ELECTRONICS – PROJECT

Verilog Programming as an HDL by using Vivado Software for Electronics Application

Name of Student	Atharva Manoj Lotankar
Class	D10 – C
Branch	Information Technology
Roll Number	39
College	Vivekanand Education Society's Institute of Technology
Sign of Student	[illegible]

Contents

Verilog Programming Projects – Electronics

Atharva Lotankar (D10C-39)

Project 1: Implementation of Logic Gates

Aim: To demonstrate all Logic Gates functions – Basic Gates like AND, OR, NOT and Derived Gates like NAND, NOR and XOR.

Theory:

The Logic gates are the basic circuits of digital electronic systems like resistors, capacitors in analog circuits. They can be constructed by using simple switches, relays, vaccum tubes, transistors or diodes. Now due to their availability in the form of integrated circuits (ICs) they are used to construct digital circuits.

Logic: It is the statement, which gives condition for high output. Logic may be expressed in terms of High/Low, True/False, 1/0, etc.

Truth Table: It is a table in which logic is expressed by taking all possible combinations of inputs and respective outputs. The number of possible combinations in a truth table is given by 2^n.

Logic Gate Boolean Algebra

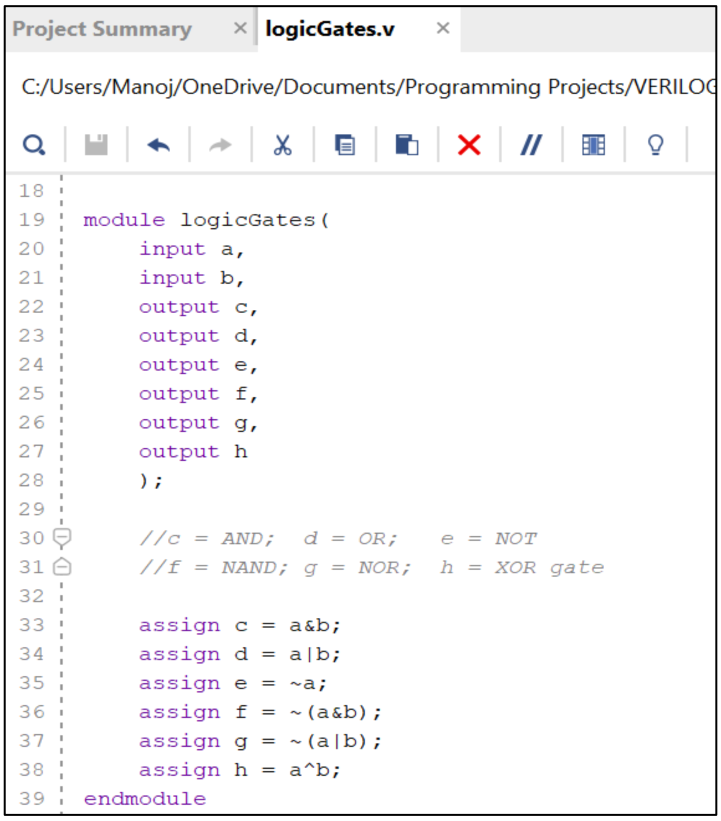
Basic Gates:

- AND Gate: Output is 1 only when all inputs are 1.
  - Boolean Expression: $Y = A \cdot B$
- OR Gate: Output is 1 when at least one input is 1.
  - Boolean Expression: $Y = A + B$
- NOT Gate: Inverts the input.
  - Boolean Expression: $Y = A'$

Derived Gates:

- NAND Gate: Output is 0 only when all inputs are 1 (inverse of AND).
  - Boolean Expression: $Y = (A \cdot B)'$
- NOR Gate: Output is 1 only when all inputs are 0 (inverse of OR).
  - Boolean Expression: $Y = (A + B)'$
- XOR Gate: Output is 1 only when the inputs are different.
  - Boolean Expression: $Y = A \oplus B = (A' \cdot B) + (A \cdot B')$

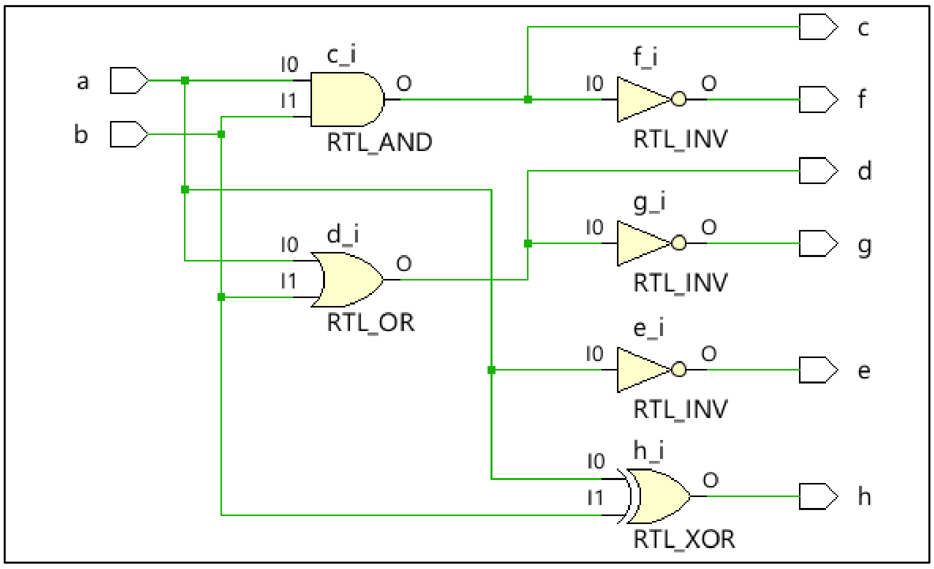
RTL Code:



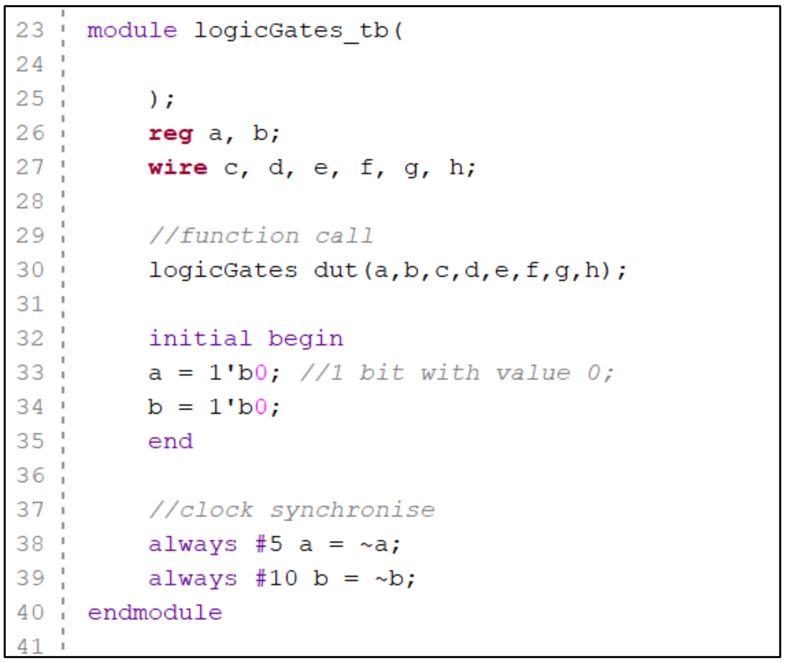
C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG

```verilog

module logicGates(
    input a,
    input b,
    output c,
    output d,
    output e,
    output f,
    output g,
    output h
    );

    //c = AND;  d = OR;   e = NOT
    //f = NAND; g = NOR;  h = XOR gate

    assign c = a&b;
    assign d = a|b;
    assign e = ~a;
    assign f = ~(a&b);
    assign g = ~(a|b);
    assign h = a^b;
endmodule
```

Schematic diagram:

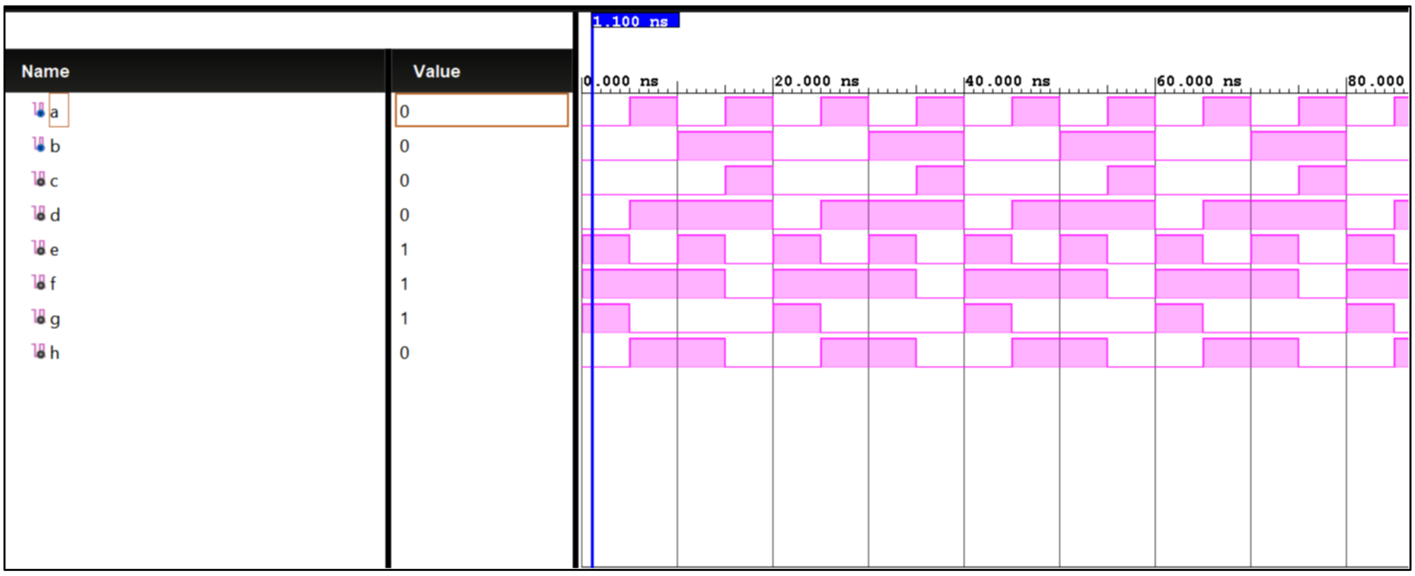


Testbench:



```
module logicGates_tb(

    );
    reg a, b;
    wire c, d, e, f, g, h;

    //function call
    logicGates dut(a,b,c,d,e,f,g,h);

    initial begin
    a = 1'b0; //1 bit with value 0;
    b = 1'b0;
    end

    //clock synchronise
    always #5 a = ~a;
    always #10 b = ~b;
endmodule
```

Wave Form:



Name	Value
a	0
b	0
c	0
d	0
e	1
f	1
g	1
h	0

Conclusion:

In conclusion, logic gates serve as the fundamental building blocks of digital electronics, enabling the processing and manipulation of binary information. Their ability to perform Boolean operations on input signals lays the groundwork for constructing complex digital circuits, including those found in computers, microprocessors, and other electronic devices. By understanding and effectively utilizing logic gates, engineers and designers can create innovative solutions that drive technological advancement and shape the digital landscape.

Project 2: Implementation of Four Bit Adder

Aim: To demonstrate the four bit adder in Verilog Programming.

Theory:

A 4-bit adder is a fundamental digital circuit capable of adding two 4-bit binary numbers. It's a crucial component in various digital systems, from simple calculators to complex microprocessors.

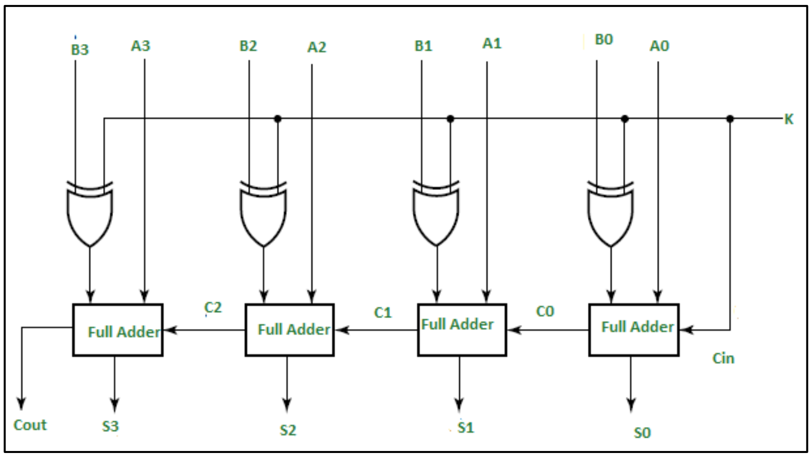
The 4-bit adder is constructed using multiple full adders, each responsible for adding two bits and a carry-in bit. The carry-out from one full adder is fed as the carry-in to the next, allowing for cascading addition of larger numbers.

***Key Components*:**

1. Full Adder:
   - Takes three inputs: two bits to be added (A and B) and a carry-in bit (Cin).
   - Produces two outputs: a sum bit (S) and a carry-out bit (Cout).
   - The truth table for a full adder is:

A	B	Cin	S	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

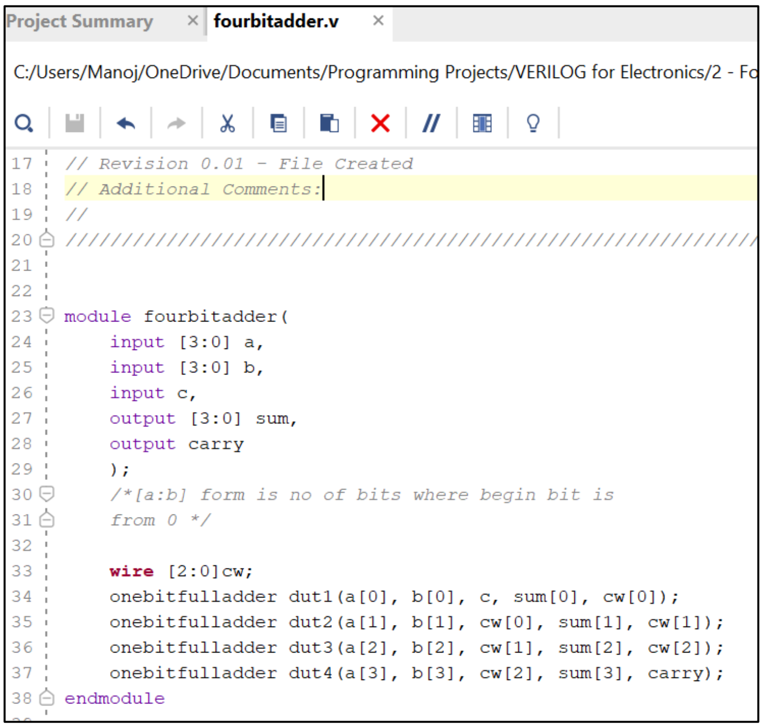
2. 4-Bit Adder:
   - Consists of four full adders connected in series.
   - The carry-out of each full adder is fed as the carry-in to the next.
   - The least significant bits (LSBs) of the two 4-bit numbers are fed into the first full adder, and the most significant bits (MSBs) are fed into the fourth full adder.



Applications:

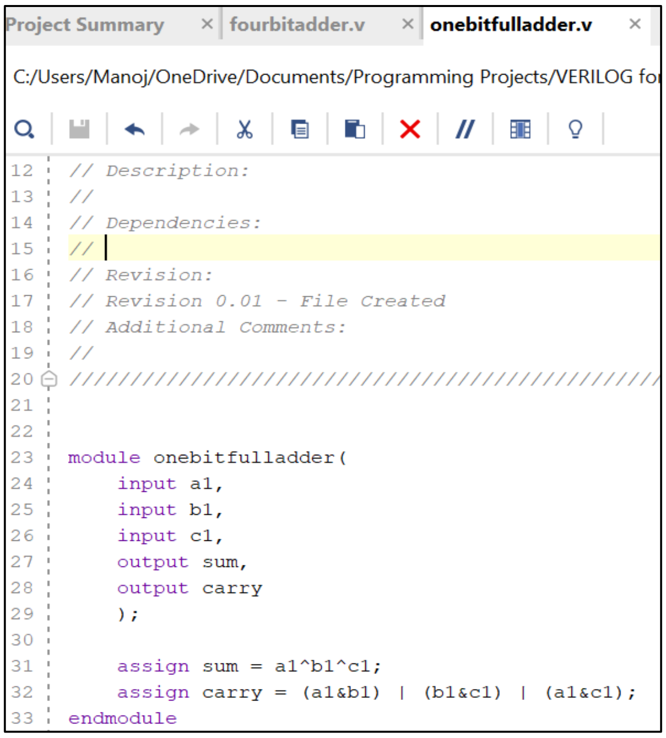
- *Arithmetic Logic Units (ALUs):* ALUs, the computational core of processors, use 4-bit adders to perform various arithmetic operations.
- *Digital Signal Processing (DSP):* 4-bit adders are used in DSP systems for filtering, modulation, and other signal processing tasks.
- *Microcontrollers:* Microcontrollers use 4-bit adders for basic arithmetic operations and address calculations.
- *Digital Counters:* 4-bit adders are used to increment and decrement counters.

