SRM Institute of Science and Technology

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 1ST Sem

DIGITAL LOGIC FUNDAMENTALS (USA23102J)

Lab Manual

Lab 1: Verification of Basic Gates and Derived Gates

Title: Verification of Basic Gates (AND, OR, NOT) and Derived Gates (NAND, NOR, XOR, XNOR)

Aim: To verify the truth tables of basic logic gates (AND, OR, NOT) and derived logic gates (NAND, NOR, XOR, XNOR) using digital logic trainers or simulation software.

Procedure:

1. Identify the ICs for each logic gate (e.g., 7408 for AND, 7432 for OR, 7404 for NOT, 7400 for NAND, 7402 for NOR, 7486 for XOR).
2. Connect the power supply (VCC and GND) to the respective pins of the ICs.
3. Connect the inputs of each gate to logic switches or input terminals.
4. Connect the output of each gate to an LED or output indicator.
5. Apply all possible input combinations for each gate.
6. Observe and record the output for each input combination.
7. Compare the observed output with the theoretical truth table for each gate.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams if using
simulation software.
// Example (conceptual, not runnable code):
// Module for AND gate
// entity AND_GATE is
//   port (A, B : in BIT; Y : out BIT);
// end AND_GATE;
// architecture BEHAVIORAL of AND_GATE is
// begin
//   Y <= A and B;
// end BEHAVIORAL;
```

Input:

- For a 2-input gate (e.g., AND, OR, NAND, NOR, XOR, XNOR): (0,0), (0,1), (1,0), (1,1)
- For a 1-input gate (e.g., NOT): (0), (1)

Expected Output:

- Truth tables for AND, OR, NOT, NAND, NOR, XOR, XNOR gates.

Lab 2: NAND as Universal Gate NOR as Universal Gate

Title: Implementation of Basic Logic Gates using Universal Gates (NAND and NOR)

Aim: To demonstrate the universality of NAND and NOR gates by implementing basic logic gates (AND, OR, NOT) using only NAND gates and then using only NOR gates.

Procedure:

1. **NAND as Universal Gate:**
   - **NOT gate:** Connect all inputs of a NAND gate together to form a NOT gate. Verify its truth table.
   - **AND gate:** Implement an AND gate using two NAND gates (NAND gate followed by a NOT gate implemented with NAND). Verify its truth table.
   - **OR gate:** Implement an OR gate using three NAND gates (two NOT gates followed by a NAND gate). Verify its truth table.
2. **NOR as Universal Gate:**
   - **NOT gate:** Connect all inputs of a NOR gate together to form a NOT gate. Verify its truth table.
   - **OR gate:** Implement an OR gate using two NOR gates (NOR gate followed by a NOT gate implemented with NOR). Verify its truth table.
   - **AND gate:** Implement an AND gate using three NOR gates (two NOT gates followed by a NOR gate). Verify its truth table.
3. Record all observations and compare with theoretical truth tables.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for
implementations.
// Example (conceptual, not runnable code for NOT using NAND):
// entity NOT_FROM_NAND is
//   port (A : in BIT; Y : out BIT);
// end NOT_FROM_NAND;
// architecture BEHAVIORAL of NOT_FROM_NAND is
// begin
//   Y <= A nand A;
// end BEHAVIORAL;
```

Input:

- Various input combinations (0, 1) for NOT, (0,0), (0,1), (1,0), (1,1) for AND/OR.

Expected Output:

- Truth tables showing correct operation of NOT, AND, OR gates when implemented using only NAND gates.
- Truth tables showing correct operation of NOT, AND, OR gates when implemented using only NOR gates.

Lab 3: Laws of Boolean Expressions

Title: Verification of Basic Laws of Boolean Algebra

Aim: To verify fundamental laws of Boolean Algebra such as Commutative Law, Associative Law, and Distributive Law using logic gates.

