BRAINWARE UNIVERSITY
School of Engineering
Department of Computer Science & Engineering—Cyber Security & Data Science
398, Ramkrishnapur Road, Barasat, North 24 Parganas, Kolkata - 700 125

Department of Computer Science & Engineering—Cyber Security & Data Science

Laboratory Manual

Bachelor of Technology in Computer Science & Engineering—Data Science

BTD39103, Computer Organization & Architecture Lab

&

Bachelor of Technology in Computer Science & Engineering—Cyber Security

BTY39104, Computer Organization & Architecture Lab

Compiled by
Mr. Pallab Paul
Assistant Professor
Dept. of CSE- CS & Ds
Brainware University

Contents

Date	Compiled by	Description
June 2025	Dr. Suparna Panchanan	Updated with 14 points for each experiment
January 2023	Dept. of CSE	Initial release

Table: Revision History

Installation of ModelSim Software

1. Download the software from the official website: https://eda.sw.siemens.com/en-US/ic/modelsim/

2. System requirement

Component	Requirement
CPU	Intel Core i3 or equivalent (x86_64)
RAM	4 GB minimum (8 GB or more recommended)
Disk Space	2–4 GB free disk space
Display	1024x768 resolution (higher recommended)
OS	Windows: 10, 11 (64-bit) Linux: RHEL/CentOS 7/8, Ubuntu 18.04/20.04 (64-bit)

3. Installation Steps

- Download **the .exe installer** for Windows.
- Run the installer with Administrator privileges.

Follow the installation wizard:

- Choose installation directory.
- Accept license agreements.
- Select components (ModelSim, licensing, documentation).
- Finish installation.

Set environment variables **(optional for command-line use):**

- Add <modelsim_install_path>\win64 to your system PATH.
- Launch ModelSim from the Start Menu or desktop.

4. Licensing

- Student Edition**:** Comes with a built-in license.

5. Test the installation:

Run a basic ModelSim simulation to verify that the installation was successful.

Experiment No. 1: To study and verify basic logic gates (AND, OR, NOT) using VHDL.

1. **Aim/Purpose of the Experiment:** To introduce **Hardware Description Languages (VHDL)** and their role in digital system design.
2. **Learning Outcomes:**
   - To familiar with the VHDL interface and code segments.
   - To develop debugging and troubleshooting skills during simulation and code writing.
3. **Prerequisites:** A logic gates are the basic building blocks of any digital system.
4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator
5. **Introduction and Theory:** HDL stands for **Hardware Description Language**. It is a **programming language** that is used to **describe, simulate,** and **create** hardware like digital circuits. HDL is mainly used to discover the faults in the design before implementing it in the hardware. There are many HDLs available in the market, but **VHDL** and **Verilog** are the most popular HDLs.

   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.
6. **Operating Procedure: Launch ISE Project Navigator:** Open the ISE Project Navigator from your desktop or through the start menu.

   Create a New Project:
   - Select File -> New Project.
   - Choose a project name, location, and top-level source type (e.g., Verilog or Schematic).
   - Select the desired device family, device, package, speed grade, and synthesis tool.

   Add Source Files:
   - You can either create a new source file (Project -> New Source) or add existing files (Project -> Add Source). For VHDL, select the VHDL file type.

   Compile the Design:
   - In the Project tab, right-click and select Compile > Compile All.

   Run Simulation:
   - Double-click the desired simulation option (e.g., Simulate Behavioral Model).

   Observe Waveforms:
   - Add the signals you want to observe to the waveform window.
   - Run the simulation and observe the results.

Code for AND Gate	Code for OR Gate	Code for NOT Gate
library IEEE; use IEEE.std_logic_1164.all; entity AND_Gate is	library IEEE; use IEEE.std_logic_1164.all; entity OR_Gate is	library IEEE; use IEEE.std_logic_1164.all; entity NOT_Gate is

port (A : in std_logic; B : in std_logic; Y : out std_logic); end AND_Gate; architecture Dataflow of AND_Gate is begin Y <= A and B; end Dataflow;	port (A : in std_logic; B : in std_logic; Y : out std_logic); end OR_Gate; architecture Dataflow of OR_Gate is begin Y <= A or B; end Dataflow;	port (A : in std_logic; Y : out std_logic); end NOT_Gate; architecture Dataflow of NOT_Gate is begin Y <= not A; end Dataflow;

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

The outputs from the simulation matched the expected behavior of basic gates.
For the AND gate: Output was high (1) only when both inputs were high.
For the OR gate: Output was high if at least one input was high.
For the NOT gate: Output was the logical inverse of the input.

9. Calculations & Analysis:

A	B	AND	OR	NOT A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

The simulated waveform or output log should be compared against this truth table.

10. Result & Interpretation:

The VHDL implementation of basic logic gates (AND, OR, NOT) was successful.
The simulated results verified the truth tables of the respective gates.
No discrepancies were found between theoretical and simulated outputs, validating the design.