RTL Code:



C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG for Electronics/2 - Fo

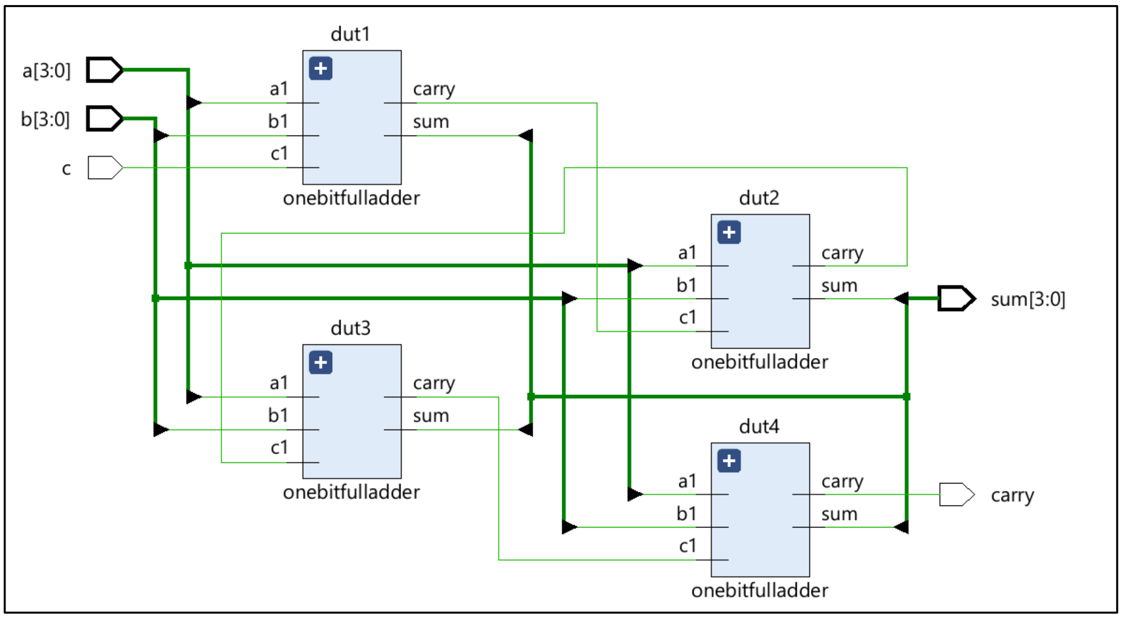
```
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////


module fourbitadder(
    input [3:0] a,
    input [3:0] b,
    input c,
    output [3:0] sum,
    output carry
    );
    /*[a:b] form is no of bits where begin bit is
    from 0 */

    wire [2:0]cw;
    onebitfulladder dut1(a[0], b[0], c, sum[0], cw[0]);
    onebitfulladder dut2(a[1], b[1], cw[0], sum[1], cw[1]);
    onebitfulladder dut3(a[2], b[2], cw[1], sum[2], cw[2]);
    onebitfulladder dut4(a[3], b[3], cw[2], sum[3], carry);
endmodule
```



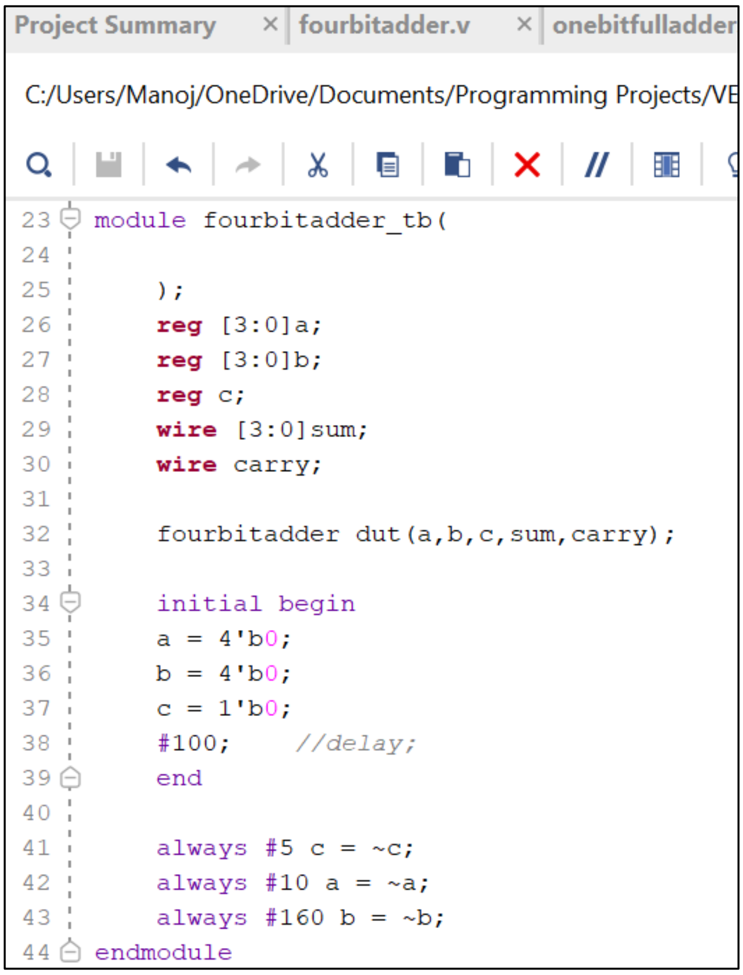
C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG fo

```
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////


module onebitfulladder(
    input a1,
    input b1,
    input c1,
    output sum,
    output carry
    );

    assign sum = a1^b1^c1;
    assign carry = (a1&b1) | (b1&c1) | (a1&c1);
endmodule
```

Schematic Diagram:



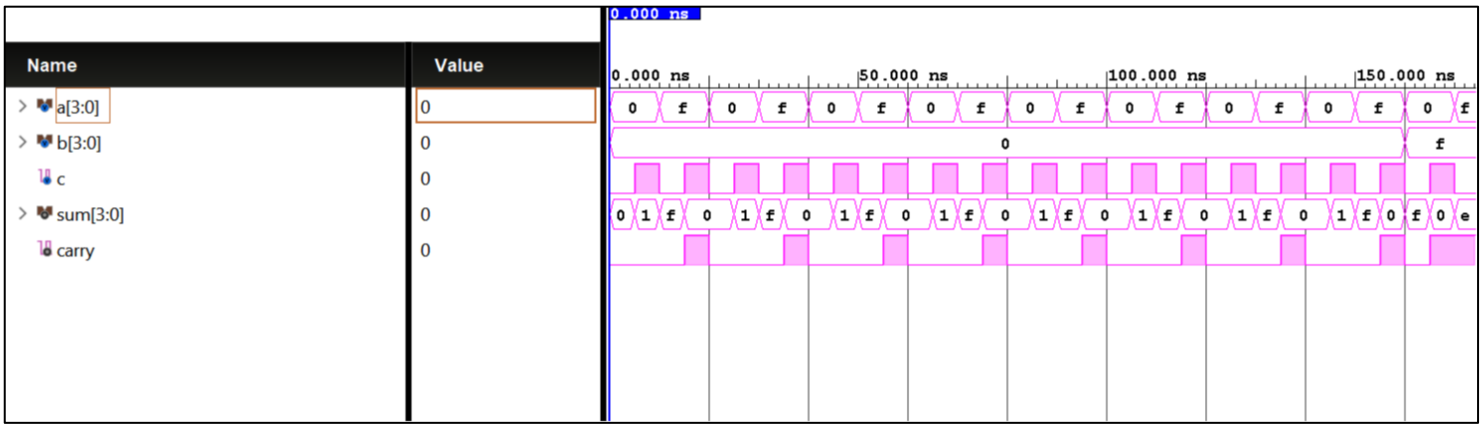
Testbench:



C:/Users/Manoj/OneDrive/Documents/Programming Projects/VE

```
module fourbitadder_tb(

    );
    reg [3:0]a;
    reg [3:0]b;
    reg c;
    wire [3:0]sum;
    wire carry;

    fourbitadder dut(a,b,c,sum,carry);

    initial begin
    a = 4'b0;
    b = 4'b0;
    c = 1'b0;
    #100;    //delay;
    end

    always #5 c = ~c;
    always #10 a = ~a;
    always #160 b = ~b;
endmodule
```

Wave Form:



Conclusion:

By understanding the fundamental principles of 4-bit adders, you can appreciate the building blocks of more complex digital systems and their role in modern technology.

Project 3: Multiplexers

Aim: To implement and demonstrate how multiple inputs can yield one output by multiplexer (MUX) device.

Theory:

<u>What is a Multiplexer?</u>

- A multiplexer, also known as a data selector, is a device that selects one of several input signals and forwards the selected input to a single output line.
- Think of it as a multi-position switch where the selection of which input to connect to the output is controlled by digital signals.
- It's a fundamental building block in digital electronics, used in a wide range of applications.

<u>Key Components:</u>

- Input Lines (I0, I1, I2, ...): These are the data sources that you want to switch between.
- Select Lines (S0, S1, ...): These are the control signals that determine which input line is connected to the output.
- Output Line (Y): This carries the selected input signal.

<u>How it Works:</u>

1. Select Line Decoding: The select lines are decoded to generate a unique combination of signals for each input line.

2. Input Selection: Based on the decoded signals, a specific input line is enabled.
3. Output Transmission: The enabled input is connected to the output line, and the selected data is transmitted.

Types of Multiplexers:

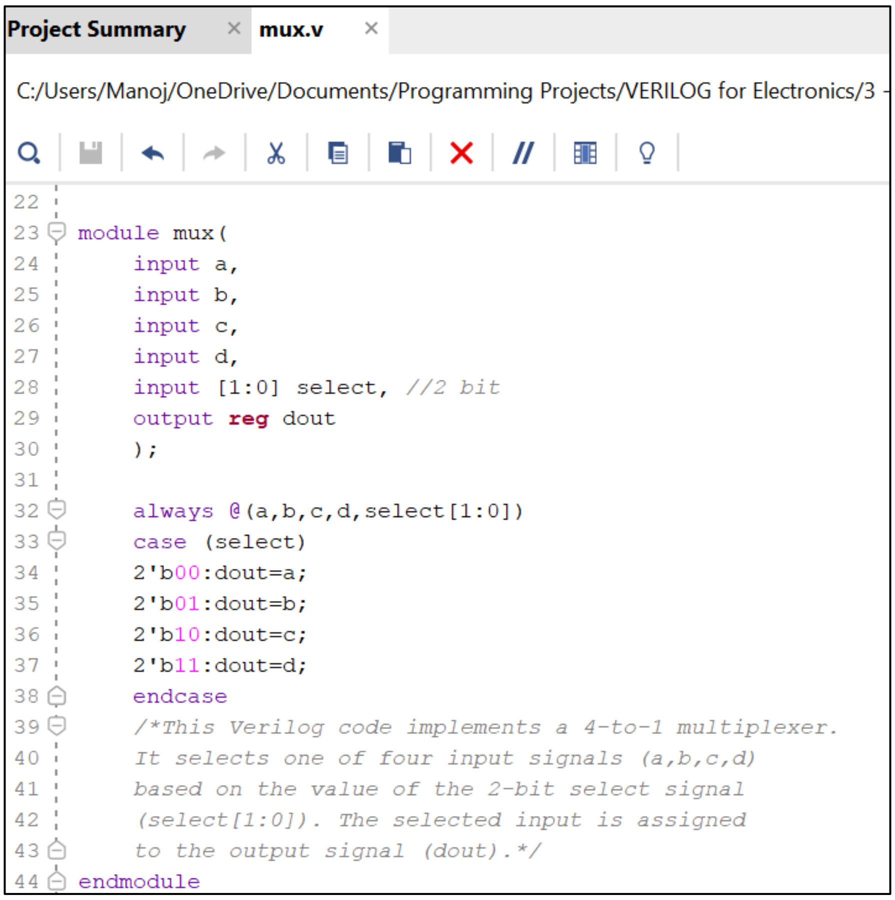
The number of input lines determines the type of multiplexer. Common types include:

- 2-to-1 Multiplexer
- 4-to-1 Multiplexer
- 8-to-1 Multiplexer
- 16-to-1 Multiplexer

Applications:

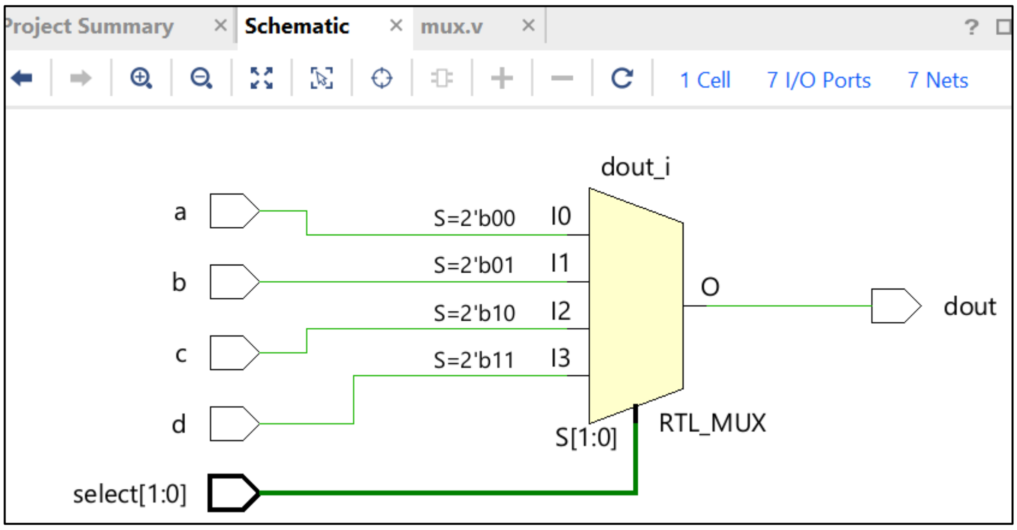
- Data Switching: Selecting different data sources for a single output.
- Address Decoding: Selecting specific memory locations.
- Protocol Switching: Switching between different communication protocols.
- Digital Signal Processing: Implementing various signal processing functions.

RTL Code:

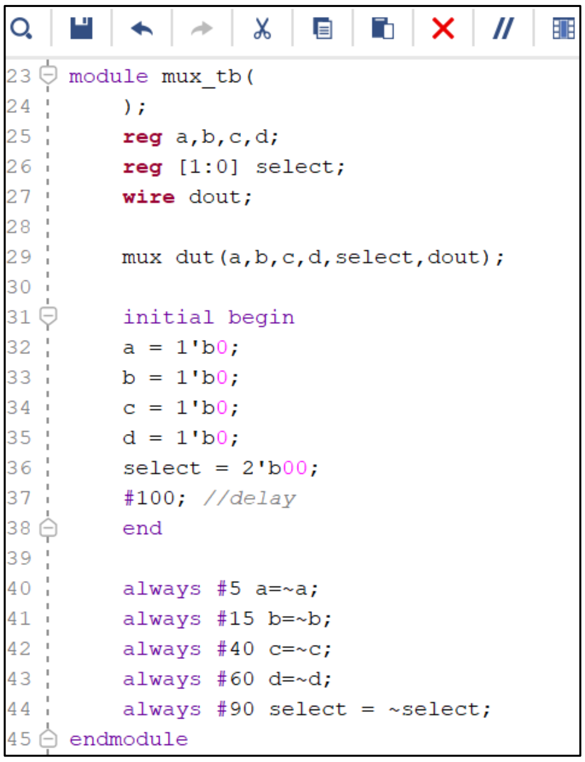

Project Summary
mux.v
C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG for Electronics/3

```verilog

module mux(
    input a,
    input b,
    input c,
    input d,
    input [1:0] select, //2 bit
    output reg dout
    );

    always @(a,b,c,d,select[1:0])
    case (select)
    2'b00:dout=a;
    2'b01:dout=b;
    2'b10:dout=c;
    2'b11:dout=d;
    endcase
    /*This Verilog code implements a 4-to-1 multiplexer.
    It selects one of four input signals (a,b,c,d)
    based on the value of the 2-bit select signal
    (select[1:0]). The selected input is assigned
    to the output signal (dout).*/
endmodule
```

Schematic Diagram:

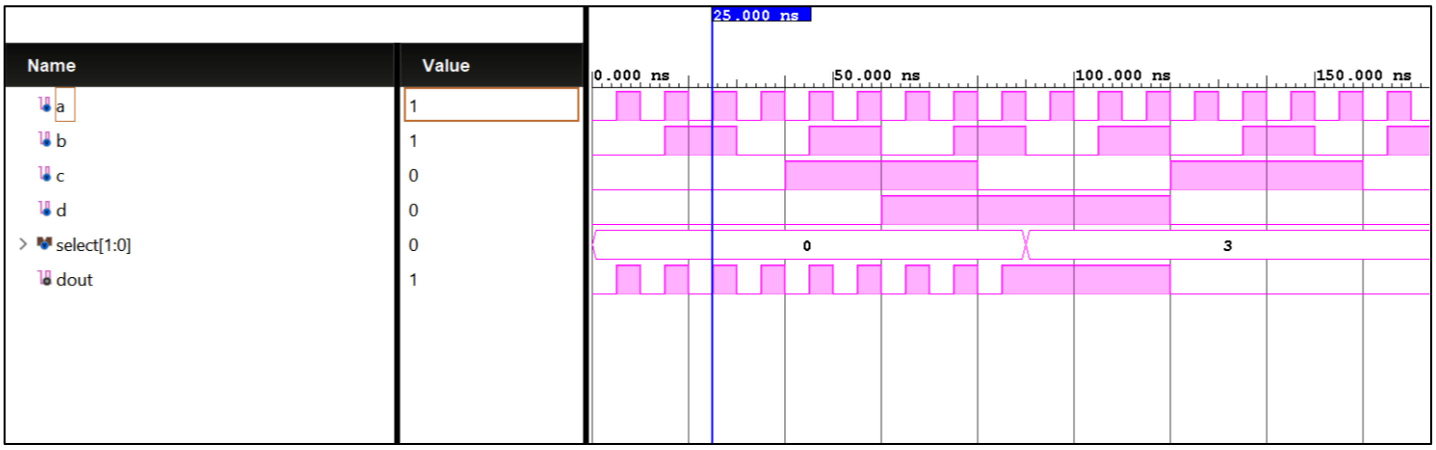


Testbench:



```
module mux_tb(
    );
    reg a,b,c,d;
    reg [1:0] select;
    wire dout;

    mux dut(a,b,c,d,select,dout);

    initial begin
    a = 1'b0;
    b = 1'b0;
    c = 1'b0;
    d = 1'b0;
    select = 2'b00;
    #100; //delay
    end

    always #5 a=~a;
    always #15 b=~b;
    always #40 c=~c;
    always #60 d=~d;
    always #90 select = ~select;
endmodule
```

Wave Form:



Name	Value
a	1
b	1
c	0
d	0
select[1:0]	0
dout	1

Conclusion:

To summarize, a multiplexer (MUX) is a digital circuit that selects one of several input signals and forwards the selected input to a single output line. The selection is controlled by a set of select lines. Verilog HDL provides a concise and efficient way to implement multiplexers using case statements and always blocks.