Procedure:

1. **Commutative Law (AND):** Implement A·B and B·A using AND gates. Verify that their outputs are identical for all input combinations.
2. **Commutative Law (OR):** Implement A+B and B+A using OR gates. Verify that their outputs are identical for all input combinations.
3. **Associative Law (AND):** Implement (A·B)·C and A·(B·C) using AND gates. Verify their equivalence.
4. **Associative Law (OR):** Implement (A+B)+C and A+(B+C) using OR gates. Verify their equivalence.
5. Record all observations.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for the
implementations.
// Example (conceptual, not runnable code for Commutative Law of AND):
// entity COMMUTATIVE_AND is
//   port (A, B : in BIT; Y1, Y2 : out BIT);
// end COMMUTATIVE_AND;
// architecture BEHAVIORAL of COMMUTATIVE_AND is
// begin
//   Y1 <= A and B;
//   Y2 <= B and A;
// end BEHAVIORAL;
```

Input:

- For two variables: (0,0), (0,1), (1,0), (1,1)
- For three variables: (0,0,0) to (1,1,1)

Expected Output:

- Truth tables demonstrating the equivalence of both sides of the Commutative and Associative laws.

Lab 4: Verifications of Distributive Law

Title: Verification of Distributive Law of Boolean Algebra

Aim: To verify the Distributive Law of Boolean Algebra, $A\cdot(B+C)=(A\cdot B)+(A\cdot C)$ and $A+(B\cdot C)=(A+B)\cdot(A+C)$, using logic gates.

Procedure:

1. **Distributive Law (AND over OR):**
   - Implement the LHS: $A\cdot(B+C)$ using one OR gate and one AND gate.
   - Implement the RHS: $(A\cdot B)+(A\cdot C)$ using two AND gates and one OR gate.
   - Apply all possible input combinations for A, B, and C.
   - Observe and compare the outputs of both sides.
2. **Distributive Law (OR over AND):**
   - Implement the LHS: $A+(B\cdot C)$ using one AND gate and one OR gate.
   - Implement the RHS: $(A+B)\cdot(A+C)$ using two OR gates and one AND gate.
   - Apply all possible input combinations for A, B, and C.
   - Observe and compare the outputs of both sides.
3. Record all observations.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for the
implementations.
// Example (conceptual, not runnable code for A.(B+C)):
// entity DISTRIBUTIVE_AND_OR_LHS is
//   port (A, B, C : in BIT; Y : out BIT);
// end DISTRIBUTIVE_AND_OR_LHS;
// architecture BEHAVIORAL of DISTRIBUTIVE_AND_OR_LHS is
//   signal temp_BC : BIT;
// begin
//   temp_BC <= B or C;
//   Y <= A and temp_BC;
// end BEHAVIORAL;
```

Input:

- All 8 possible input combinations for A, B, C (000 to 111).

Expected Output:

- Truth tables demonstrating the equivalence of both sides of the Distributive Law for both forms.

Lab 5: Simplifying Boolean Expressions using theorems

Title: Simplification of Boolean Expressions using Boolean Algebra Theorems

Aim: To simplify given Boolean expressions using Boolean algebra theorems and verify the simplified expression's equivalence with the original expression using logic gates or simulation.

Procedure:

1. Choose a complex Boolean expression (e.g., F=AB+ABC+ABC).
2. Simplify the expression algebraically using Boolean theorems (e.g., consensus theorem, absorption law, De Morgan's theorems, etc.).
3. Implement the original (unsimplified) expression using logic gates.
4. Implement the simplified expression using logic gates.
5. Apply all possible input combinations to both circuits.
6. Observe and compare the outputs of both circuits for each input combination.
7. Record the simplification steps and the observed truth tables.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for both
original and simplified expressions.
// Example (conceptual, not runnable code for an expression):
// entity ORIGINAL_EXPRESSION is
//   port (A, B, C : in BIT; Y : out BIT);
// end ORIGINAL_EXPRESSION;
// architecture BEHAVIORAL of ORIGINAL_EXPRESSION is
// begin
//   Y <= (not A and B) or (A and B and C) or (A and not B and C);
// end BEHAVIORAL;