11. Follow-up Questions:

What is the difference between std_logic and bit in VHDL?
How would you implement a NAND or NOR gate in VHDL?

12. Extension and Follow-up Activities (if applicable):

Implement the same logic using if-else or when-else constructs in behavioral modeling.
Compare structural and behavioral VHDL models of logic gates for performance and readability.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.

Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 2: Realization of truth table of Universal gates (NAND, NOR)

1. **Aim/Purpose of the Experiment:** To introduce **Hardware Description Languages (VHDL)** and their role in digital system design.
2. **Learning Outcomes:**
   - To familiar with the VHDL interface and code segments.
   - To develop debugging and troubleshooting skills during simulation and code writing.
3. **Prerequisites:** A logic gates are the basic building blocks of any digital system.
4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator
5. **Introduction and Theory:** As discussed in experiment 1.
6. **Operating Procedure:** As discussed in experiment 1.

Code for NAND Gate	Code for NOR Gate
library IEEE; use IEEE.STD_LOGIC_1164.ALL; entity NAND_Gate is Port (A : in STD_LOGIC; B : in STD_LOGIC; Y : out STD_LOGIC); end NAND_Gate; architecture Behavioral of NAND_Gate is begin Y <= A nand B; end Behavioral;	library IEEE; use IEEE.STD_LOGIC_1164.ALL; entity NOR_Gate is Port (A : in STD_LOGIC; B : in STD_LOGIC; Y : out STD_LOGIC); end NOR_Gate; architecture Behavioral of NOR_Gate is begin Y <= A nor B; end Behavioral;

7. **Precautions and/or Troubleshooting:**

   Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).

   Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. **Observations:**

   The output of the **NAND gate** is LOW only when **both inputs are HIGH**.

   The output of the **NOR gate** is HIGH only when **both inputs are LOW**.

9. **Calculations & Analysis:**

A	B	NAND (A NAND B)	NOR (A NOR B)
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

VHDL outputs were verified against this table.

10. Result & Interpretation:

The VHDL implementation of universal logic gates (NAND, NOR) was successful.
The simulated results verified the truth tables of the respective gates.
No discrepancies were found between theoretical and simulated outputs, validating the design.

11. Follow-up Questions:

Why are NAND and NOR gates called universal gates?
What would happen if you used incorrect signal types like `bit` instead of `std_logic`?

12. Extension and Follow-up Activities (if applicable):

Design AND, OR, NOT gates using only NAND gates **in VHDL and verify via simulation.**
Repeat the same using NOR gates**, proving their universality.**

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 3: Realization of truth table of Universal gates (XOR, XNOR).

1. **Aim/Purpose of the Experiment:** To design and simulate the behavior of **XOR** and **XNOR** logic gates using **VHDL**, and to verify their truth tables through simulation.
2. **Learning Outcomes:**
   - Understand the logical behavior of XOR and XNOR gates.
   - Write VHDL code for XOR and XNOR gates using behavioral modeling.
3. **Prerequisites:**
   - Familiarity with VHDL syntax and modeling techniques.
   - Knowledge of simulation tools
4. **Materials/Equipment/Apparatus / Devices/Software required: ModelSim Simulator**
5. **Introduction and Theory:** As discussed in experiment 1.
6. **Operating Procedure:** As discussed in experiment 1.

Code for NAND Gate	Code for NOR Gate

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
 	
entity xor_gate is
  Port (
  A : in STD_LOGIC;
  B : in STD_LOGIC;
  Y : out STD_LOGIC
  );
end xor_gate;
 	
architecture Behavioral of xor_gate is
begin
  Y <= A xor B;
end Behavioral;
	library IEEE;
	use IEEE.STD_LOGIC_1164.ALL;
	
	entity xnor_gate is
	  Port (
	  A : in STD_LOGIC;
	  B : in STD_LOGIC;
	  Y : out STD_LOGIC
	  );
	end xnor_gate;
	
	architecture Behavioral of xnor_gate is
	begin
	  Y <= A xnor B;
	end Behavioral;

7. **Precautions and/or Troubleshooting:**

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. **Observations:**

The **XOR gate** output is high only when the number of high inputs is **odd.**
The **XNOR gate** output is high only when the inputs are **equal**.

9. Calculations & Analysis:

A	B	A XOR B	A XNOR B
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

VHDL outputs were verified against this table.

10. Result & Interpretation:

Each output was observed in waveform simulations.
For both XOR and XNOR, VHDL results matched logical definitions.
No discrepancies were found between theoretical and simulated outputs, validating the design.

11. Follow-up Questions:

How can you implement XOR using only NAND gates?
In what practical applications are XOR and XNOR gates most commonly used?

12. Extension and Follow-up Activities (if applicable):

Design a parity generator and checker using XOR/XNOR gates.
Compare performance of behavioral and structural VHDL models for XOR/XNOR.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 4: Implementation of Adder (Half and Full Adder)

1. **Aim/Purpose of the Experiment:** To design and simulate the behavior of **Half Adder** and **Full Adder** circuits using **VHDL**, and to verify their functionality through waveform simulation.