Project 4: Flip Flops: The Building Blocks of Digital Memory

Aim: To demonstrate the usage of Flip Flop to understand its significance in Electronics.

Theory:

Flip-flops are fundamental electronic circuits that form the backbone of digital systems. They are essentially bistable multivibrators, meaning they can exist in one of two stable states: 0 or 1. These states are maintained until a specific input signal triggers a change.

Types of Flip-Flops

1. *SR Flip-Flop (Set-Reset Flip-Flop):*
   - Inputs: S (Set), R (Reset)
   - Outputs: Q (Output), Q' (Complement of Q)
   - Operation:
     - S = 0, R = 0: No change in the output.
     - S = 1, R = 0: Sets the output Q to 1.
     - S = 0, R = 1: Resets the output Q to 0.
     - S = 1, R = 1: This is an indeterminate state and should be avoided.
2. *JK Flip-Flop:*
   - Inputs: J, K, Clock
   - Outputs: Q, Q'
   - Operation:
     - J = 0, K = 0: No change in the output.
     - J = 0, K = 1: Resets the output Q to 0.
     - J = 1, K = 0: Sets the output Q to 1.
     - J = 1, K = 1: Toggles the output Q.

3. *T Flip-Flop (Toggle Flip-Flop):*
    - Input: T (Toggle), Clock
    - Outputs: Q, Q'
    - Operation:
        - T = 0: No change in the output.
        - T = 1: Toggles the output Q.
4. *D Flip-Flop (Data Flip-Flop):*
    - Input: D (Data), Clock
    - Outputs: Q, Q'
    - Operation:
        - On the positive edge of the clock pulse, the input D is transferred to the output Q.

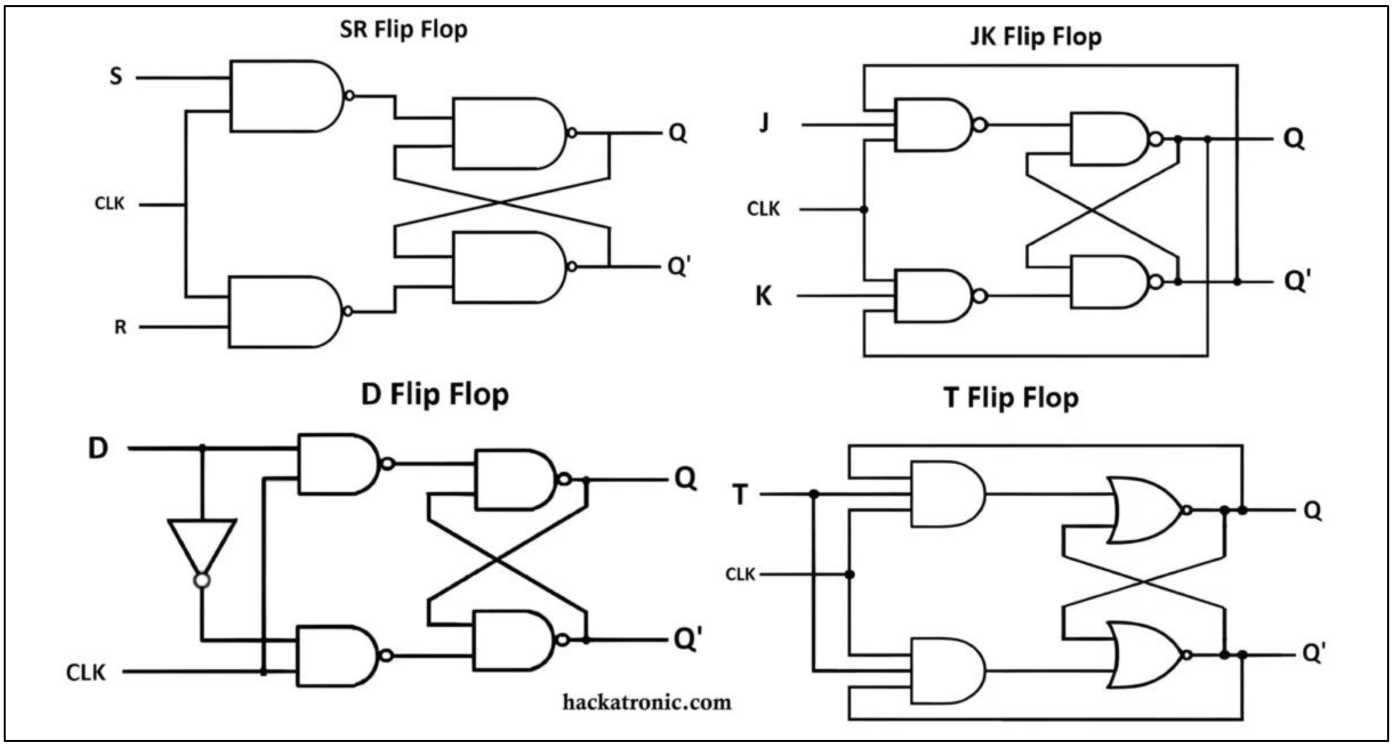
Clock Signal

A clock signal is a periodic square wave that synchronizes the operation of flip-flops and other sequential logic circuits. It provides a timing reference for when the flip-flop should change its state.

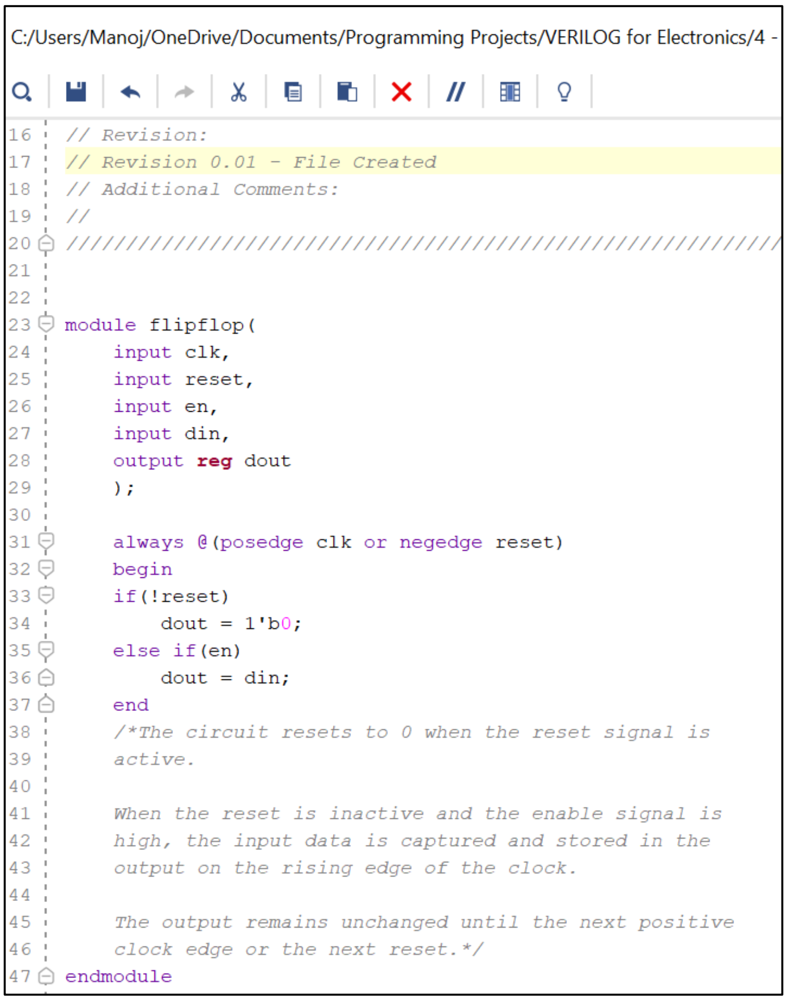
<u>Applications of Flip-Flops</u>

Flip-flops are ubiquitous in digital systems, including:

- Registers: To store data.
- Counters: To count events.
- Shift registers: To shift data.
- Finite state machines: To implement sequential logic.
- Memory units: To store large amounts of data.



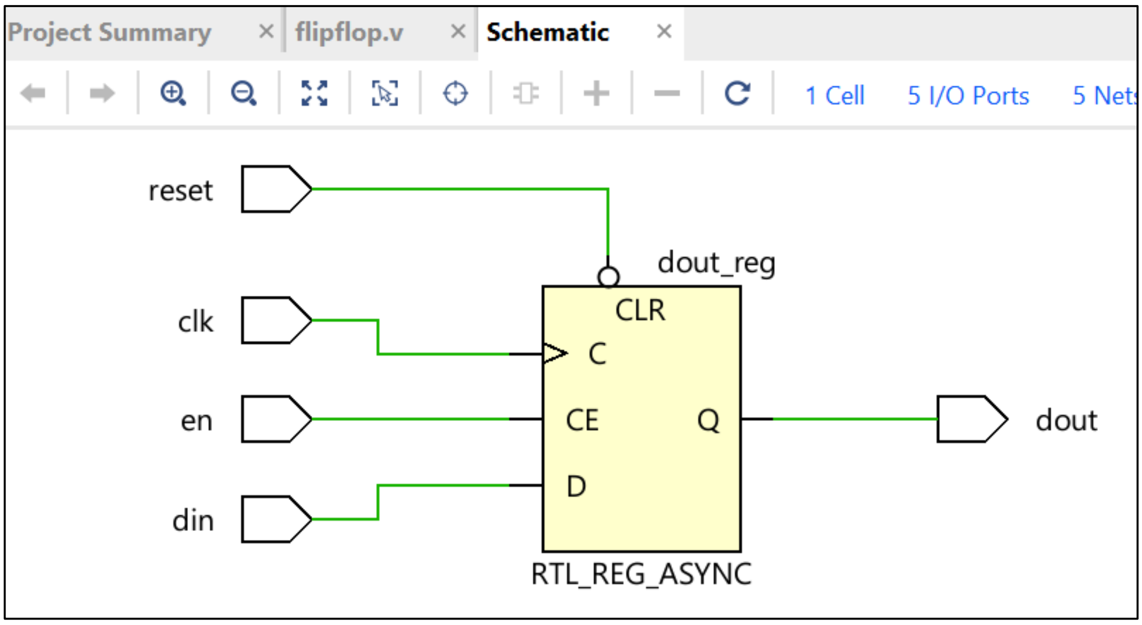
RTL Code:



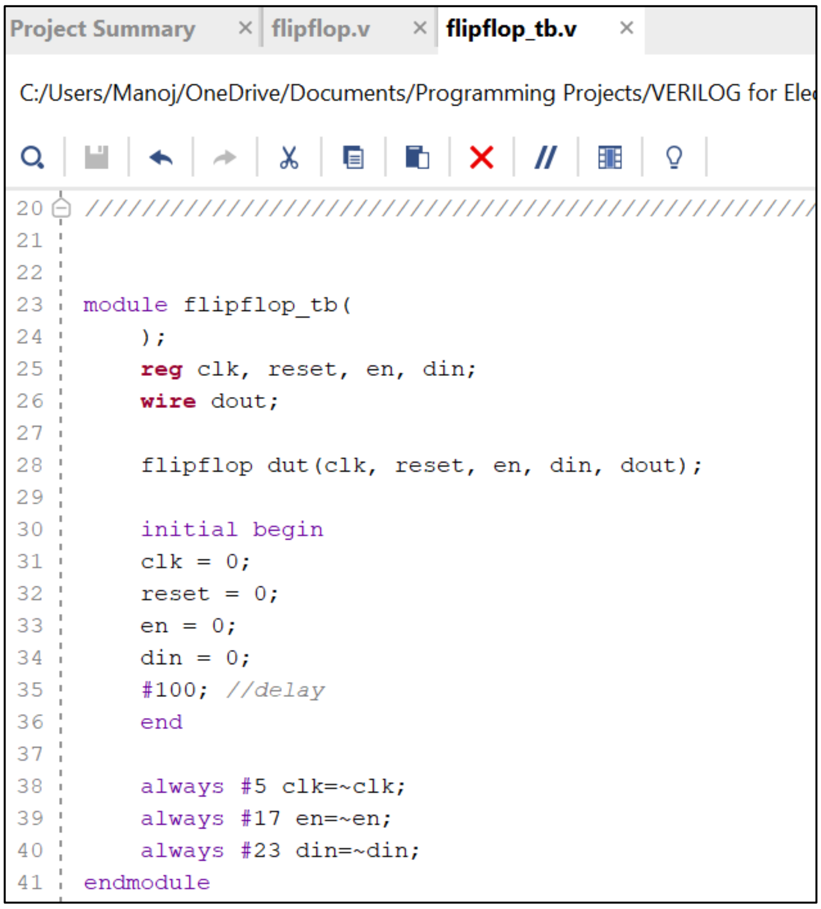
C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG for Electronics/4 -

```verilog
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////


module flipflop(
    input clk,
    input reset,
    input en,
    input din,
    output reg dout
    );

    always @(posedge clk or negedge reset)
    begin
    if(!reset)
        dout = 1'b0;
    else if(en)
        dout = din;
    end
    /*The circuit resets to 0 when the reset signal is
    active.

    When the reset is inactive and the enable signal is
    high, the input data is captured and stored in the
    output on the rising edge of the clock.

    The output remains unchanged until the next positive
    clock edge or the next reset.*/
endmodule
```

Schematic Diagram:



Testbench:

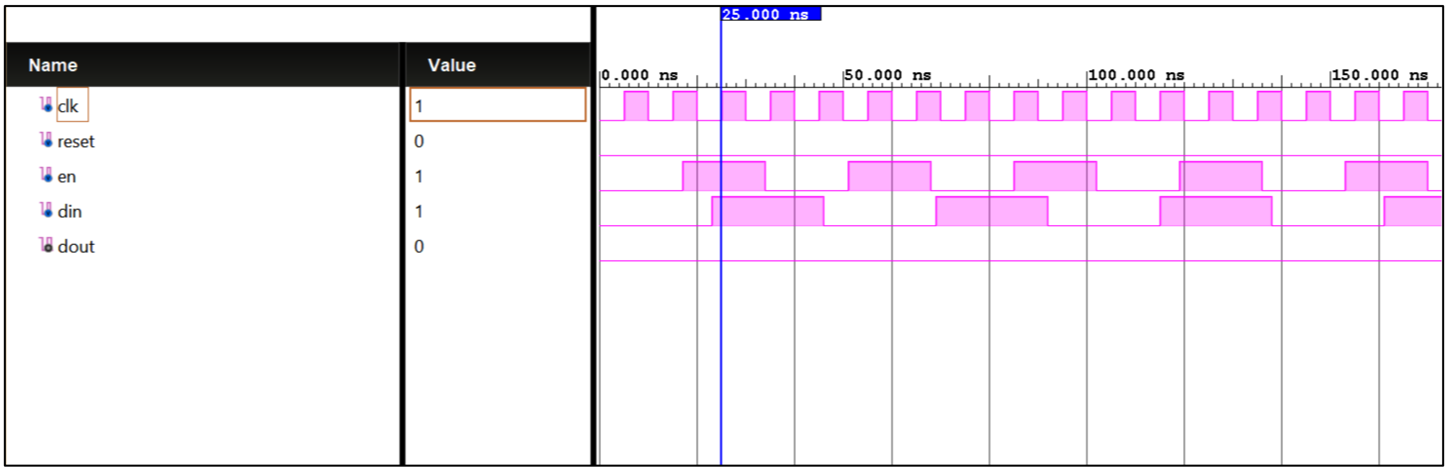


Project Summary × flipflop.v × flipflop_tb.v ×

C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG for Ele

```
////////////////////////////////////////////////////


module flipflop_tb(
    );
    reg clk, reset, en, din;
    wire dout;

    flipflop dut(clk, reset, en, din, dout);

    initial begin
    clk = 0;
    reset = 0;
    en = 0;
    din = 0;
    #100; //delay
    end

    always #5 clk=~clk;
    always #17 en=~en;
    always #23 din=~din;
endmodule
```

Wave Form:



Name	Value
clk	1
reset	0
en	1
din	1
dout	0

Conclusion:

In conclusion, flip-flops are essential components in digital electronics. Their ability to store and manipulate binary information makes them indispensable in the design of modern electronic devices. By understanding their operation and characteristics, engineers can create complex digital systems with diverse functionalities.