// entity SIMPLIFIED_EXPRESSION is
//   port (A, B, C : in BIT; Y : out BIT);
// end SIMPLIFIED_EXPRESSION;
// architecture BEHAVIORAL of SIMPLIFIED_EXPRESSION is
// begin
//   // Assuming simplification leads to a simpler form, e.g., Y <= B + A and
C;
//   Y <= B or (A and C);
// end BEHAVIORAL;
```

Input:

- All possible input combinations for the variables in the expression.

Expected Output:

- The simplified Boolean expression.
- Truth tables showing that the original and simplified circuits produce identical outputs.

Lab 6: Implementation of Binary Addition and Subtraction

Title: Implementation of Binary Addition and Subtraction

Aim: To implement and verify the operation of binary addition and subtraction circuits.

Procedure:

1. **Binary Addition:**
   - Design a circuit for 2-bit or 4-bit parallel binary addition using full adders.
   - Connect the inputs (two binary numbers) and observe the sum and carry outputs.
   - Test with various binary numbers.
2. **Binary Subtraction (using 2's complement):**
   - Design a circuit for 2-bit or 4-bit binary subtraction using full adders and inverters (for 2's complement).
   - Convert the subtrahend to its 2's complement.
   - Add the minuend to the 2's complement of the subtrahend.
   - Observe the difference output.
   - Test with various binary numbers.
3. Record all observations.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for
binary adder/subtractor.
// Example (conceptual, not runnable code for a 4-bit adder):
// entity FOUR_BIT_ADDER is
//   port (A, B : in STD_LOGIC_VECTOR(3 downto 0);
//         Cin : in STD_LOGIC;
//         Sum : out STD_LOGIC_VECTOR(3 downto 0);
//         Cout : out STD_LOGIC);
// end FOUR_BIT_ADDER;
// architecture BEHAVIORAL of FOUR_BIT_ADDER is
// begin
//   -- Logic for 4-bit addition using multiple full adders
// end BEHAVIORAL;
```

Input:

- Pairs of binary numbers (e.g., for 2-bit: 00+00, 01+10, 11+01, etc.)

Expected Output:

- Correct sum and carry for addition.
- Correct difference for subtraction.

Lab 7: Half Adder and Full Adder

Title: Implementation and Verification of Half Adder and Full Adder Circuits

Aim: To design, implement, and verify the truth tables and functionalities of a Half Adder and a Full Adder.

Procedure:

1. **Half Adder:**
   - Design a Half Adder circuit using basic logic gates (XOR for Sum, AND for Carry).
   - Implement the circuit on a breadboard or in simulation.
   - Apply all possible input combinations (00, 01, 10, 11) for the two input bits.
   - Observe and record the Sum and Carry outputs.
2. **Full Adder:**
   - Design a Full Adder circuit using basic logic gates or by cascading two Half Adders and an OR gate.
   - Implement the circuit.
   - Apply all possible input combinations (000 to 111) for the three input bits (A, B, Carry-in).
   - Observe and record the Sum and Carry-out outputs.
3. Compare observed results with theoretical truth tables.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for Half
and Full Adders.
// Example (conceptual, not runnable code for Half Adder):
// entity HALF_ADDER is
//   port (A, B : in BIT; Sum, Carry : out BIT);
// end HALF_ADDER;
// architecture BEHAVIORAL of HALF_ADDER is
// begin
//   Sum <= A xor B;
//   Carry <= A and B;
// end BEHAVIORAL;
```

Input:

- Half Adder: (0,0), (0,1), (1,0), (1,1)
- Full Adder: (0,0,0) to (1,1,1)

Expected Output:

- Truth table for Half Adder (Sum, Carry).
- Truth table for Full Adder (Sum, Carry-out).

Lab 8: Half Subtractor and Full Subtractor

Title: Implementation and Verification of Half Subtractor and Full Subtractor Circuits

Aim: To design, implement, and verify the truth tables and functionalities of a Half Subtractor and a Full Subtractor.

Procedure:

1. **Half Subtractor:**
   - Design a Half Subtractor circuit using basic logic gates (XOR for Difference, AND/NOT for Borrow-out).
   - Implement the circuit.
   - Apply all possible input combinations (00, 01, 10, 11) for the two input bits.
   - Observe and record the Difference and Borrow-out outputs.
2. **Full Subtractor:**
   - Design a Full Subtractor circuit using basic logic gates or by cascading two Half Subtractors and an OR gate.
   - Implement the circuit.
   - Apply all possible input combinations (000 to 111) for the three input bits (Minuend, Subtrahend, Borrow-in).
   - Observe and record the Difference and Borrow-out outputs.
3. Compare observed results with theoretical truth tables.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for Half
and Full Subtractors.
// Example (conceptual, not runnable code for Half Subtractor):
// entity HALF_SUBTRACTOR is
//   port (A, B : in BIT; Diff, Borrow : out BIT);
// end HALF_SUBTRACTOR;
// architecture BEHAVIORAL of HALF_SUBTRACTOR is
// begin
//   Diff <= A xor B;
//   Borrow <= (not A) and B;
// end BEHAVIORAL;
```

Input:

- Half Subtractor: (0,0), (0,1), (1,0), (1,1)
- Full Subtractor: (0,0,0) to (1,1,1)

Expected Output:

- Truth table for Half Subtractor (Difference, Borrow-out).
- Truth table for Full Subtractor (Difference, Borrow-out).

Lab 9: Implementation of multiplexer

Title: Implementation and Verification of a Multiplexer (MUX)

Aim: To design, implement, and verify the operation of a multiplexer (data selector) using logic gates.

Procedure:

1. **2-to-1 MUX:**
   - Design a 2-to-1 MUX using AND, OR, and NOT gates.
   - Implement the circuit.
   - Apply various data inputs and select line combinations.
   - Observe that the output corresponds to the selected input.
2. **4-to-1 MUX:**
   - Design a 4-to-1 MUX using AND, OR, and NOT gates, or by cascading 2-to-1 MUXes.
   - Implement the circuit.
   - Apply various data inputs (D0,D1,D2,D3) and select line combinations (S1,S0).
   - Observe that the output corresponds to the selected input.
3. Verify the truth table of the multiplexer.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for the
multiplexer.
// Example (conceptual, not runnable code for 2-to-1 MUX):
// entity MUX_2_TO_1 is
//   port (D0, D1, S : in BIT; Y : out BIT);
// end MUX_2_TO_1;
// architecture BEHAVIORAL of MUX_2_TO_1 is
// begin
//   Y <= (not S and D0) or (S and D1);
// end BEHAVIORAL;
```

Input:

- For 2-to-1 MUX: (D0, D1, S) combinations (e.g., 0,1,0; 1,0,1)
- For 4-to-1 MUX: (D0, D1, D2, D3, S1, S0) combinations

Expected Output:

- Truth table showing that the output follows the selected input.

Lab 10: Implementation of DeMultiplexer

Title: Implementation and Verification of a Demultiplexer (DeMUX)

Aim: To design, implement, and verify the operation of a demultiplexer (data distributor) using logic gates.

Procedure:

1. **1-to-2 DeMUX:**
    - Design a 1-to-2 DeMUX using AND and NOT gates.
    - Implement the circuit.
    - Apply data input and select line combinations.
    - Observe that the input data is routed to the selected output line.
2. **1-to-4 DeMUX:**
    - Design a 1-to-4 DeMUX using AND and NOT gates, or by cascading 1-to-2 DeMUXes.
    - Implement the circuit.
    - Apply data input (Din) and select line combinations (S1,S0).
    - Observe that the input data is routed to the selected output line (Y0,Y1,Y2,Y3).
3. Verify the truth table of the demultiplexer.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for the
demultiplexer.
// Example (conceptual, not runnable code for 1-to-2 DeMUX):
// entity DEMUX_1_TO_2 is
//   port (Din, S : in BIT; Y0, Y1 : out BIT);
// end DEMUX_1_TO_2;
// architecture BEHAVIORAL of DEMUX_1_TO_2 is
// begin
//   Y0 <= Din and (not S);
//   Y1 <= Din and S;
// end BEHAVIORAL;
```