2. **Learning Outcomes:**
   - Understand the logical functions of Half Adder and Full Adder circuits.

3. **Prerequisites:**
   - Familiarity with VHDL syntax and modeling techniques.
   - Knowledge of simulation tools

4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator

5. **Introduction and Theory:** As discussed in experiment 1.

6. **Operating Procedure:** As discussed in experiment 1.

Code for Half Adder	Code for Full Adder

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
 	
entity half_adder is
  Port (
  A : in STD_LOGIC;
  B : in STD_LOGIC;
  SUM : out STD_LOGIC;
  CARRY : out STD_LOGIC
  );
end half_adder;
 	
architecture dataflow of half_adder is
begin
  SUM <= A xor B;
  CARRY <= A and B;
end dataflow;
	library IEEE;
	use IEEE.STD_LOGIC_1164.ALL;
	
	entity full_adder is
	  Port (
	  A : in STD_LOGIC;
	  B : in STD_LOGIC;
	  Cin : in STD_LOGIC;
	  SUM : out STD_LOGIC;
	  CARRY : out STD_LOGIC
	  );
	end full_adder;
	
	architecture dataflow of full_adder is
	begin
	  SUM <= A xor B xor Cin;
	  CARRY <= (A and B) or (B and Cin) or (A and Cin);
	end dataflow;

7. **Precautions and/or Troubleshooting:**

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. **Observations:**

Sum output is correct for all input combinations using XOR.
Carry output is correct using AND gate logic.

Sum output matches the three-input XOR behavior.

Carry output is accurate based on two intermediate carry calculations.

9. Calculations & Analysis:

Truth Table of Half Adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Truth Table of Full Adder

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

VHDL outputs were verified against this table.

10. Result & Interpretation:

The **Half Adder** and **Full Adder** circuits were **successfully implemented and simulated** using VHDL.
Simulation waveforms and outputs matched theoretical values.
The designs validate the **correct logical operation** of single-bit binary addition.

11. Follow-up Questions:

How is a Full Adder constructed using two Half Adders?
Why do we need a Carry input in a Full Adder but not in a Half Adder?

12. Extension and Follow-up Activities (if applicable):

Design a 4-bit Ripple Carry Adder using Full Adders in structural VHDL.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.

Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 5: Implementation of subtractor (Half and Full Subtractor).

1. **Aim/Purpose of the Experiment:** To design, implement, and simulate a **Half Subtractor** and **Full Subtractor** using **VHDL**, and to verify their truth tables through simulation.

2. **Learning Outcomes:**
   - Understand the working principles and truth tables of half and full subtractors.
   - Write VHDL code to model half and full subtractors using behavioral and structural approaches.

3. **Prerequisites:**
   - Familiarity with VHDL syntax and modeling techniques.
   - Basic understanding of digital logic design and Boolean algebra.
   - Familiarity with the truth tables of half and full subtractors.

4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator

5. **Introduction and Theory:** As discussed in experiment 1.

6. **Operating Procedure:** As discussed in experiment 1.

Code for Half Subtractor	Code for Full Subtractor

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
 	
entity half_subtractor is
  Port (
  A : in STD_LOGIC;
  B : in STD_LOGIC;
  Diff : out STD_LOGIC;
  Borrow : out STD_LOGIC
  );
end half_subtractor;
 	
architecture dataflow of half_subtractor is
begin
  Diff <= A xor B;
  Borrow <= (not A) and B;
end dataflow;
	library IEEE;
	use IEEE.STD_LOGIC_1164.ALL;
	
	entity full_subtractor is
	  Port (
	  A : in STD_LOGIC;
	  B : in STD_LOGIC;
	  Bin : in STD_LOGIC;
	  Diff : out STD_LOGIC;
	  Borrow : out STD_LOGIC
	  );
	end full_subtractor;
	
	architecture dataflow of full_subtractor is
	begin
	  Diff <= A xor B xor Bin;
	  Borrow <= ((not A) and (B or Bin)) or (B and Bin);
	end dataflow;

7. **Precautions and/or Troubleshooting:**

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. **Observations:**

The **Half Subtractor** produced correct outputs for **difference (D)** and **borrow (B)** based on inputs A and B**.**
The **Full Subtractor** correctly handled three inputs (A, B, Bin) and provided appropriate outputs for **difference**

(D)and **borrow-out (Bout)**.

9. Calculations & Analysis:

Truth Table of Half Subtractor

A	B	Difference (D)	Borrow (B)
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Truth Table of Full Subtractor

A	B	Bin	Difference (D)	Borrow (Bout)
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

VHDL outputs were verified against these tables.