Project 5: Implementation of Latches

Aim: To demonstrate and understand concept of latches after Flipflop experimentation.

Theory:

Latches are Building blocks of sequential circuits. A latch is a fundamental digital circuit element that stores a single bit of information. It's a simple sequential circuit with one or more inputs and one output. The output of a latch remains stable until a specific input condition forces it to change.

Types of Latches

1. *SR Latch:*
    - Inputs: S (Set), R (Reset)
    - Output: Q
    - Operation:
        - S=1, R=0: Q=1 (Set)
        - S=0, R=1: Q=0 (Reset)
        - S=0, R=0: No change in Q
        - S=1, R=1: Indeterminate state (should be avoided)
2. *JK Latch:*
    - Inputs: J (Set), K (Reset)
    - Output: Q
    - Operation:
        - J=1, K=0: Q=1 (Set)
        - J=0, K=1: Q=0 (Reset)
        - J=0, K=0: No change in Q
        - J=1, K=1: Toggle (Q inverts)
3. *Latch with Enable:*
    - Inputs: D (Data), E (Enable)
    - Output: Q
    - Operation:
        - E=1: Q=D (Data is latched)
        - E=0: No change in Q

Key Points:

- Latches are sensitive to input changes.
- They can be used to store information temporarily.
- Latches are often used as building blocks for flip-flops.
- It's crucial to avoid the indeterminate state in SR latches.

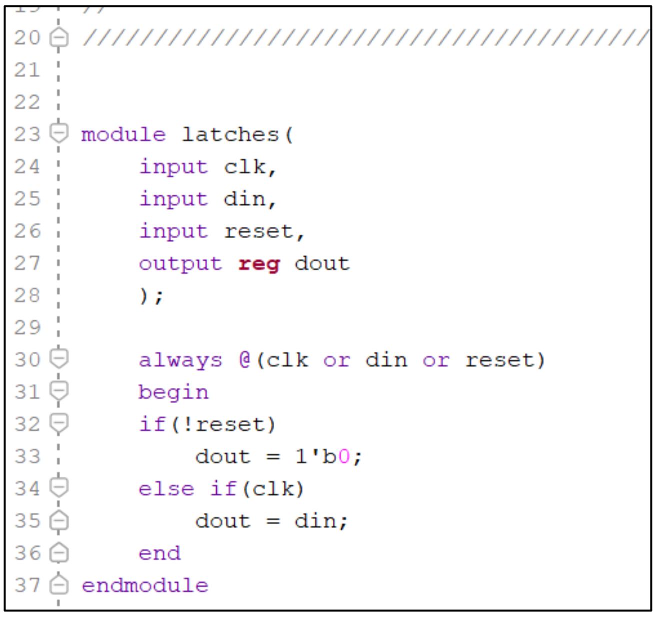
Applications:

- Data storage in sequential circuits
- Building flip-flops
- Designing counters and registers

How are they different from Flipflops?

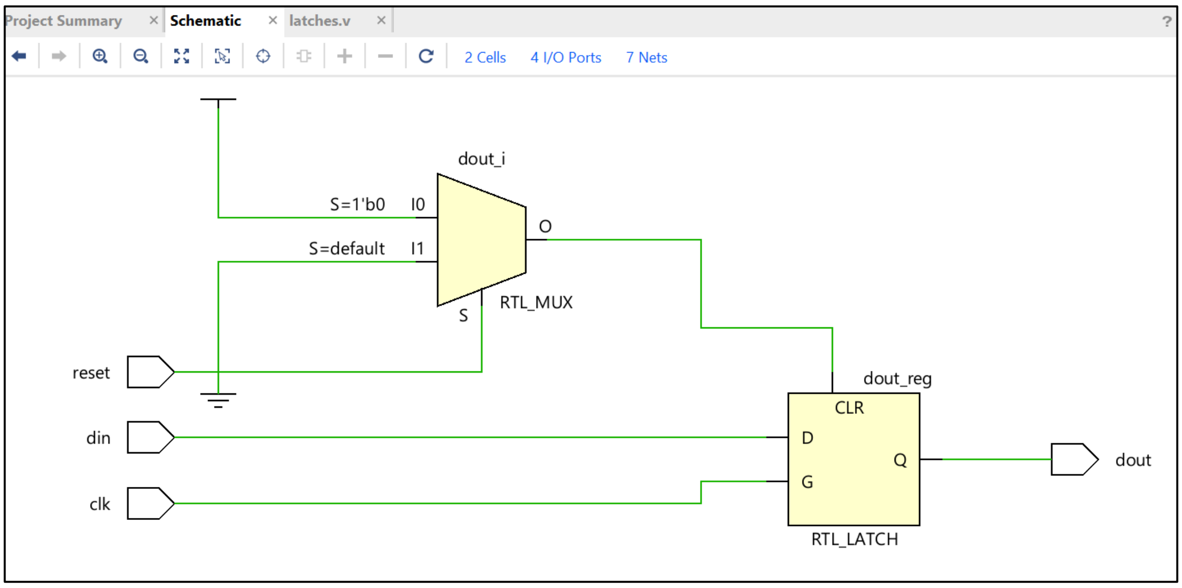
Point of Distinction	**Latch**	**Flip-Flop**
Triggering Mechanism	Level-Triggered	Edge-Triggered
Clock Signal	Not Required	Required
Sensitivity to Input Changes	Continuously Sensitive	Sensitive only at the active edge of the clock
Output Change	Changes immediately when input changes (while enabled)	Changes only at the active edge of the clock
Synchronization	Asynchronous	Synchronous
Timing Precision	Less precise	More precise
Complexity	Simpler	More complex
Stability	Less stable	More stable
Applications	Simpler circuits, data storage	Complex sequential circuits, counters, registers

RTL Codes:

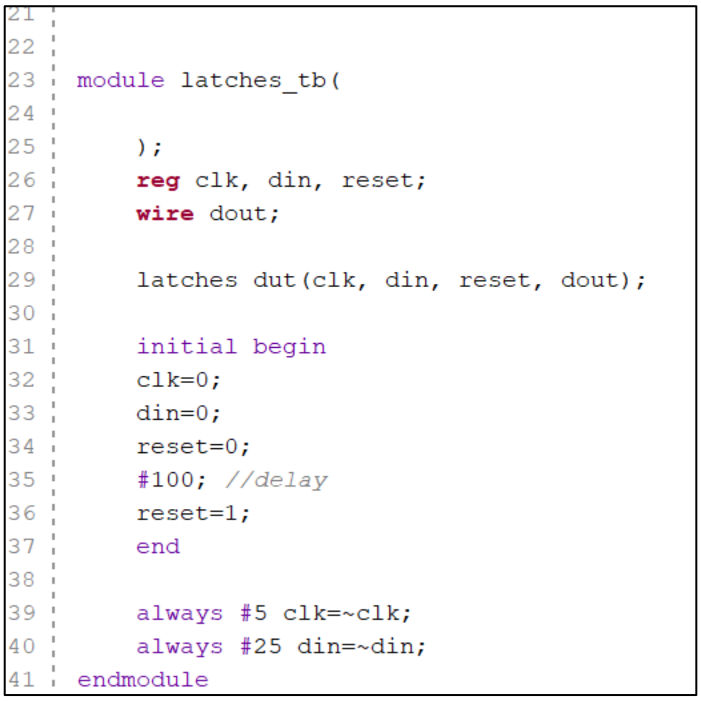


```
//////////////////////////////////////////


module latches(
    input clk,
    input din,
    input reset,
    output reg dout
    );

    always @(clk or din or reset)
    begin
    if(!reset)
        dout = 1'b0;
    else if(clk)
        dout = din;
    end
endmodule
```

Schematic Diagram:

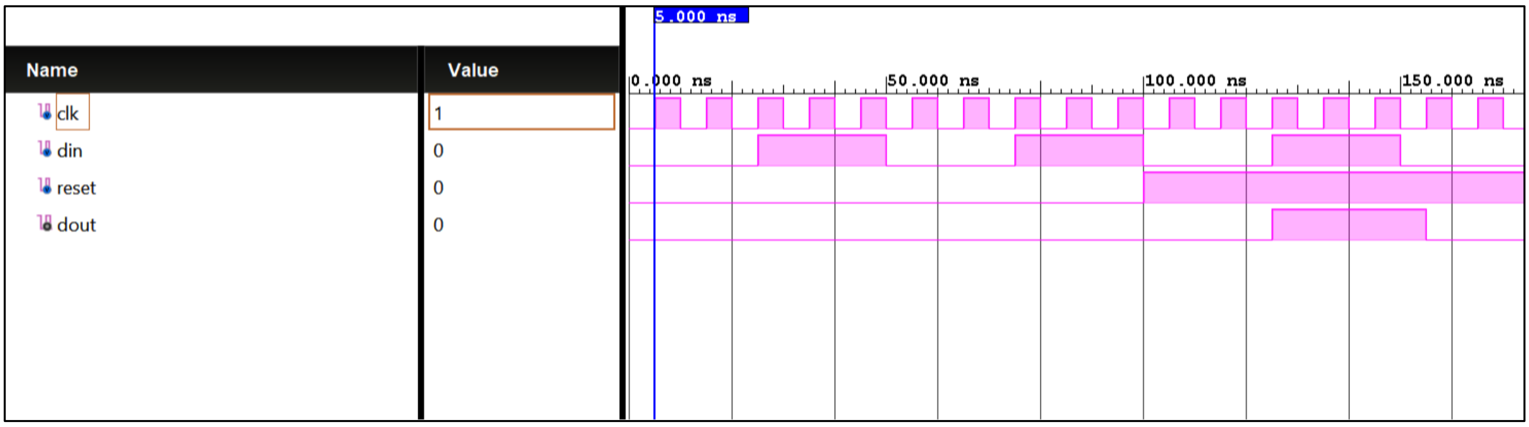


Testbench:



```
module latches_tb(

    );
    reg clk, din, reset;
    wire dout;

    latches dut(clk, din, reset, dout);

    initial begin
    clk=0;
    din=0;
    reset=0;
    #100; //delay
    reset=1;
    end

    always #5 clk=~clk;
    always #25 din=~din;
endmodule
```

Wave Form:



Name	Value
clk	1
din	0
reset	0
dout	0

Conclusion:

In essence, latches are fundamental building blocks in digital circuits, serving as simple memory elements. They store a single bit of information and are essential for various applications, including data storage, control circuits, and the construction of more complex sequential logic components like flip-flops.

Project 6: Counters

Aim: To understand and imply the concepts of counters.

Theory:

Counters are sequential circuits used for counting purpose – analogy to `int counter++;` in general program languages. They are a fundamental type of sequential circuit that increment or decrement a stored numerical value with each clock pulse. They are widely used in digital systems for timing, frequency division, address generation, and various other applications.

Types of Counters

1. *Asynchronous Counters:*
   - Each flip-flop in an asynchronous counter is triggered by the output of the preceding flip-flop.
   - This leads to a propagation delay between stages, limiting the maximum operating frequency.
   - Common types include ripple counters and synchronous counters.
2. *Synchronous Counters:*
   - All flip-flops in a synchronous counter are triggered by the same clock edge.
   - This allows for higher operating frequencies compared to asynchronous counters.
   - Common types include binary counters, decade counters, and modulo-N counters.

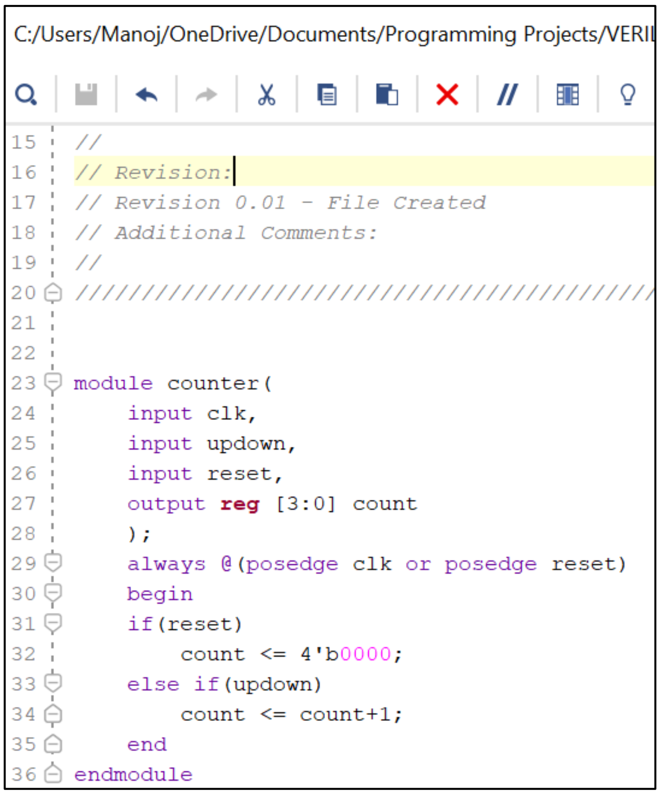
Design Considerations for Counters

1. Modulus: The modulus of a counter determines the number of states it can cycle through.
2. Clock Signal: The clock signal provides the timing reference for the counter's operation.
3. Reset Signal: The reset signal can be used to initialize the counter to a specific state.
4. Enable Signal: The enable signal controls whether the counter is active or inactive.
5. Output Encoding: The output of the counter can be encoded in various formats, such as binary, BCD, or Gray code.

Applications of Counters

- Timing Circuits: Counters can be used to generate time delays or measure time intervals.
- Frequency Division: Counters can divide a high-frequency clock signal into a lower-frequency signal.
- Address Generation: Counters can generate sequential addresses for memory access.
- Digital Clock Design: Counters are used to keep track of seconds, minutes, hours, and other time units.
- Digital Signal Processing: Counters can be used for various signal processing tasks, such as filtering and modulation.