Input:

- For 1-to-2 DeMUX: (Din, S) combinations (e.g., 1,0; 1,1)
- For 1-to-4 DeMUX: (Din, S1, S0) combinations

Expected Output:

- Truth table showing that the input data appears only on the selected output line, and other output lines are low.

Lab 11: Implementation of Shift Registers and Serial

Title: Implementation and Verification of Shift Registers (SISO, SIPO, PISO, PIPO)

Aim: To implement and verify the operation of various types of shift registers: Serial-In Serial-Out (SISO), Serial-In Parallel-Out (SIPO), Parallel-In Serial-Out (PISO), and Parallel-In Parallel-Out (PIPO).

Procedure:

1. **SISO Shift Register:**
   - Design a 4-bit SISO shift register using D flip-flops.
   - Apply a serial input data stream and clock pulses.
   - Observe the serial output after a certain number of clock cycles.
2. **SIPO Shift Register:**
   - Design a 4-bit SIPO shift register using D flip-flops.
   - Apply a serial input data stream and clock pulses.
   - Observe the parallel output after all bits have been shifted in.
3. **PISO Shift Register:**
   - Design a 4-bit PISO shift register.
   - Load parallel data into the register.
   - Apply clock pulses and observe the serial output.
4. **PIPO Shift Register:**
   - Design a 4-bit PIPO shift register.
   - Load parallel data into the register.
   - Observe the parallel output directly.
5. Record the data shifting process for each type.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for shift
registers.
// Example (conceptual, not runnable code for a 4-bit SISO register):
// entity SISO_REGISTER is
//   port (Clk, Serial_In : in BIT; Serial_Out : out BIT);
// end SISO_REGISTER;
// architecture BEHAVIORAL of SISO_REGISTER is
//   signal Q : BIT_VECTOR(3 downto 0);
// begin
//   process(Clk)
//   begin
//     if rising_edge(Clk) then
//       Q(0) <= Serial_In;
//       Q(1) <= Q(0);
//       Q(2) <= Q(1);
//       Q(3) <= Q(2);
//     end if;
//   end process;
//   Serial_Out <= Q(3);
// end BEHAVIORAL;
```

Input:

- Serial data streams (e.g., 1011, 0101)
- Parallel data (e.g., 1011)
- Clock pulses

Expected Output:

- Observation of data shifting from input to output based on the type of shift register.

Lab 12: Four Bit Binary Shift Counters

Title: Implementation and Verification of Four-Bit Binary Shift Counters

Aim: To design, implement, and verify the operation of a 4-bit binary shift counter (e.g., a Johnson counter or a ring counter implemented with shift register principles).

Procedure:

1. **Johnson Counter (Twisted Ring Counter):**
   - Design a 4-bit Johnson counter using D flip-flops where the inverted output of the last flip-flop is fed back to the input of the first flip-flop.
   - Initialize the counter to a specific state (e.g., 0000).
   - Apply clock pulses and observe the sequence of states at the outputs of the flip-flops.
2. Record the state sequence and verify it matches the theoretical sequence for a Johnson counter.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for the
shift counter.
// Example (conceptual, not runnable code for a 4-bit Johnson Counter):
// entity JOHNSON_COUNTER is
//   port (Clk, Reset : in BIT; Q : out BIT_VECTOR(3 downto 0));
// end JOHNSON_COUNTER;
// architecture BEHAVIORAL of JOHNSON_COUNTER is
//   signal current_Q : BIT_VECTOR(3 downto 0);
// begin
//   process(Clk, Reset)
//   begin
//     if Reset = '1' then
//       current_Q <= "0000";
//     elsif rising_edge(Clk) then
//       current_Q(0) <= not current_Q(3); -- Twisted feedback
//       current_Q(1) <= current_Q(0);
//       current_Q(2) <= current_Q(1);
//       current_Q(3) <= current_Q(2);
//     end if;
//   end process;
//   Q <= current_Q;
// end BEHAVIORAL;
```