10. Result & Interpretation:

The **Half Subtractor** and **Full Subtractor** circuits were successfully implemented in VHDL.
Simulation results confirmed the **correct logical operation** of both subtractors.
No discrepancies were found between theoretical and simulated outputs, validating the design.

11. Follow-up Questions:

What are the key differences between a half subtractor and a full subtractor?
Can subtractors be designed using only NAND gates? How?

12. Extension and Follow-up Activities (if applicable):

Design an 8-bit subtractor using structural VHDL by connecting multiple full subtractors.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.

Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 6: Implementation of 2-bit comparator.

1. **Aim/Purpose of the Experiment:** To design and implement a 2-bit digital comparator circuit using logic gates and verify its functionality through simulation.

2. **Learning Outcomes:**
   - Understand the working principle of digital comparators.
   - Design and implement a 2-bit comparator using VHDL Code.

3. **Prerequisites:**
   - Familiarity with VHDL syntax and modeling techniques.
   - Basic understanding of digital logic design and Boolean algebra.
   - Logic comparators and their truth tables.

4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator

5. **Introduction and Theory:** As discussed in experiment 1.

6. **Operating Procedure:** As discussed in experiment 1.

Code for 2-bit Comparator

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

entity Comparator_2bit is
   Port (
      A : in STD_LOGIC_VECTOR (1 downto 0);
      B : in STD_LOGIC_VECTOR (1 downto 0);
      A_gt_B : out STD_LOGIC;
      A_eq_B : out STD_LOGIC;
      A_lt_B : out STD_LOGIC
   );
end Comparator_2bit;

architecture Behavioral of Comparator_2bit is
begin
   process(A, B)
      variable A_int : unsigned(1 downto 0);
      variable B_int : unsigned(1 downto 0);
   begin
      A_int := unsigned(A);
      B_int := unsigned(B);

      if A_int > B_int then
         A_gt_B <= '1';
         A_eq_B <= '0';
```

```
        A_lt_B <= '0';
    elsif A_int = B_int then
        A_gt_B <= '0';
        A_eq_B <= '1';
        A_lt_B <= '0';
    else
        A_gt_B <= '0';
        A_eq_B <= '0';
        A_lt_B <= '1';
    end if;
  end process;
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

The **Comparator circuit** produces correct outputs for **(A>B)**, **(A<B) and (A=B)** based on inputs A and B**.**
Inputs: Two 2-bit binary numbers `A (1 downto 0)` and `B (1 downto 0)`.
Outputs

- A_gt_B (A > B)
- A_eq_B (A = B)
- A_lt_B (A < B)

Truth Table of 2-bit Comparator

A	B	A_gt_B	A_eq_B	A_lt_B
00	00	0	1	0
01	10	0	0	1
11	10	1	0	0

9. Calculations & Analysis: Logic Expressions:

- A_gt_B = (A1 AND NOT B1) OR (A1 == B1 AND A0 AND NOT B0)
- A_eq_B = (A1 XNOR B1) AND (A0 XNOR B0)
- A_lt_B = (NOT A1 AND B1) OR (A1 == B1 AND NOT A0 AND B0)

These expressions are translated into VHDL code and tested using simulation.

10. Result & Interpretation:

The VHDL code successfully implements a 2-bit comparator. Simulation results confirm the correctness of outputs A_gt_B, A_eq_B, and A_lt_B for all input combinations. Hence, the design is functionally correct.

11. Follow-up Questions:

How does this design scale for larger bit-widths?
Can this comparator be implemented using structural modeling in VHDL?

12. Extension and Follow-up Activities (if applicable):

Combine the comparator with a multiplexer to form a more complex system.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 7: Implementation of Flip Flops (SR FF).

1. **Aim/Purpose of the Experiment:** To design and implement an **SR (Set-Reset) Flip-Flop** using **VHDL** (VHSIC Hardware Description Language), simulate the circuit, and verify its functional behavior.

2. **Learning Outcomes:**
   - Understand the working of SR Flip-Flops and their truth table.
   - Interpret simulation waveforms for flip-flops.
   - Understand the importance of synchronous digital design.

3. **Prerequisites:**
   - Familiarity with VHDL syntax and modeling techniques.
   - Be familiar with simulation tools like **ModelSim**.
   - Have basic knowledge of sequential circuits design.

4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator

5. **Introduction and Theory:** As discussed in experiment 1.

6. **Operating Procedure:** As discussed in experiment 1.

Code for SR FF

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

entity SR_FF is
   Port (
      S    : in  STD_LOGIC;
      R    : in  STD_LOGIC;
      CLK  : in  STD_LOGIC;
      Q    : out STD_LOGIC;
      QBAR : out STD_LOGIC
   );
end SR_FF;

architecture Behavioral of SR_FF is
   signal state : STD_LOGIC := '0';
begin
   process(CLK)
   begin
      if rising_edge(CLK) then
         if (S = '0' and R = '0') then
            -- No change; retain current state
            state <= state;
         elsif (S = '1' and R = '0') then
            -- Set state
            state <= '1';
```

```
        elsif (S = '0' and R = '1') then
          -- Reset state
          state <= '0';
        else
          -- Invalid condition: both S and R are '1'
          state <= 'X';  -- Undefined state
        end if;
      end if;
    end process;

    Q    <= state;
    QBAR <= not state;
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).

Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

- Observe the output **Q** and **Q'** for different combinations of **S (Set)** and **R (Reset)**

S	**R**	**Q (previous)**	**Q (next)**	**Remark**
0	0	0 or 1	No Change	Hold
0	1	X	0	Reset
1	0	X	1	Set
1	1	X	Invalid	Invalid State

9. Calculations & Analysis: Logic Expressions:

- Use timing diagrams from simulation to analyze the behavior
- Compare the expected outputs from the truth table with the simulated outputs.
- Check the outputs at rising or falling edge triggering.

10. Result & Interpretation:

The SR Flip-Flop was successfully implemented in VHDL.

11. Follow-up Questions:

Why is the **S = R = 1** condition considered invalid in SR flip-flops?
What is the difference between a **latch** and a **flip-flop**?

12. Extension and Follow-up Activities (if applicable):

Implement and simulate other flip-flops like JK, D, and T using VHDL.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.

Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 8: Implementation of Flip Flops (D FF).

1. **Aim/Purpose of the Experiment:** To design and implement a **D Flip-Flop (D FF)** using **VHDL (VHSIC Hardware Description Language)** and simulate its behavior to verify its functionality.
2. **Learning Outcomes:**
   - Understand the working of a D Flip-Flop.
   - Interpret simulation results and validate functionality.
3. **Prerequisites:**
   - Familiarity with VHDL syntax and modeling techniques.
   - Understanding of sequential circuits.
4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator
5. **Introduction and Theory:** As discussed in experiment 1.
6. **Operating Procedure:** As discussed in experiment 1.

Code for D FF

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

entity D_FF is
   Port (
      D   : in  STD_LOGIC;
      CLK : in  STD_LOGIC;
      RST : in  STD_LOGIC;
      Q   : out STD_LOGIC;
      Qb  : out STD_LOGIC
   );
end D_FF;

architecture Behavioral of D_FF is
begin
   process(CLK)
   begin
      if rising_edge(CLK) then
         if RST = '1' then
            Q  <= '0';
            Qb <= '1';
         else
            Q  <= D;
            Qb <= not D;
         end if;
      end if;
   end process;
```

```
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

- Output `Q` with respect to input `D` and clock transitions.
- Behavior at rising edge of the clock.

Truth Table of D-FF

Time (ns)	Clock	D Input	Q Output
0	0	0	0
10	↑	1	1
20	↑	0	0

9. Calculations & Analysis:

Timing Analysis: Verify that Q follows D at the rising edge of the clock.
Truth Table Verification: Match simulated outputs with the expected truth table of a D FF.
These expressions are translated into VHDL code and tested using simulation.

10. Result & Interpretation:

The D Flip-Flop has been successfully implemented and simulated.
The simulation waveform confirms that the flip-flop captures the input D at the rising edge of the clock and holds it until the next rising edge.

11. Follow-up Questions:

How can this D FF design be modified to include asynchronous reset?
How would you implement a T Flip-Flop using D Flip-Flops?

12. Extension and Follow-up Activities (if applicable):

Implement an edge-triggered D FF with asynchronous set/reset.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 9: Implementation of Flip Flops (JK FF).

1. **Aim/Purpose of the Experiment:** To design, implement, and simulate a **JK Flip-Flop** using VHDL and understand its working behavior in response to input signals and clock pulses.

2. **Learning Outcomes:**
   - Understand the concept and working of master-slave flip-flops.
   - Simulate and analyze timing behavior using waveform outputs.

3. **Prerequisites:**
   - Have a basic understanding of VHDL syntax and structure
   - Know the operation of SR, JK, and D Flip-Flops.

4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator

5. **Introduction and Theory:** As discussed in experiment 1.

6. **Operating Procedure:** As discussed in experiment 1.

Code for JK FF

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

entity JK_FF is
   Port (
      J    : in  STD_LOGIC;
      K    : in  STD_LOGIC;
      CLK  : in  STD_LOGIC;
      Q    : out STD_LOGIC;
      Qbar  : out STD_LOGIC
   );
end JK_FF;

architecture Behavioral of JK_FF is
   signal state : STD_LOGIC := '0';
begin
   process(CLK)
   begin
      if rising_edge(CLK) then
         case (J & K) is
            when "00" => state <= state;         -- No change
            when "01" => state <= '0';          -- Reset
            when "10" => state <= '1';          -- Set
            when "11" => state <= not state;      -- Toggle
            when others => null;
         end case;
      end if;
```

```
  end process;

  Q   <= state;
  Qbar <= not state;
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

- The state changes of Q and $\bar{Q}$ based on J, K, and Clock inputs.