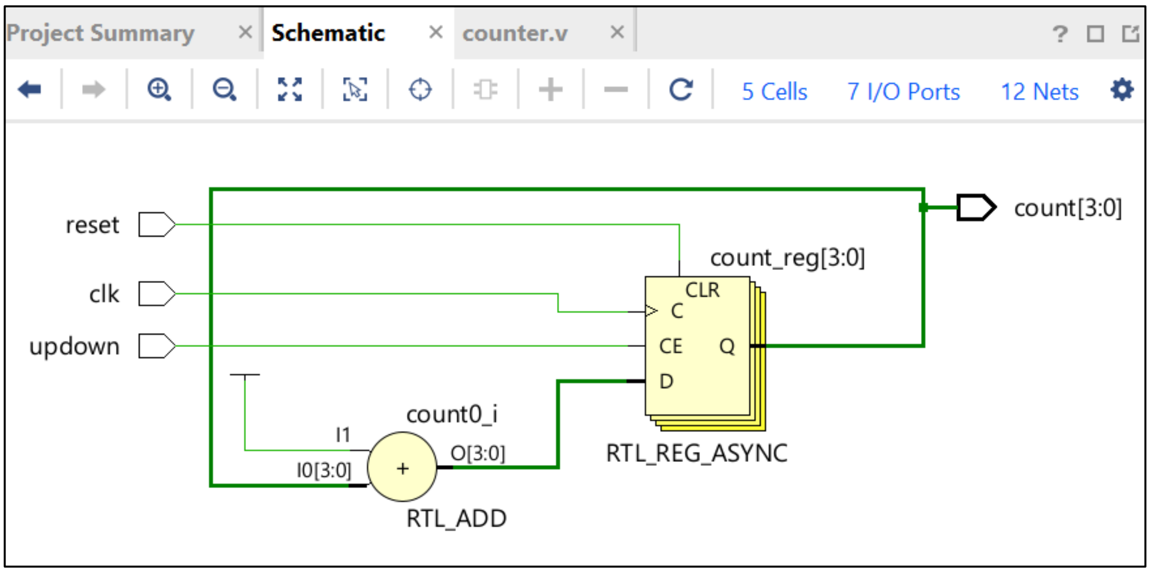
RTL Code:



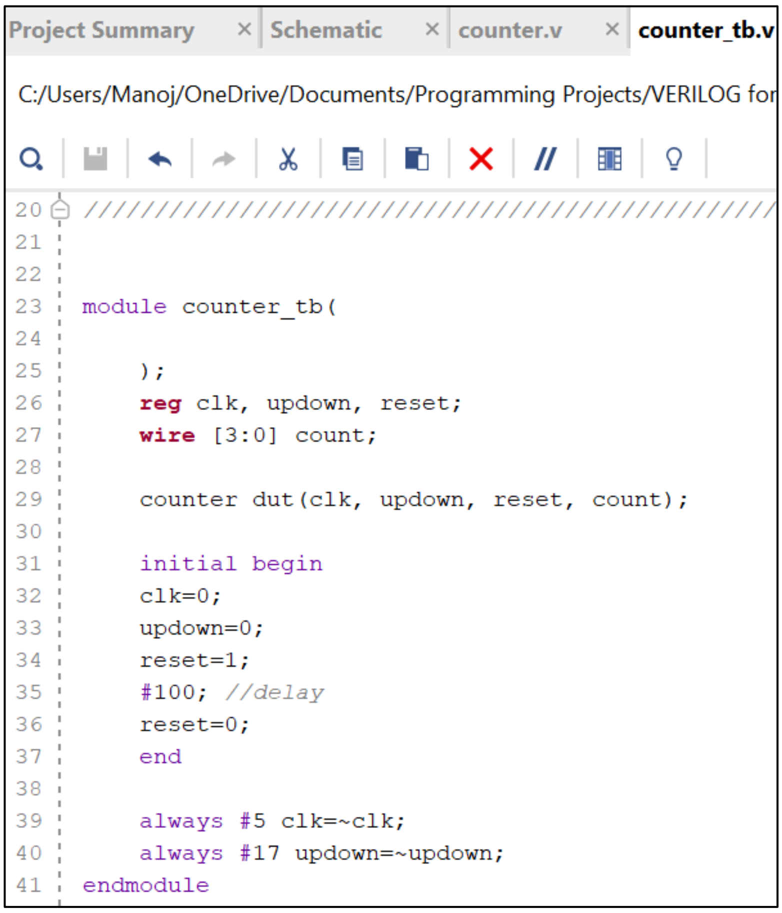
C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERIL

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////


module counter(
    input clk,
    input updown,
    input reset,
    output reg [3:0] count
    );
    always @(posedge clk or posedge reset)
    begin
    if(reset)
        count <= 4'b0000;
    else if(updown)
        count <= count+1;
    end
endmodule
```

Schematic Diagram:



Testbench:

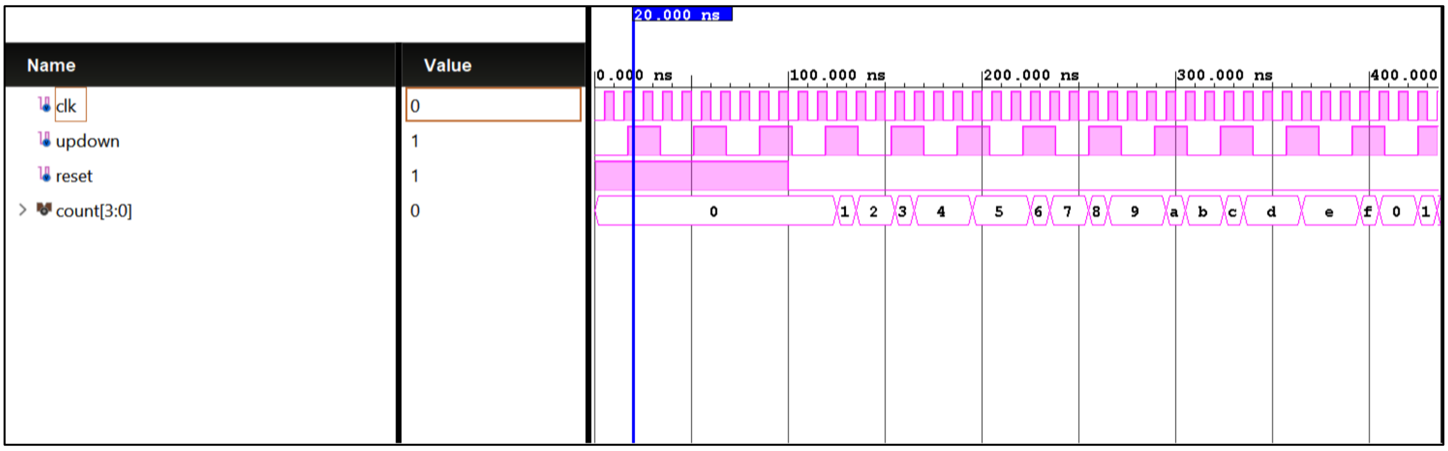


Project Summary × Schematic × counter.v × counter_tb.v

C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG for

```
//////////////////////////////////////////////////


module counter_tb(

    );
    reg clk, updown, reset;
    wire [3:0] count;

    counter dut(clk, updown, reset, count);

    initial begin
    clk=0;
    updown=0;
    reset=1;
    #100; //delay
    reset=0;
    end

    always #5 clk=~clk;
    always #17 updown=~updown;
endmodule
```

Wave Form:



Name	Value
clk	0
updown	1
reset	1
count[3:0]	0

Conclusion:

In summation, counters are indispensable components in digital circuits, serving as sequential circuits that systematically increment or decrement a stored numerical value with each clock pulse. They are widely employed in diverse applications, including timing circuits, frequency division, address generation, and digital signal processing.

Project 7: Demonstration of Serial In Serial Out Register

Aim: To understand and demonstrate Serial In Serial Out Register and how it is useful in Electronics DSD.

Theory:

A Serial-In Serial-Out (SISO) shift register is a fundamental digital circuit that sequentially shifts data bits into and out of a series of flip-flops. This type of shift register is characterized by its simple structure and operation, making it a versatile component in many digital systems.

Core Components and Operation:

- *Flip-Flops*: The primary building blocks of a SISO shift register are flip-flops, typically D flip-flops. Each flip-flop stores one bit of data.
- *Clock Signal*: A clock signal synchronizes the shifting operation.
- *Serial Input*: The data to be shifted into the register is applied to the input of the first flip-flop.
- *Serial Output*: The data shifted out of the last flip-flop is the serial output.

Operation:

1. Data Input: The data to be stored is applied to the input of the first flip-flop.
2. Clock Pulse: With each clock pulse, the data in each flip-flop is shifted to the next flip-flop.
3. Data Output: The data that was originally in the last flip-flop is shifted out and becomes the output.

Key Characteristics:

- Serial Data Transfer: Data is transferred one bit at a time.
- Simple Structure: It consists of a chain of flip-flops.
- Delay: Each stage introduces a delay, limiting the maximum clock frequency.

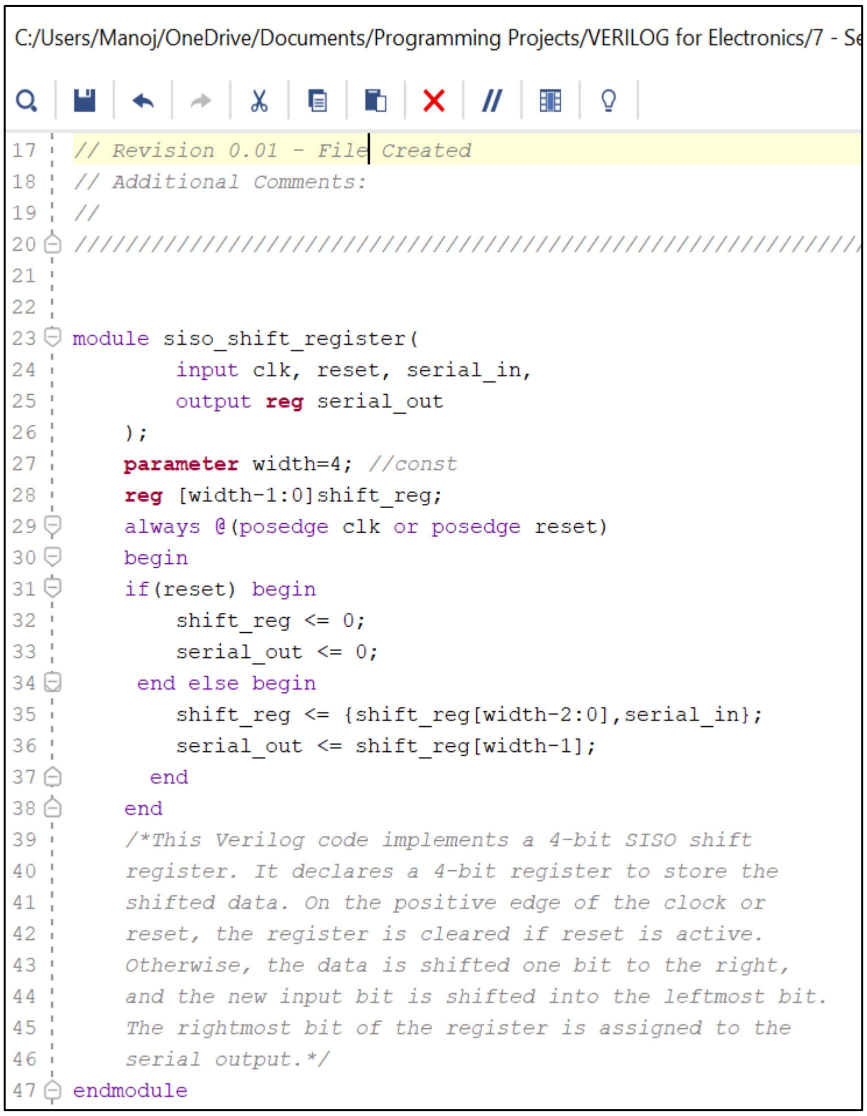
Applications:

- Serial communication
- Time delay generation
- Data storage and retrieval

Limitations of SISO Shift Registers:

- *Low Data Transfer Rate*: Serial data transfer is inherently slower than parallel data transfer.
- *Susceptibility to Noise*: Long shift registers can be more prone to noise interference.

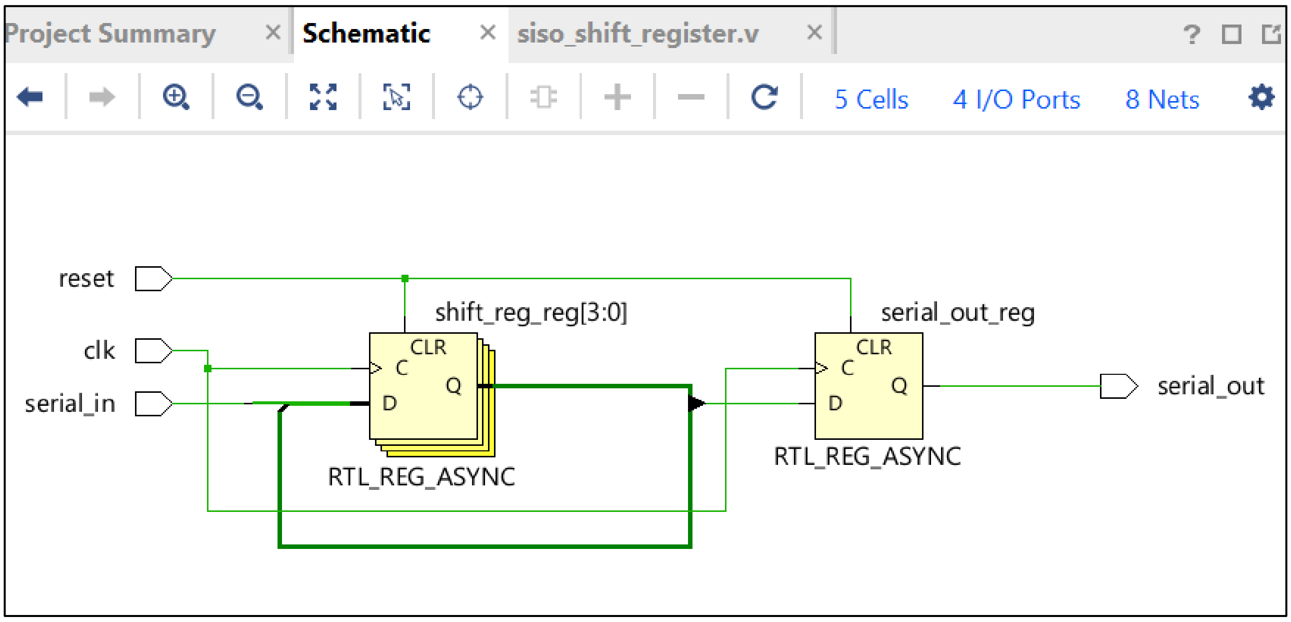
RTL Code:



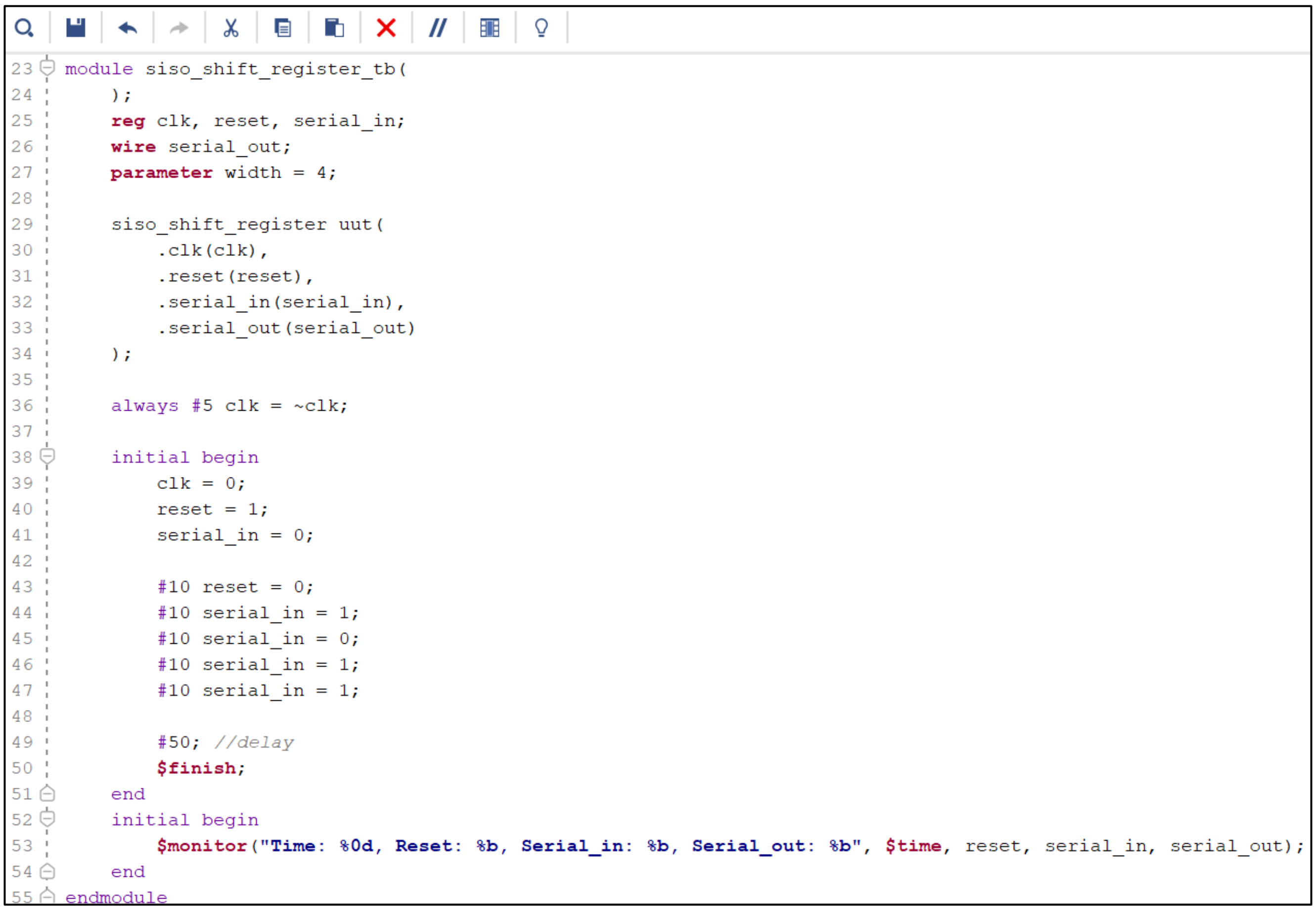
C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG for Electronics/7 - Se