Input:

- Clock pulses
- Reset signal (if applicable)

Expected Output:

- The unique 8-state sequence (0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001) for a 4-bit Johnson counter.

Lab 13: Ring Counters

Title: Implementation and Verification of Ring Counters

Aim: To design, implement, and verify the operation of a ring counter.

Procedure:

1. Design a 4-bit ring counter using D flip-flops where the output of the last flip-flop is fed back to the input of the first flip-flop (without inversion).
2. Initialize the counter with a single '1' (e.g., 1000).
3. Apply clock pulses and observe the sequence of states at the outputs of the flip-flops.
4. Record the state sequence and verify it matches the theoretical sequence for a ring counter.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for the
ring counter.
// Example (conceptual, not runnable code for a 4-bit Ring Counter):
// entity RING_COUNTER is
//   port (Clk, Reset : in BIT; Q : out BIT_VECTOR(3 downto 0));
// end RING_COUNTER;
// architecture BEHAVIORAL of RING_COUNTER is
//   signal current_Q : BIT_VECTOR(3 downto 0);
// begin
//   process(Clk, Reset)
//   begin
//     if Reset = '1' then
//       current_Q <= "1000"; -- Initial state for ring counter
//     elsif rising_edge(Clk) then
//       current_Q(0) <= current_Q(3); -- Direct feedback
//       current_Q(1) <= current_Q(0);
//       current_Q(2) <= current_Q(1);
//       current_Q(3) <= current_Q(2);
//     end if;
//   end process;
//   Q <= current_Q;
// end BEHAVIORAL;
```

Input:

- Clock pulses
- Reset signal (to set initial state)

Expected Output:

- The 4-state sequence (1000, 0100, 0010, 0001) for a 4-bit ring counter.

Lab 14: Implementation of DOWN Counter

Title: Implementation and Verification of a DOWN Counter

Aim: To design, implement, and verify the operation of a synchronous or asynchronous DOWN counter.

Procedure:

1. **Asynchronous 3-bit DOWN Counter:**
   - Design a 3-bit asynchronous (ripple) DOWN counter using JK or T flip-flops.
   - Connect the output of each flip-flop to the clock input of the next, with appropriate connections to achieve counting down.
   - Initialize the counter to its maximum state (e.g., 111).
   - Apply clock pulses and observe the sequence of states (111, 110, 101, ..., 000).
2. **Synchronous 3-bit DOWN Counter:**
   - Design a 3-bit synchronous DOWN counter using JK or T flip-flops, where all flip-flops are clocked simultaneously.
   - Determine the input logic for each flip-flop to achieve counting down.
   - Initialize the counter to its maximum state.
   - Apply clock pulses and observe the sequence of states.
3. Record the state sequence and verify it matches the theoretical sequence.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for the
DOWN counter.
// Example (conceptual, not runnable code for a 3-bit asynchronous DOWN
counter):
// entity ASYNC_DOWN_COUNTER is
//   port (Clk, Reset : in BIT; Q : out BIT_VECTOR(2 downto 0));
// end ASYNC_DOWN_COUNTER;
// architecture BEHAVIORAL of ASYNC_DOWN_COUNTER is
//   signal Q_internal : BIT_VECTOR(2 downto 0);
// begin
//   process(Clk, Reset) -- First flip-flop
//   begin
//     if Reset = '1' then
//       Q_internal(0) <= '1';
//     elsif falling_edge(Clk) then -- For down counter, often falling edge
//       Q_internal(0) <= not Q_internal(0);
//     end if;
//   end process;

//   process(Q_internal(0)) -- Second flip-flop clocked by Q0
//   begin
//     if Reset = '1' then
//       Q_internal(1) <= '1';
//     elsif falling_edge(Q_internal(0)) then
//       Q_internal(1) <= not Q_internal(1);
//     end if;
//   end process;

//   process(Q_internal(1)) -- Third flip-flop clocked by Q1
//   begin
//     if Reset = '1' then
//       Q_internal(2) <= '1';
//     elsif falling_edge(Q_internal(1)) then
//       Q_internal(2) <= not Q_internal(2);
//     end if;
//   end process;

//   Q <= Q_internal;
// end BEHAVIORAL;
```