9. Calculations & Analysis:

Truth Table of J-K FF

J	K	Q(next)	Description
0	0	Q	No Change
0	1	0	Reset
1	0	1	Set
1	1	~Q	Toggle

10. Result & Interpretation:

The VHDL implementation of the JK Flip-Flop was successfully simulated.

11. Follow-up Questions:

How does the JK flip-flop differ from SR and D flip-flops?
Can a JK flip-flop be used to build a T flip-flop? How?

12. Extension and Follow-up Activities (if applicable):

Modify the JK flip-flop to include asynchronous reset and preset inputs.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 10: Implementation of Flip Flops (Master Slave FF).

1. **Aim/Purpose of the Experiment:** To design, implement, and simulate a **Master-Slave Flip-Flop** using VHDL and understand its working behavior in response to input signals and clock pulses.
2. **Learning Outcomes:**
   - Understand the concept and working of master-slave flip-flops.
   - Simulate and analyze timing behavior using waveform outputs.
3. **Prerequisites:**
   - Have a basic understanding of VHDL syntax and structure
   - Know the operation of SR, JK, and D Flip-Flops.
4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator
5. **Introduction and Theory:** As discussed in experiment 1.
6. **Operating Procedure:** As discussed in experiment 1.

Code for Master Slave FF

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

-- Entity declaration for the Master-Slave D Flip-Flop
entity master_slave_dff is
   Port (
      D   : in  STD_LOGIC;
      clk : in  STD_LOGIC;
      Q   : out STD_LOGIC
   );
end master_slave_dff;

architecture Behavioral of master_slave_dff is
   signal master_q : STD_LOGIC;
begin

   -- Process for the Master latch (transparent when clk is high)
   process(clk, D)
   begin
      if clk = '1' then
         master_q <= D;
      end if;
   end process;

   -- Process for the Slave latch (transparent when clk is low)
   process(clk, master_q)
   begin
```

```
    if clk = '0' then
      Q <= master_q;
    end if;
  end process;

end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

- The state changes of Q and $\overline{Q}$ based on J, K, and Clock inputs.
- Output waveform showing master and slave behavior.

9. Calculations & Analysis:

- The VHDL implementation of the Master-Slave Flip-Flop was successfully simulated.
- Compare outputs of Master and Slave latches to confirm sequential action.

10. Result & Interpretation:

The VHDL implementation of the Master-Slave Flip-Flop was successfully simulated.

11. Follow-up Questions:

Why is a Master-Slave configuration used instead of a single latch?
How does clock edge-triggering help in synchronous circuit design?

12. Extension and Follow-up Activities (if applicable):

Use a Master-Slave Flip-Flop to build a frequency divider.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings: A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 11: Implementation of 3-bit UP Counter.

1. **Aim/Purpose of the Experiment:** To design and implement a 3-bit UP counter using VHDL and simulate its behavior to understand sequential circuit design and timing analysis in digital systems.

2. **Learning Outcomes:**

   - Understand the concept of binary counters and their applications.
   - Simulate and verify the functionality of a 3-bit UP counter.

3. **Prerequisites:**

   - Understand the basic operation of flip-flops.
   - Know the truth table and timing diagram of JK Flip-Flop.

4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator

5. **Introduction and Theory:** As discussed in experiment 1.

6. **Operating Procedure:** As discussed in experiment 1.

Code for 3-bit UP counter

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

entity up_counter_3bit is
   Port (
      clk   : in  STD_LOGIC;
      rst   : in  STD_LOGIC;
      count : out STD_LOGIC_VECTOR(2 downto 0)
   );
end up_counter_3bit;

architecture Behavioral of up_counter_3bit is
   signal temp_count : UNSIGNED(2 downto 0) := (others => '0');
begin
   process(clk, rst)
   begin
      if rst = '1' then
         temp_count <= (others => '0');
      elsif rising_edge(clk) then
         temp_count <= temp_count + 1;
      end if;
   end process;

   count <= STD_LOGIC_VECTOR(temp_count);
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

- The counter output increments by 1 on each rising edge of the clock.
- After reaching the maximum value (7 for 3-bit), it wraps around to 0.

9. Calculations & Analysis:

- Number of states = 2^3 = 8 (000 to 111 in binary)
- Bit **Width** = 3
- Wrap-**around behavior**: After 111 (7), the next state is 000 (0).
- Clock **cycle analysis**: Each state change occurs on the rising edge of the clock.
- Reset **behavior** (if implemented): The counter resets to 0 when reset is activated.

10. Result & Interpretation:

The 3-bit UP counter was successfully implemented using VHDL. Simulation results confirmed that the counter increments correctly on each clock cycle and wraps around after reaching the maximum count.

11. Follow-up Questions:

What modifications are needed to convert the UP counter into a DOWN counter?
Explain the differences between synchronous and asynchronous counters.