```verilog
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////


module siso_shift_register(
        input clk, reset, serial_in,
        output reg serial_out
    );
    parameter width=4; //const
    reg [width-1:0]shift_reg;
    always @(posedge clk or posedge reset)
    begin
    if(reset) begin
        shift_reg <= 0;
        serial_out <= 0;
     end else begin
        shift_reg <= {shift_reg[width-2:0],serial_in};
        serial_out <= shift_reg[width-1];
      end
    end
    /*This Verilog code implements a 4-bit SISO shift
    register. It declares a 4-bit register to store the
    shifted data. On the positive edge of the clock or
    reset, the register is cleared if reset is active.
    Otherwise, the data is shifted one bit to the right,
    and the new input bit is shifted into the leftmost bit.
    The rightmost bit of the register is assigned to the
    serial output.*/
endmodule
```

Schematic Diagram:



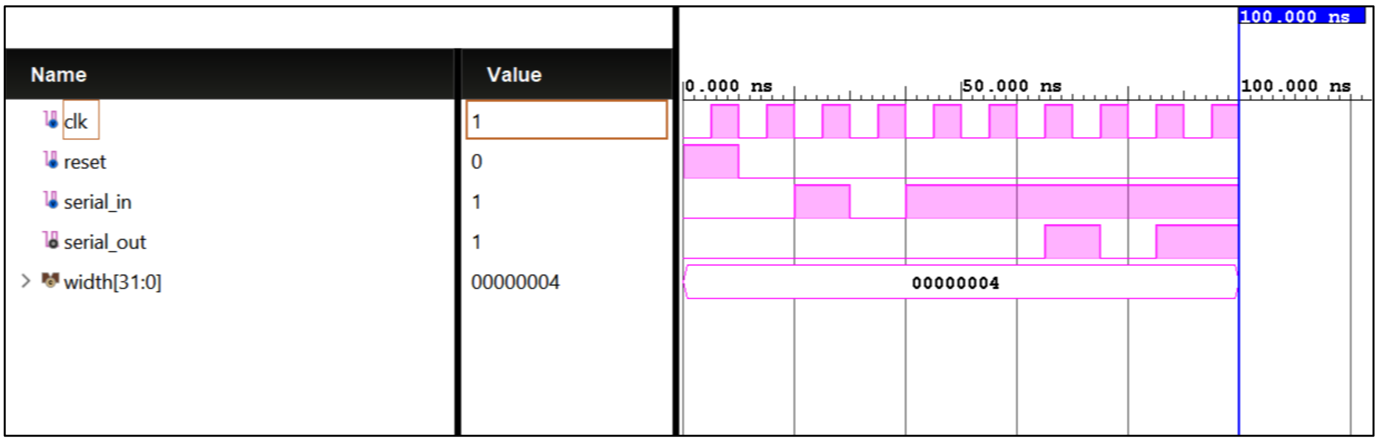
Testbench:



```
module siso_shift_register_tb(
    );
    reg clk, reset, serial_in;
    wire serial_out;
    parameter width = 4;

    siso_shift_register uut(
        .clk(clk),
        .reset(reset),
        .serial_in(serial_in),
        .serial_out(serial_out)
    );

    always #5 clk = ~clk;

    initial begin
        clk = 0;
        reset = 1;
        serial_in = 0;

        #10 reset = 0;
        #10 serial_in = 1;
        #10 serial_in = 0;
        #10 serial_in = 1;
        #10 serial_in = 1;

        #50; //delay
        $finish;
    end
    initial begin
        $monitor("Time: %0d, Reset: %b, Serial_in: %b, Serial_out: %b", $time, reset, serial_in, serial_out);
    end
endmodule
```

/* This Verilog code defines a testbench for a 4-bit SISO shift register. It instantiates the siso_shift_register module and applies a clock signal, reset signal, and serial input. The testbench monitors the serial output and displays the values on the console. It initializes the reset signal, applies a sequence of input bits, and then finishes the simulation after a delay.*/

Wave Form:



Name	Value
clk	1
reset	0
serial_in	1
serial_out	1
width[31:0]	00000004

Conclusion:

In a nutshell, SISO shift registers are foundational digital circuits that efficiently transfer and store data serially. Their simple structure and operation make them versatile components in diverse digital systems, though their serial data transfer nature can limit their performance in high-speed applications.

Project 8: Implementation of Finite State Machine (FSM) of a sequence detector

Aim: To implement and demonstrate the finite state machine of a sequence detector and understand how FSM operates.

Theory:

A Finite State Machine (FSM) is a mathematical model of computation used to design digital systems. It comprises a finite number of states, transitions between these states, and a set of input and output symbols. In the realm of sequence detection, an FSM can be employed to recognize specific sequences of input bits.

A typical FSM encompasses the following components:

1. *States*: A finite set of discrete states that the machine can occupy at any given moment.
2. *Input Alphabet*: The collection of potential input symbols that the machine can receive.
3. *Output Alphabet*: The set of potential output symbols that the machine can produce.
4. *Transition Function*: A function that maps the current state and input symbol to the subsequent state.
5. *Output Function*: A function that maps the current state and input symbol to the output symbol.

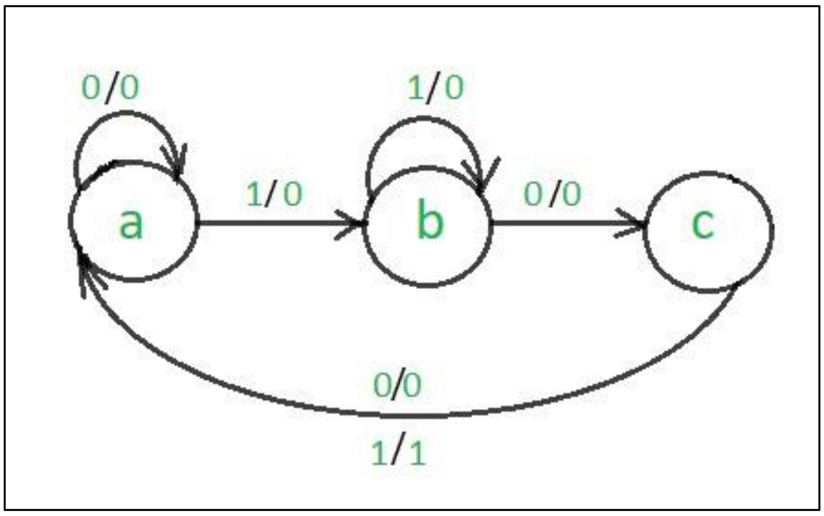
A sequence detector is the digital circuit that detects some input signal sequences from a set of the binary data. One can determine whether incoming bits are equal to a prestored sequence, thus widely used in communication systems, data processing, and digital signal processing. Meanwhile, implemented through several technologies, among them, state machines, and programmable logic devices, sequence detectors have applications in digital electronics and telecommunications. This technology can be applied in the area of bioinformatics, by detecting specific nucleotide sequences in DNA or RNA, thus showing its applicability across various fields.

The steps to design a non-overlapping 101 Mealy sequence detectors are:

-

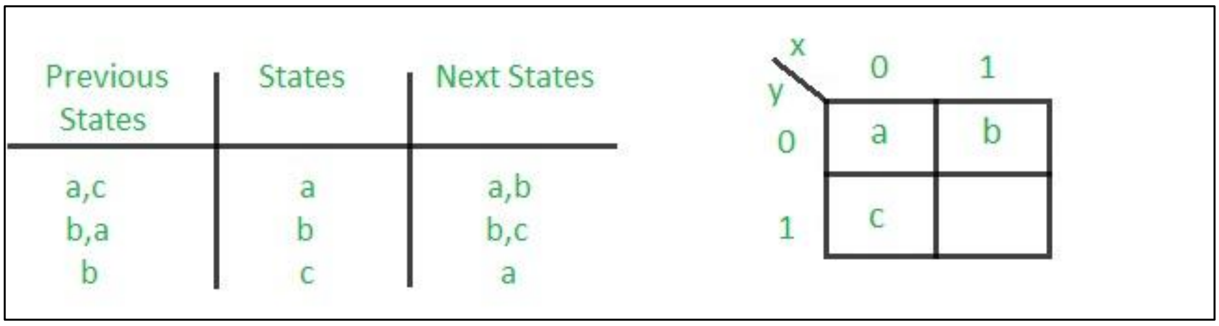
Step 1: Develop the state diagram –

The state diagram of a Mealy machine for a 101 sequence detector is:



Step 2: Code Assignment –

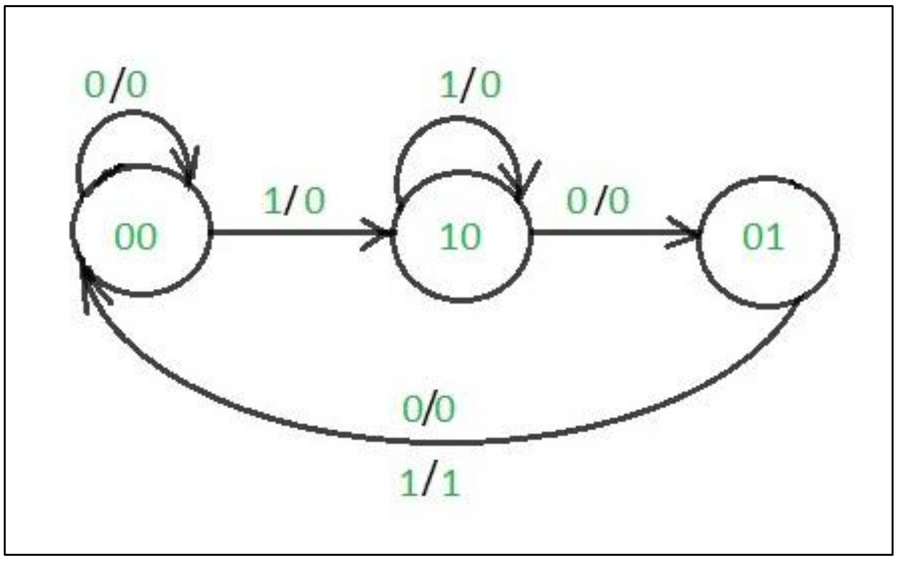
Rule 1: States having the same next states for a given input condition should have adjacent assignments.
Rule 2: States that are the next states to a single state must be given adjacent assignments.
Rule 1 given preference over Rule 2.



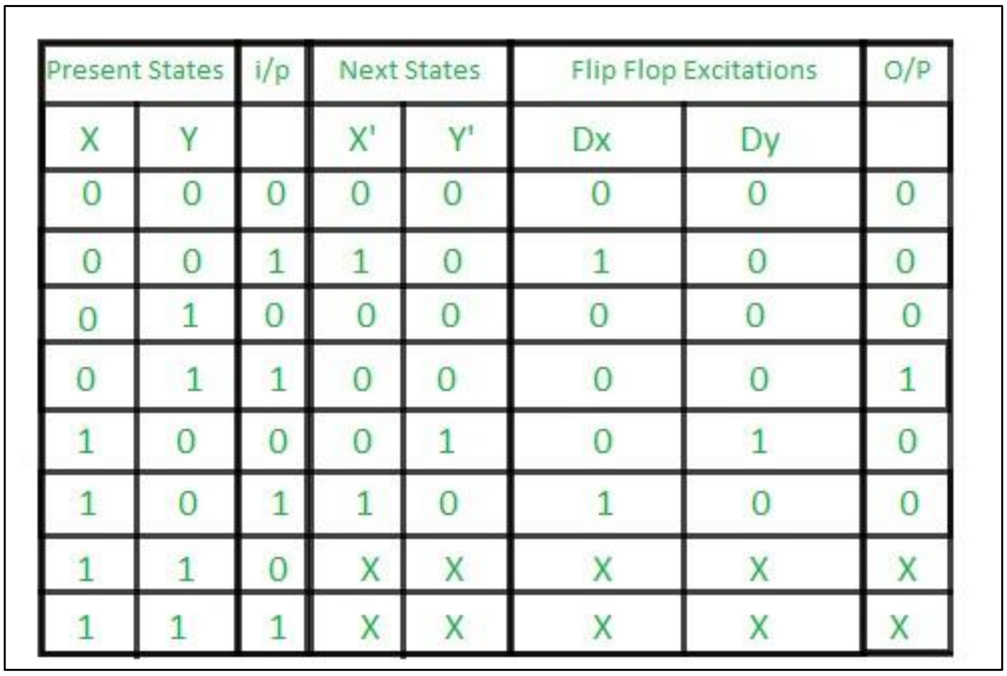
Previous States	States	Next States
a,c	a	a,b
b,a	b	b,c
b	c	a

y \ x	0	1
0	a	b
1	c	

The state diagram after the code assignment is:

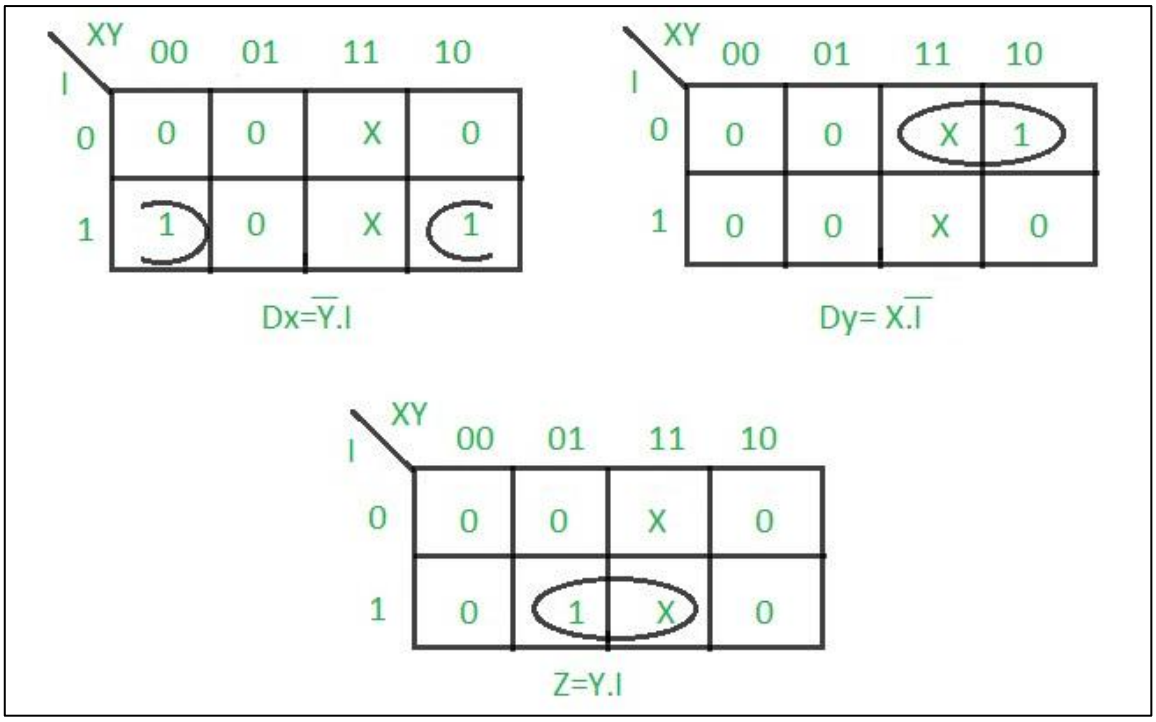


<u>Step 3: Make Present State/Next State table –</u>
We'll use D-Flip Flops for design purposes.

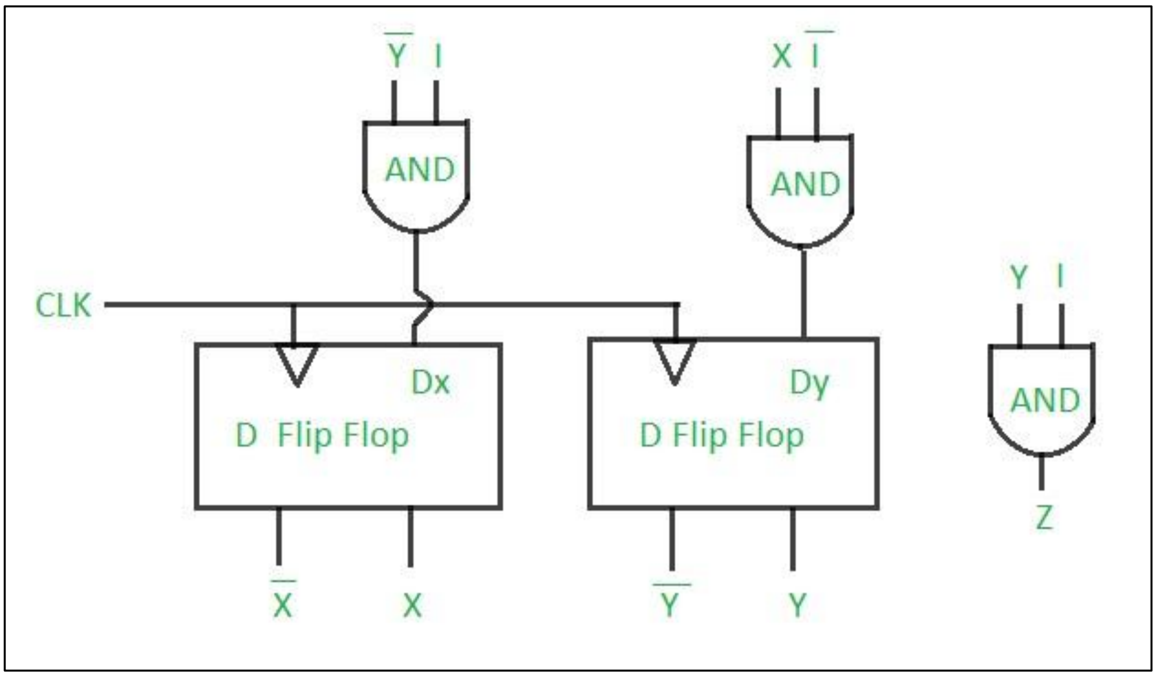


Present States		i/p	Next States		Flip Flop Excitations		O/P
X	Y		X'	Y'	Dx	Dy	
0	0	0	0	0	0	0	0
0	0	1	1	0	1	0	0
0	1	0	0	0	0	0	0
0	1	1	0	0	0	0	1
1	0	0	0	1	0	1	0
1	0	1	1	0	1	0	0
1	1	0	X	X	X	X	X
1	1	1	X	X	X	X	X

Step 4: Draw K-maps for Dx, Dy and output (Z) –



Step 5: Finally implement the circuit –



This is the final circuit for a Mealy 101 non-overlapping sequence detector.

Advantages of Sequence Detector

1. Pattern Detection: The Detectors distinguish high fidelity data streams and, hence, enhance the integrity of communication systems.
2. Flexibility: Their range runs from telecommunication to bioinformatics and hence supports great flexibility in design and development.
3. High Resolution: Advanced sequence detectors can differentiate between sequences in noisier environments too. Hence, they are dependable for any critical applications.
4. Integrability with other digital elements: It can easily be integrated with other digital elements. The circuits become more useful to the general system.

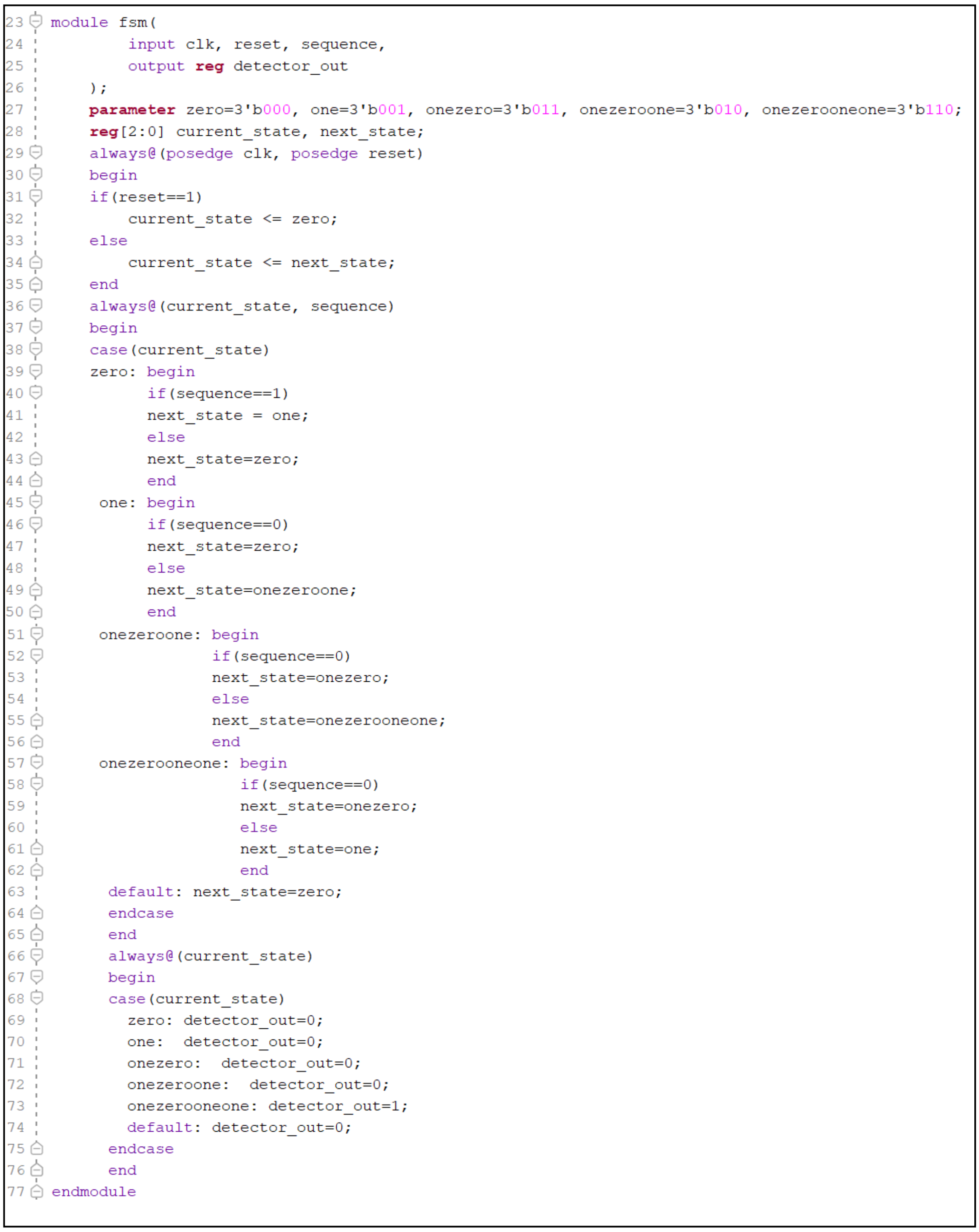
Disadvantages of Sequence Detector

1. Design Complexity: A sequence detector for sequences can sometimes consume more resources and hence might demand long techniques of design.
2. Latency: The implementation of the sequence detectors is also prone to latency effects during the detection of the sequence. This latency is more likely to be a problem in real-time systems.
3. Resource Intensive: Advanced sequence detectors are sometimes resource-intensive too; therefore, it makes them cost-inefficient.
4. Scalability Challenges: The more complex sequences tend to be, the further design and resource requirements push up can pose scalability challenges.

Applications of Sequence Detector

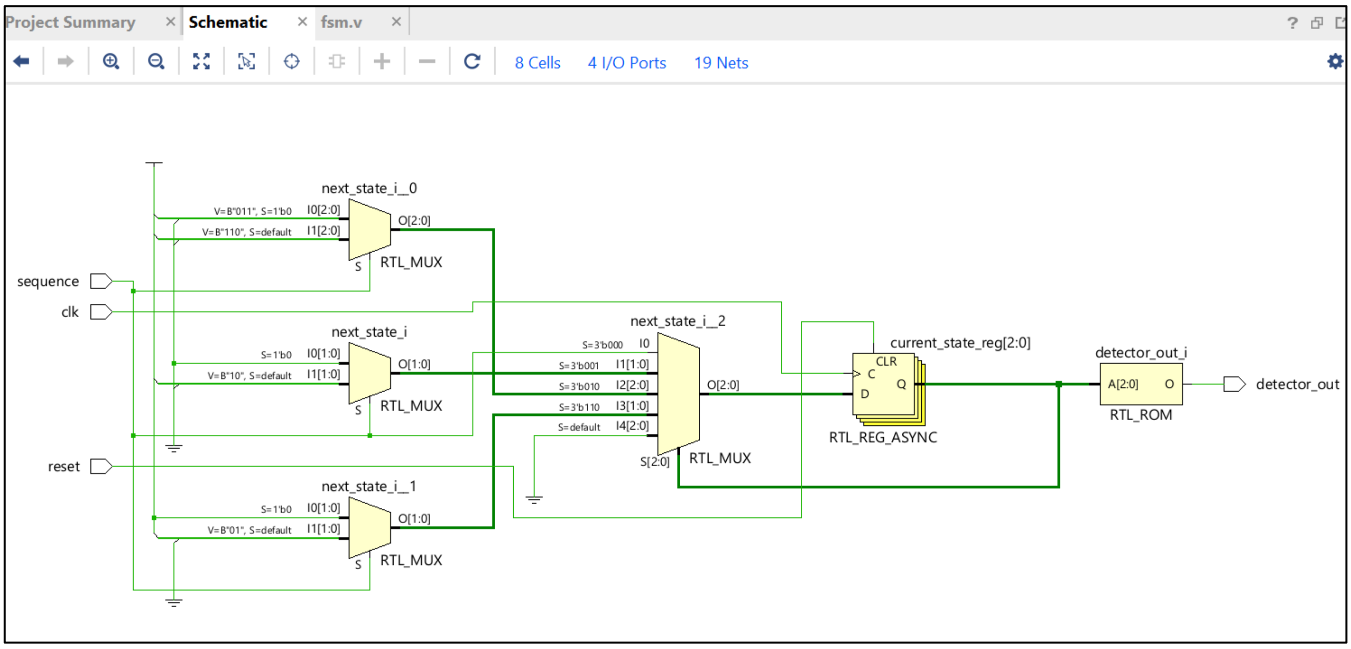
- Data Compression: It is used in algorithms that need pattern identification for specific sequences of data storage.
- Control Systems: It is applied in control systems that perform monitoring and decision-making based on patterns of the input signal observed.
- Bioinformatics: Applied to find specific nucleotide sequences in DNA or RNA for purposes of genetic analysis and study.
- Pattern recognition: Applied to a vast amount of applications from image and machine learning down to pattern recognition in datasets.
- Embedded systems: Embedded systems are Applied in microcontrollers as well as digital circuits with applications requiring control logic to identify sequences.

RTL Code:

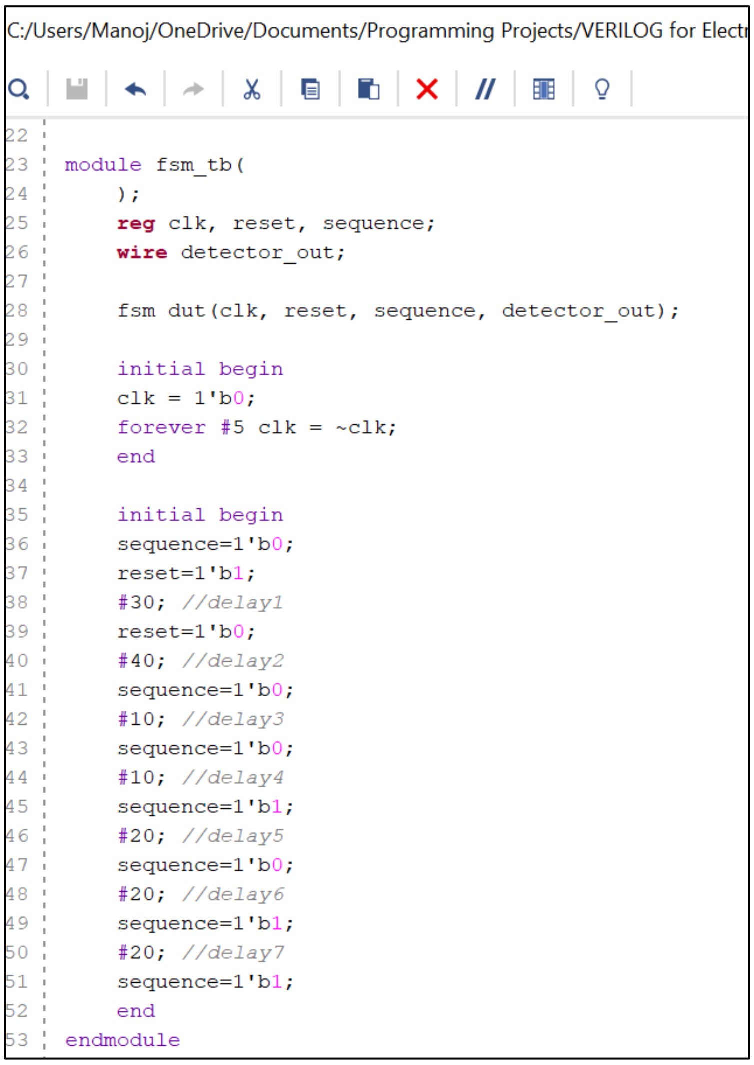


