  BEGIN
    Clk_s <= '1';
    WAIT FOR 10 ns;
    Clk_s <= '0';
    WAIT FOR 10 ns;
  END PROCESS clk_proc;
  Vector_proc: PROCESS
  BEGIN
    Rst_s <= '1';
    WAIT FOR 5 NS;
    Rst_s <= '0';
    FOR index IN 0 To 4 LOOP
      WAIT UNTIL Clk_s='1' AND Clk_s'EVENT;
    END LOOP;
```

```
    WAIT FOR 5 NS;
    ASSERT output_s = X"88" REPORT "Failed output=88";
    WAIT;
  END PROCESS Vector_proc;
END TBarch;
```

In figure 4.140 that shown the signal and simulation of 8 bits LFSR.

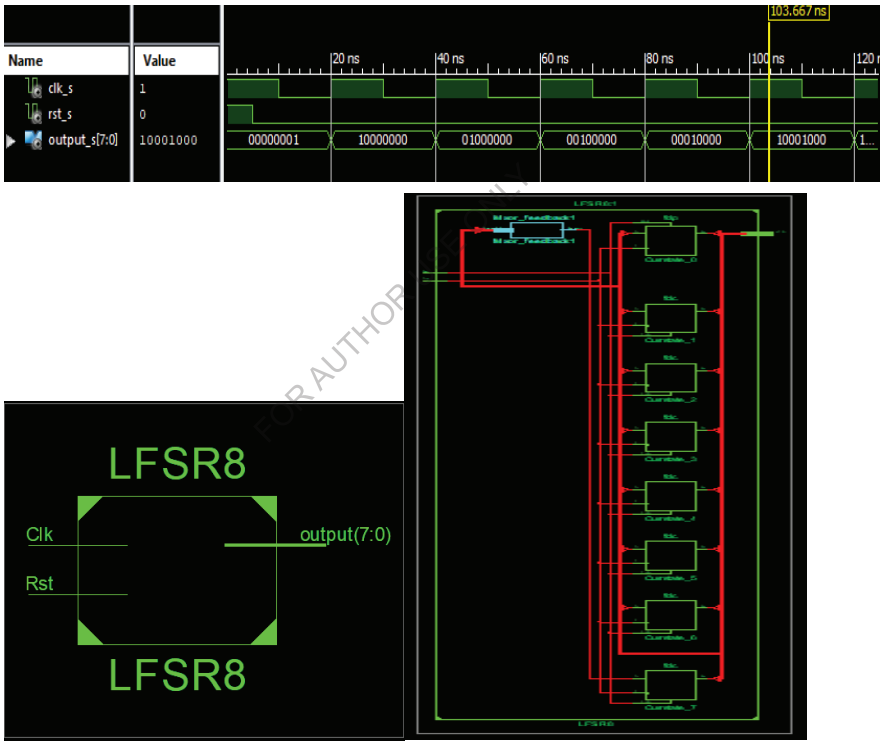


Figure 4.140: Signal and Simulation of 8 Bits LFSR

Discussion

1- Write and simulate the VHDL code for 9 bits LFSR.

CHAPTER FOUR
VHDL EXPERIEMENTS

Reference

1- Navabi, Zainalabedin. VHDL: Analysis and modeling of digital systems. McGraw-Hill, Inc., 1997.

2- Lipsett, Roger, Erich Marschner, and Moe Shahdad. "VHDL-the language." IEEE Design & Test of Computers 3.2 (1986): 28-41.

3- Sjoholm, Stefan, and Lennart Lindh. VHDL for Designers. Vol. 59. Englewood Cliffs: Prentice Hall, 1997.

4- Christen, Ernst, and Kenneth Bakalar. "VHDL-AMS-a hardware description language for analog and mixed-signal applications." IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 46.10 (1999): 1263-1272.

5- Skahill, Kevin. VHDL for programmable logic. Addison-Wesley Longman Publishing Co., Inc., 1996.

6- Chang, Kou-Chuan. Digital design and modeling with VHDL and synthesis. Los Alamitos: IEEE Computer Society Press, 1997.

7- Navabi, Zainalabedin. VHDL: Analysis and modeling of digital systems. McGraw-Hill, Inc., 1997.

8- Pedroni, Volnei A. Circuit design and simulation with VHDL. MIT Press, 2010.

9- Chu, Pong P. FPGA prototyping by VHDL examples: Xilinx Spartan-3 version. John Wiley & Sons, 2011.

10- Olivas, José Á., et al. "Methodology to test and validate a VHDL inference engine through the Xilinx system generator." Soft Computing for Hybrid Intelligent Systems. Springer, Berlin, Heidelberg, 2008. 325-331.