12. Extension and Follow-up Activities (if applicable):

Design a bidirectional (UP/DOWN) counter using a control signal.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings**:**

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 12: Implementation of 3-bit Down Counter.

1. **Aim/Purpose of the Experiment:** To design and implement a 3-bit down counter using VHDL and simulate its behavior to understand the principles of digital counters.
2. **Learning Outcomes:**
   - Write VHDL code to implement a 3-bit down counter.
   - Analyze timing diagrams and understand counter behavior in digital systems.
3. **Prerequisites:**
   - Understand the basic operation of flip-flops.
   - Know the truth table and timing diagram of JK Flip-Flop.
4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator
5. **Introduction and Theory:** As discussed in experiment 1.
6. **Operating Procedure:** As discussed in experiment 1.

Code for 3-bit Down counter

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

entity down_counter_3bit is
   Port (
      clk   : in  STD_LOGIC;
      reset : in  STD_LOGIC;
      count : out STD_LOGIC_VECTOR (2 downto 0)
   );
end down_counter_3bit;

architecture Behavioral of down_counter_3bit is
   signal temp_count : STD_LOGIC_VECTOR (2 downto 0);
begin
   process(clk, reset)
   begin
      if reset = '1' then
         temp_count <= "111"; -- Initialize to 7
      elsif rising_edge(clk) then
         temp_count <= temp_count - 1;
      end if;
   end process;

   count <= temp_count;
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

- The counter counts down from 7 to 0 and then wraps around to 7.
- Output changes on each negative (or positive) edge of the clock, based on design.

9. Calculations & Analysis:

- **Total States:** $2^3=8$
- **Counting Sequence:** 7 → 6 → 5 → 4 → 3 → 2 → 1 → 0 → 7 (wraps around).
- **Clock Frequency:** Determine based on simulation time steps.
- **Timing Diagram:** Observe and analyze to confirm synchronous behavior.

10. Result & Interpretation:

The 3-bit down counter correctly cycles through 8 states in descending order.
The counter wraps around after reaching 0, confirming circular behavior.

11. Follow-up Questions:

What changes would be needed for a 4-bit down counter?
How does clock skew affect the performance of counters?

12. Extension and Follow-up Activities (if applicable):

Design a 4-bit BCD counter and observe its behavior.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 13: Implementation of 3-bit UP/Down Counter.

1. **Aim/Purpose of the Experiment: To design, implement, and simulate a** 3-bit UP/Down counter **using** VHDL (VHSIC Hardware Description Language) **and verify its operation through simulation tools.**
2. **Learning Outcomes:**
   - Write and simulate VHDL code for digital sequential circuits.
   - Distinguish between UP, DOWN, and UP/DOWN counting logic.
3. **Prerequisites:**
   - Understand the basic operation of combinational circuit and sequential circuit.
   - VHDL syntax and structure.
4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator
5. **Introduction and Theory:** As discussed in experiment 1.
6. **Operating Procedure:** As discussed in experiment 1.

Code for 3-bit Up/Down counter

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

entity up_down_counter is
   Port (
      clk    : in  STD_LOGIC;
      reset  : in  STD_LOGIC;
      up_down : in  STD_LOGIC;  -- '1' for up, '0' for down
      count  : out STD_LOGIC_VECTOR(2 downto 0)
   );
end up_down_counter;

architecture Behavioral of up_down_counter is
   signal count_reg : UNSIGNED(2 downto 0) := (others => '0');
begin
   process(clk)
   begin
      if rising_edge(clk) then
         if reset = '1' then
            count_reg <= (others => '0');
         else
            if up_down = '1' then
               count_reg <= count_reg + 1;
            else
               count_reg <= count_reg - 1;
```

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

  count <= STD_LOGIC_VECTOR(count_reg);
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

- Observe the binary output of the counter for UP and DOWN modes.
- Note the behavior at maximum (111) and minimum (000) values.

9. Calculations & Analysis:

Present State	UP (Next State)	DOWN (Next State)
000	001	111
001	010	000
010	011	001
011	100	010
100	101	011
101	110	100
110	111	101
111	000	110

10. Result & Interpretation:

The 3-bit UP/Down counter successfully counts in both directions as per the `up_down` control signal.
The simulation/waveform confirms the correct sequence of binary values.

11. Follow-up Questions:

What is the maximum count of an N-bit counter?
How can you modify the design to count only even numbers?

12. Extension and Follow-up Activities (if applicable):

Modify the code to support an N-bit (e.g., 4-bit or 8-bit) counter.

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.

Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 14: Implementation of 3-bit Shift register.

1. **Aim/Purpose of the Experiment:** To design, simulate, and implement a 3-bit shift register using VHDL, demonstrating knowledge of sequential circuits and hardware description languages.

2. **Learning Outcomes:**
   - Understand the functionality of shift registers.
   - Simulate and verify the operation of a 3-bit shift register.

3. **Prerequisites:**
   - Understanding of flip-flops and sequential circuits.
   - Familiarity with VHDL syntax and structure.

4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator

5. **Introduction and Theory:** As discussed in experiment 1.

6. **Operating Procedure:** As discussed in experiment 1.

Code for 3-bit Shift Register