```
module fsm(
        input clk, reset, sequence,
        output reg detector_out
    );
    parameter zero=3'b000, one=3'b001, onezero=3'b011, onezeroone=3'b010, onezerooneone=3'b110;
    reg[2:0] current_state, next_state;
    always@(posedge clk, posedge reset)
    begin
    if(reset==1)
        current_state <= zero;
    else
        current_state <= next_state;
    end
    always@(current_state, sequence)
    begin
    case(current_state)
    zero: begin
          if(sequence==1)
          next_state = one;
          else
          next_state=zero;
          end
     one: begin
          if(sequence==0)
          next_state=zero;
          else
          next_state=onezeroone;
          end
     onezeroone: begin
                 if(sequence==0)
                 next_state=onezero;
                 else
                 next_state=onezerooneone;
                 end
     onezerooneone: begin
                    if(sequence==0)
                    next_state=onezero;
                    else
                    next_state=one;
                    end
      default: next_state=zero;
      endcase
      end
      always@(current_state)
      begin
      case(current_state)
        zero: detector_out=0;
        one:  detector_out=0;
        onezero:  detector_out=0;
        onezeroone:  detector_out=0;
        onezerooneone: detector_out=1;
        default: detector_out=0;
      endcase
      end
endmodule
```

Schematic diagram:



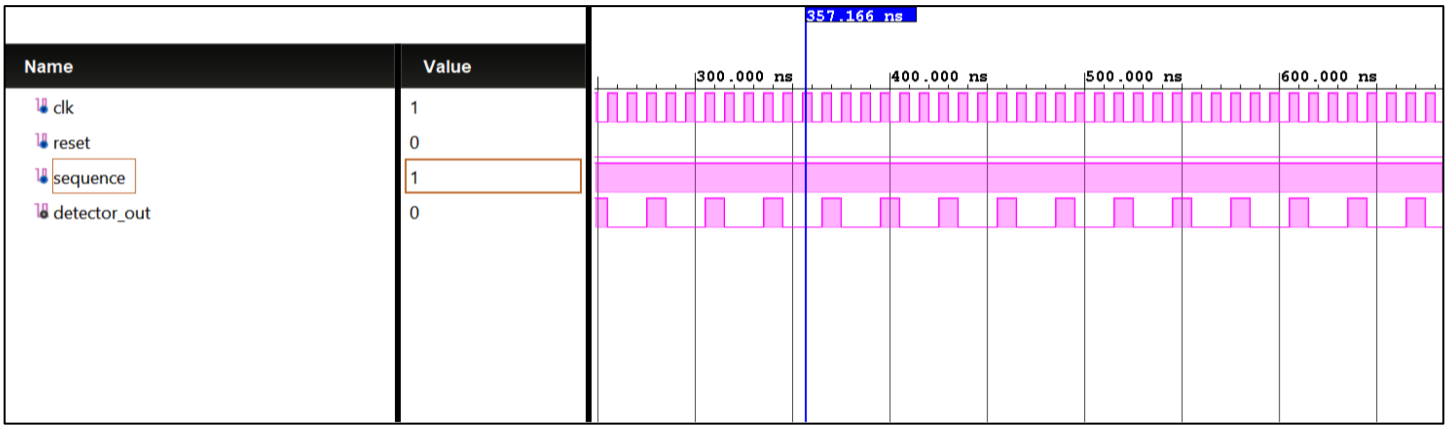
Testbench:



C:/Users/Manoj/OneDrive/Documents/Programming Projects/VERILOG for Electr