Input:

- Clock pulses
- Reset signal (to set initial state)

Expected Output:

- The counter sequence decrementing from max value to 000.

Lab 15: Implementation of DOWN Counter

Title: Implementation and Verification of a DOWN Counter (Repeat)

Aim: To design, implement, and verify the operation of a synchronous or asynchronous DOWN counter.

Procedure:

1. **Asynchronous 3-bit DOWN Counter:**
   - Design a 3-bit asynchronous (ripple) DOWN counter using JK or T flip-flops.
   - Connect the output of each flip-flop to the clock input of the next, with appropriate connections to achieve counting down.
   - Initialize the counter to its maximum state (e.g., 111).
   - Apply clock pulses and observe the sequence of states (111, 110, 101, ..., 000).
2. **Synchronous 3-bit DOWN Counter:**
   - Design a 3-bit synchronous DOWN counter using JK or T flip-flops, where all flip-flops are clocked simultaneously.
   - Determine the input logic for each flip-flop to achieve counting down.
   - Initialize the counter to its maximum state.
   - Apply clock pulses and observe the sequence of states.
3. Record the state sequence and verify it matches the theoretical sequence.

Source Code:

```
// This section would contain VHDL/Verilog code or circuit diagrams for the
DOWN counter.
// Example (conceptual, not runnable code for a 3-bit asynchronous DOWN
counter):
// entity ASYNC_DOWN_COUNTER_2 is
//   port (Clk, Reset : in BIT; Q : out BIT_VECTOR(2 downto 0));
// end ASYNC_DOWN_COUNTER_2;
// architecture BEHAVIORAL of ASYNC_DOWN_COUNTER_2 is
//   signal Q_internal : BIT_VECTOR(2 downto 0);
// begin
//   process(Clk, Reset) -- First flip-flop
//   begin
//     if Reset = '1' then
//       Q_internal(0) <= '1';
//     elsif falling_edge(Clk) then -- For down counter, often falling edge
//       Q_internal(0) <= not Q_internal(0);
//     end if;
//   end process;

//   process(Q_internal(0)) -- Second flip-flop clocked by Q0
//   begin
//     if Reset = '1' then
//       Q_internal(1) <= '1';
//     elsif falling_edge(Q_internal(0)) then
//       Q_internal(1) <= not Q_internal(1);
//     end if;
//   end process;

//   process(Q_internal(1)) -- Third flip-flop clocked by Q1
//   begin
//     if Reset = '1' then
//       Q_internal(2) <= '1';
//     elsif falling_edge(Q_internal(1)) then
//       Q_internal(2) <= not Q_internal(2);
//     end if;
//   end process;

//   Q <= Q_internal;
// end BEHAVIORAL;
```

Input:

- Clock pulses
- Reset signal (to set initial state)

Expected Output:

- The counter sequence decrementing from max value to 000.