```
library ieee;
use ieee.std_logic_1164.all;

entity SISO is
   Port (
      Din   : in  std_logic;
      CLK   : in  std_logic;
      Reset : in  std_logic;
      Dout  : out std_logic
   );
end SISO;

architecture Behavioral of SISO is
   signal t1, t2 : std_logic := '0';
begin
   process (CLK, Reset)
   begin
      if Reset = '1' then
         t1   <= '0';
         t2   <= '0';
         Dout <= '0';
      elsif rising_edge(CLK) then
         Dout <= t2;
         t2   <= t1;
         t1   <= Din;
      end if;
   end process;
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

Clock Cycle	Input (D)	Q(2)	Q(1)	Q(0)	Comment
0	1	0	0	0	Initial state
1	1	0	0	1	1 shifted in
2	0	0	1	0	0 shifted in
3	1	1	0	1	1 shifted in

9. Calculations & Analysis:

Analyze timing diagrams to ensure correct propagation of input bit D through Q (0) to Q (2) over successive clock cycles. Simulation waveforms should match expected register behavior.

10. Result & Interpretation:

The 3-bit shift register was successfully implemented and verified using VHDL. Simulation results confirm correct serial shifting of input data through the register on each clock cycle. The design meets functional requirements.

11. Follow-up Questions:

How does the shift register behave on every clock pulse?
What modifications are needed to implement a left shift register?

12. Extension and Follow-up Activities (if applicable):

Design a **universal shift register** (shift left, right, parallel load, hold).

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Experiment No. 15: Implementation of 4-bit ALU.

1. **Aim/Purpose of the Experiment:** To design and implement a 4-bit Arithmetic Logic Unit (ALU) using VHDL, capable of performing basic arithmetic and logical operations.

2. **Learning Outcomes:**
   - Understand the structure and functionality of an ALU.
   - Implement arithmetic and logical operations using VHDL.

3. **Prerequisites:**
   - Familiarity with VHDL programming language and its syntax.
   - Basic understanding of digital logic design (logic gates, adders, multiplexers).

4. **Materials/Equipment/Apparatus / Devices/Software required:** ModelSim Simulator

5. **Introduction and Theory:** As discussed in experiment 1.

6. **Operating Procedure:** As discussed in experiment 1.

Code for 4-bit ALU

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

entity alu is
   Port (
      inp_a   : in  signed(3 downto 0);
      inp_b   : in  signed(3 downto 0);
      sel     : in  STD_LOGIC_VECTOR(2 downto 0);
      out_alu : out signed(3 downto 0)
   );
end alu;

architecture Behavioral of alu is
begin
   process(inp_a, inp_b, sel)
   begin
      case sel is
         when "000" => out_alu <= inp_a + inp_b;      -- Addition
         when "001" => out_alu <= inp_a - inp_b;      -- Subtraction
         when "010" => out_alu <= inp_a - 1;          -- Decrement
         when "011" => out_alu <= inp_a + 1;          -- Increment
         when "100" => out_alu <= inp_a and inp_b;    -- Bitwise AND
         when "101" => out_alu <= inp_a or inp_b;     -- Bitwise OR
         when "110" => out_alu <= not inp_a;          -- Bitwise NOT
         when "111" => out_alu <= inp_a xor inp_b;    -- Bitwise XOR
         when others => out_alu <= (others => '0');   -- Default case
```

```
    end case;
  end process;
end Behavioral;
```

7. Precautions and/or Troubleshooting:

Ensure all VHDL syntax is correct (use appropriate libraries like IEEE.STD_LOGIC_1164.ALL).
Confirm all signal declarations and assignments are consistent with data types (STD_LOGIC).

8. Observations:

Behavior of the ALU for different operations like addition, subtraction, AND, OR, etc.

9. Calculations & Analysis:

Confirm the result by manually calculating the operations.
Analyze status flags for correctness.

10. Result & Interpretation:

The designed 4-bit ALU successfully performed the following operations as intended:
Arithmetic: ADD, SUB, Mul, Div.
Logical: AND, OR, XOR, NOT

11. Follow-up Questions:

How can status flags (Zero, Carry, Overflow, Negative) be used in processors?
How can the ALU be integrated into a simple processor?

12. Extension and Follow-up Activities (if applicable):

Design a **universal shift register** (shift left, right, parallel load, hold).

13. Assessments:

Completeness, clarity, and accuracy of the submitted report.
Assessment based on correct use of libraries, entity-architecture structure, and signal assignments.

14. Suggested readings:

A VHDL Primer by J Bhasker, **Publisher**: Pearson Education India.

Important Link

https://archive.nptel.ac.in/courses/106/102/106102181/