```verilog

module fsm_tb(
    );
    reg clk, reset, sequence;
    wire detector_out;

    fsm dut(clk, reset, sequence, detector_out);

    initial begin
    clk = 1'b0;
    forever #5 clk = ~clk;
    end

    initial begin
    sequence=1'b0;
    reset=1'b1;
    #30; //delay1
    reset=1'b0;
    #40; //delay2
    sequence=1'b0;
    #10; //delay3
    sequence=1'b0;
    #10; //delay4
    sequence=1'b1;
    #20; //delay5
    sequence=1'b0;
    #20; //delay6
    sequence=1'b1;
    #20; //delay7
    sequence=1'b1;
    end
endmodule
```

Wave Form:



Name	Value
clk	1
reset	0
sequence	1
detector_out	0

Conclusion:

A group of sequence detectors play a highly important role in many digital systems because they assist in determining the repeated patterns of input signals. Their applications range from telecommunications to bioinformatics and include research in the area of digital signal processing, reiterating the fact that the use of sequence detectors spans several disciplines. However, on the other hand, several advantages do exist, such as efficiency in pattern recognition and adaptability, but there are challenges, like design complexity and latency issues. Increasingly sophisticated and efficient detectors of sequence will further allow technology to reach ever greater functionality, and with its enhanced use in digital as well as in biological contexts.

Thus, FSM implementation aids for such sequence detector.

Module 4.1:

Verilog overview:

1. Data Types
   - Verilog provides several data types for modelling hardware.
   - Common types include:
     - `wire`: Represents a physical connection; continuously driven by an assign statement or module output.
     - `reg`: Holds a value in procedural blocks (e.g., `always` blocks); doesn't imply storage unless specified.
     - `integer`: Used for storing integer values in simulations, not synthesizable for hardware.
     - `time` and `realtime`: Used for simulation timing analysis, primarily non-synthesizable.

2. Constants
   - Fixed values used in design, represented by literal values or defined using `parameter`.
   - **Numeric constants**: Written with format `<size>'<base><value>`, like `8'hFF` (8-bit hexadecimal value).
   - **String constants**: Enclosed in double quotes, useful for display in simulations.

3. Parameters
   - Parameters define fixed values that can be used throughout the module and changed when the module is instantiated.
   - Defined with `parameter` keyword, making designs more flexible and configurable.
   - Example:

```
parameter WIDTH = 8;
```

4. Wires
   - Represents physical connections, carrying continuous signals between components.
   - Values on `wire` are driven by assignments or connected modules.
   - Used mainly for combinational logic.

Example:

```
wire result;
assign result = a & b;
```

5. Registers (reg)
   - Stores values within procedural blocks (`always`, `initial`).
   - Used for representing sequential elements (e.g., flip-flops).
   - Values in `reg` can change based on conditions in procedural statements, making them suitable for storing states.

Example:

```
reg q;
always @(posedge clk) begin
        q <= d;
end
```

Core features of Verilog:

1. Primitive Logic Gates: Built-in support for basic logic gates such as AND, OR, and NAND.

2. User-Defined Primitives (UDP) : Flexibility to create combinational or sequential logic primitives.

3. Switch-Level Modeling: Built-in primitives for switch-level modeling like PMOS and NMOS transistors.

4. Timing and Delays: Explicit constructs for specifying pin-to-pin delays, path delays, and timing checks.

5. Design Modeling Styles
- Behavioral Style: Uses procedural constructs to describe behavior.
- Dataflow Style: Utilizes continuous assignments to model data flows.
- Structural Style: Uses gate and module instantiations to describe design structure.
- Mixed Style: Combines elements of all three styles.

6. Data Types
- Net Data Type: Represents physical connections between structural elements.
- Register Data Type: Represents abstract data storage elements.

7. Hierarchical Design: Allows modeling of hierarchical designs using module instantiation, supporting arbitrary design sizes.

8. Standardization: Non-proprietary and IEEE standard, ensuring it is both human and machine-readable.

Module 4.2:

1. Half Adder

   Dataflow Model:
```
module half_adder_df(
        input A, B,
        output Sum, Carry
);
        assign Sum = A ^ B;
        assign Carry = A & B;
endmodule
```

   Behavioral Model:
```
module half_adder_beh(
        input A, B,
        output reg Sum, Carry
);
        always @(*) begin
        Sum = A ^ B;
        Carry = A & B;
        end
endmodule
```

Structural Model:

```
module half_adder_str(
        input A, B,
        output Sum, Carry
);
        xor(Sum, A, B);
        and(Carry, A, B);
endmodule
```

2. Full Adder

   Dataflow Model:

```
module full_adder_df(
        input A, B, Cin,
        output Sum, Cout
);
        assign Sum = A ^ B ^ Cin;
        assign Cout = (A & B) | (B & Cin) | (Cin & A);
endmodule
```

Behavioral Model:

```
module full_adder_beh(
        input A, B, Cin,
        output reg Sum, Cout
);
        always @(*) begin
        Sum = A ^ B ^ Cin;
        Cout = (A & B) | (B & Cin) | (Cin & A);
        end
endmodule
```

Structural Model:

```
module full_adder_str(
        input A, B, Cin,
        output Sum, Cout
);
        wire S1, C1, C2;
        half_adder_str ha1(A, B, S1, C1);
        half_adder_str ha2(S1, Cin, Sum, C2);
        or(Cout, C1, C2);
endmodule
```

3. Half Subtractor

   Dataflow Model:

```
module half_subtractor_df(
        input A, B,
        output Diff, Borrow
);
        assign Diff = A ^ B;
        assign Borrow = ~A & B;
endmodule
```

Behavioral Model:

```
module half_subtractor_beh(
        input A, B,
        output reg Diff, Borrow
);
        always @(*) begin
        Diff = A ^ B;
        Borrow = ~A & B;
        end
endmodule
```

Structural Model:

```
module half_subtractor_str(
        input A, B,
        output Diff, Borrow
);
        xor(Diff, A, B);
        and(Borrow, ~A, B);
endmodule
```

4. Full Subtractor

Dataflow Model:

```
module full_subtractor_df(
        input A, B, Bin,
        output Diff, Bout
);
        assign Diff = A ^ B ^ Bin;
        assign Bout = (~A & B) | ((~A | B) & Bin);
endmodule
```

Behavioral Model:

```
module full_subtractor_beh(
        input A, B, Bin,
        output reg Diff, Bout
);
        always @(*) begin
        Diff = A ^ B ^ Bin;
        Bout = (~A & B) | ((~A | B) & Bin);
        end
endmodule
```

Structural Model:

```
module full_subtractor_str(
        input A, B, Bin,
        output Diff, Bout
);
        wire D1, B1, B2;
        half_subtractor_str hs1(A, B, D1, B1);
        half_subtractor_str hs2(D1, Bin, Diff, B2);
        or(Bout, B1, B2);
endmodule
```

5. 4:1 Multiplexer

   Dataflow Model:

```
module mux4x1_df(
        input [3:0] D,
        input [1:0] S,
        output Y
);
        assign Y = (S == 2'b00) ? D[0] :
        (S == 2'b01) ? D[1] :
        (S == 2'b10) ? D[2] :
        D[3];
endmodule
```

   Behavioral Model:

```
module mux4x1_beh(
        input [3:0] D,
        input [1:0] S,
        output reg Y
);
        always @(*) begin
        case (S)
        2'b00: Y = D[0];
        2'b01: Y = D[1];
        2'b10: Y = D[2];
        2'b11: Y = D[3];
        endcase
        end
endmodule
```

   Structural Model:

```
module mux4x1_str(
        input [3:0] D,
        input [1:0] S,
        output Y
);
        wire nS0, nS1;
```

```
            wire Y0, Y1, Y2, Y3;

            not(nS0, S[0]);
            not(nS1, S[1]);

            and(Y0, D[0], nS1, nS0);
            and(Y1, D[1], nS1, S[0]);
            and(Y2, D[2], S[1], nS0);
            and(Y3, D[3], S[1], S[0]);

            or(Y, Y0, Y1, Y2, Y3);
endmodule
```

6. 1:4 Demultiplexer

Dataflow Model:

```
module demux1x4_df(
            input D,
            input [1:0] S,
            output [3:0] Y
);
            assign Y[0] = D & ~S[1] & ~S[0];
            assign Y[1] = D & ~S[1] & S[0];
            assign Y[2] = D & S[1] & ~S[0];
            assign Y[3] = D & S[1] & S[0];
endmodule
```

Behavioral Model:

```
module demux1x4_beh(
            input D,
            input [1:0] S,
            output reg [3:0] Y
);
            always @(*) begin
            Y = 4'b0000;
            case (S)
            2'b00: Y[0] = D;
            2'b01: Y[1] = D;
            2'b10: Y[2] = D;
            2'b11: Y[3] = D;
            endcase
            end
endmodule
```

Structural Model:

```
module demux1x4_str(
            input D,
            input [1:0] S,
            output [3:0] Y
);
        wire nS0, nS1;
        not(nS0, S[0]);
        not(nS1, S[1]);

        and(Y[0], D, nS1, nS0);
        and(Y[1], D, nS1, S[0]);
        and(Y[2], D, S[1], nS0);
        and(Y[3], D, S[1], S[0]);
endmodule
```

7. 2:4 Decoder

Dataflow Model:

```
module decoder2x4_df(
        input [1:0] A,
        output [3:0] Y
);
        assign Y[0] = ~A[1] & ~A[0];
        assign Y[1] = ~A[1] & A[0];
        assign Y[2] = A[1] & ~A[0];
        assign Y[3] = A[1] & A[0];
endmodule
```

Behavioral Model:

```
module decoder2x4_beh(
        input [1:0] A,
        output reg [3:0] Y
);
        always @(*) begin
        case (A)
        2'b00: Y = 4'b0001;
        2'b01: Y = 4'b0010;
        2'b10: Y = 4'b0100;
        2'b11: Y = 4'b1000;
        endcase
        end
endmodule
```

Structural Model:

```
module decoder2x4_str(
        input [1:0] A,
        output [3:0] Y
);
        wire nA0, nA1;
        not(nA0, A[0]);
        not(nA1, A[1]);

        and(Y[0], nA1, nA0);
        and(Y[1], nA1, A[0]);
        and(Y[2], A[1], nA0);
        and(Y[3], A[1], A[0]);
endmodule
```

8. 1-Bit Comparator

Dataflow Model:

```
module comparator1bit_df(
        input A, B,
        output A_eq_B, A_gt_B, A_lt_B
);
        assign A_eq_B = (A == B);
        assign A_gt_B = (A > B);
        assign

 A_lt_B = (A < B);
endmodule
```

Behavioral Model:

```
module comparator1bit_beh(
        input A, B,
        output reg A_eq_B, A_gt_B, A_lt_B
);
        always @(*) begin
        A_eq_B = (A == B);
        A_gt_B = (A > B);
        A_lt_B = (A < B);
        end
endmodule
```

Structural Model:

```
module comparator1bit_str(
        input A, B,
        output A_eq_B, A_gt_B, A_lt_B
);
        wire nA, nB;
        not(nA, A);
        not(nB, B);
        and(A_eq_B, ~A ^ ~B);
        and(A_gt_B, A, nB);
        and(A_lt_B, nA, B);
endmodule
```

Module 5.1:

1. D ff:

```
module d_flipflop(
        input D,        // Data input
        input clk,        // Clock input
        input rst,        // Reset input (active high)
        output reg Q   // Output
);

        always @(posedge clk or posedge rst) begin
        if (rst)
        Q <= 1'b0;       // Reset output to 0
        else
        Q <= D;          // Store the value of D on rising edge of clock
        end
endmodule
```

2. T ff:

```
module t_flipflop(
        input T,          // Toggle input
        input clk,        // Clock input
        input rst,        // Reset input (active high)
        output reg Q   // Output
);

        always @(posedge clk or posedge rst) begin
        if (rst)
        Q <= 1'b0;       // Reset output to 0
        else if (T)
        Q <= ~Q;         // Toggle output if T is high
        // If T is low, Q remains the same
        end
endmodule
```

3. Mod counter (Asynchronous mod 5 counter):

```
module mod5_counter (
        input clk,        // Clock signal
        input rst,        // Asynchronous reset (active high)
        output reg [2:0] Q // 3-bit output (since MOD-5 counts from 0 to 4)
);

        always @(posedge clk or posedge rst) begin
        if (rst) begin
        Q <= 3'b000;  // Reset the counter to 0
        end else begin
        if (Q == 3'b100)  // When Q reaches 4 (MOD-5), reset to 0
                Q <= 3'b000;
        else
                Q <= Q + 1'b1;  // Increment the counter
        end
        end
endmodule
```

For synchronous counter write only posedge clk in always condition

4. SISO

Behavioral Model:

```
module siso_beh(
        input clk, reset, serial_in,
        output reg serial_out
);
        reg [3:0] shift_reg;  // 4-bit shift register

        always @(posedge clk or posedge reset) begin
        if (reset)
        shift_reg <= 4'b0000;
        else begin
        shift_reg <= {shift_reg[2:0], serial_in};  // Shift left by 1, adding serial_in
        serial_out <= shift_reg[3];                // Output the MSB
        end
        end
endmodule
```

Structural Model:

```
module siso_str(
        input clk, reset, serial_in,
        output serial_out
);
        wire d0, d1, d2, d3;
        wire q0, q1, q2, q3;

        dff dff0(.clk(clk), .reset(reset), .d(serial_in), .q(q0));
        dff dff1(.clk(clk), .reset(reset), .d(q0), .q(q1));
        dff dff2(.clk(clk), .reset(reset), .d(q1), .q(q2));
        dff dff3(.clk(clk), .reset(reset), .d(q2), .q(q3));

        assign serial_out = q3;
endmodule
```

5. PIPO Shift Register (Parallel-In Parallel-Out)

Behavioral Model:

```
module pipo_beh(
        input clk, reset,
        input [3:0] parallel_in,
        output reg [3:0] parallel_out
);
        always @(posedge clk or posedge reset) begin
        if (reset)
        parallel_out <= 4'b0000;
        else
        parallel_out <= parallel_in;
        end
endmodule
```

Structural Model:

```
module pipo_str(
        input clk, reset,
        input [3:0] parallel_in,
        output [3:0] parallel_out
);
        dff dff0(.clk(clk), .reset(reset), .d(parallel_in[0]), .q(parallel_out[0]));
        dff dff1(.clk(clk), .reset(reset), .d(parallel_in[1]), .q(parallel_out[1]));
        dff dff2(.clk(clk), .reset(reset), .d(parallel_in[2]), .q(parallel_out[2]));
        dff dff3(.clk(clk), .reset(reset), .d(parallel_in[3]), .q(parallel_out[3]));
endmodule
```

Module 5.2:

1. Mealy overlapping 101:

```
module mealy_fsm_overlapping(
        input clk, reset, in,
        output reg out
);
        typedef enum reg [1:0] {IDLE, S1, S10} state_t;
        state_t current_state, next_state;

        always @(posedge clk or posedge reset) begin
        if (reset)
        current_state <= IDLE;
        else
        current_state <= next_state;
        end

        always @(*) begin
        out = 0;
        case (current_state)
        IDLE: begin
                next_state = in ? S1 : IDLE;
        end
        S1: begin
                next_state = in ? S1 : S10;
        end
        S10: begin
                if (in) begin
                next_state = S1; // Transition back to S1 for overlapping detection
                out = 1;         // Sequence 101 detected
                end else
```

```
            next_state = IDLE;
        end
        default: next_state = IDLE;
        endcase
        end
endmodule
```

2. Mealy non-overlapping 101:

```
module mealy_fsm_non_overlapping(
        input clk, reset, in,
        output reg out
);
        typedef enum reg [1:0] {IDLE, S1, S10} state_t;
        state_t current_state, next_state;

        always @(posedge clk or posedge reset) begin
        if (reset)
        current_state <= IDLE;
        else
        current_state <= next_state;
        end

        always @(*) begin
        out = 0;
        case (current_state)
        IDLE: begin
            next_state = in ? S1 : IDLE;
        end
        S1: begin
            next_state = in ? S1 : S10;
        end
        S10: begin
            if (in) begin
            next_state = IDLE; // Reset to IDLE for non-overlapping detection
            out = 1;           // Sequence 101 detected
            end else
            next_state = IDLE;
        end
        default: next_state = IDLE;
        endcase
        end
endmodule
```

3. Moore overlapping 101:

```
module moore_fsm_overlapping(
        input clk, reset, in,
        output reg out
);
```

```
        typedef enum reg [1:0] {IDLE, S1, S10, S101} state_t;
        state_t current_state, next_state;

        always @(posedge clk or posedge reset) begin
        if (reset)
        current_state <= IDLE;
        else
        current_state <= next_state;
        end

        always @(*) begin
        case (current_state)
        IDLE: next_state = in ? S1 : IDLE;
        S1: next_state = in ? S1 : S10;
        S10: next_state = in ? S101 : IDLE;
        S101: next_state = in ? S1 : S10; // Go to S1 for overlapping detection
        default: next_state = IDLE;
        endcase
        end

        always @(*) begin
        out = (current_state == S101) ? 1 : 0;  // Output high when in state S101
        end
endmodule
```

4. Moore non-overlapping 101:

```
module moore_fsm_non_overlapping(
        input clk, reset, in,
        output reg out
);
        typedef enum reg [1:0] {IDLE, S1, S10, S101} state_t;
        state_t current_state, next_state;

        always @(posedge clk or posedge reset) begin
        if (reset)
        current_state <= IDLE;
        else
        current_state <= next_state;
        end

        always @(*) begin
        case (current_state)
        IDLE: next_state = in ? S1 : IDLE;
        S1: next_state = in ? S1 : S10;
        S10: next_state = in ? S101 : IDLE;
        S101: next_state = IDLE; // Reset to IDLE for non-overlapping detection
        default: next_state = IDLE;
        endcase
        end

        always @(*) begin
        out = (current_state == S101) ? 1 : 0;  // Output high when in state S101
        end
endmodule
```