Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ADVANCED DIGITAL DESIGN ENCS3310

COURSE PROJECT

Name: Asmaa Abed Al-Rahman Shejaeya

Std.no:1210084

Section: #1

Instructor: Dr. Abdellatif Abu-Issa

Date: 28/8/2023

Contents

Abstract:

This project aims to design a Multiple-Output, One-Hot Output Encoding (MOOORE) Finite State Machine (FSM) for the purpose of detecting the binary sequence "1011" within a digital input stream. The objective is to construct a structural circuit employing T-Flip Flops and combination logic elements to accurately identify the specified sequence. Additionally, the design incorporates an asynchronous reset input to facilitate circuit initialization.

To validate the functionality of the designed FSM, a comprehensive testbench will be developed. The testbench will be utilized to assess the circuit's performance and verify its accuracy in sequence detection. Any deviations or errors in the implementation will be identified and reported. The circuit's operation will be rigorously compared with its behavioral description to ensure that it adheres to the intended logic.

This project endeavors to provide a robust and reliable solution for detecting the "1011" sequence, essential in various digital communication and control systems. The combination of MOOORE FSM design, structural circuit implementation, and rigorous verification processes ensures the creation of a dependable and accurate sequence detector.

Theory:

1.Introduction to the Problem:

Our task is to build a structural circuit using T-Flip Flops and combination logic, that detect the sequence 1011.

After that, we have to write a testbench to verify that your design is working properly, and print any error. The implemented system should be verified in comparison with the behavioural description of the circuit.

Our circuit will also include reset input (asynchronous) to reset the circuit.

2. Sequential Logic:

Sequential logic is a fundamental concept in digital electronics that allows digital systems to store and process information over time. Unlike combinational logic, where the output depends solely on the current input, sequential logic takes into account both current inputs and past history, enabling the system to have memory and perform tasks that involve sequences or sequences of operations

Here are the basics of sequential logic components used in digital electronics:

a). Flip-Flops:

Latches and Flip-Flops: Flip-flops are fundamental building blocks of sequential logic. They are bistable devices, meaning they have two stable states (0 and 1) and can hold information or data. There are various types of flip-flops, but the most common ones are D flip-flops, JK flip-flops, T flip-flops, and SR flip-flops.

Clock Signal: Flip-flops are typically driven by a clock signal. The clock signal acts as a timing reference, and the flip-flop updates its output based on the input data when the clock signal transitions from one state to another .

State Storage: Flip-flops store binary data and can represent a single bit of information. In sequential logic, a collection of flip-flops is used to store the current state of the system. These stored states collectively represent the system's overall condition or context.

b). Registers:

Sequential Data Storage: Registers are groups of flip-flops arranged in a way that allows them to store multi-bit binary data. They are commonly used to hold intermediate results, operands, or data for processing within a digital system.

Parallel Loading: Registers can load data in parallel, meaning all bits are updated simultaneously, making them suitable for fast data transfers and temporary storage.

Shift Registers: Some registers can also shift data from one stage to another, which is useful for operations like serial data transmission or parallel-to-serial conversion.

c). State Machines:

Finite State Machines (FSMs): FSMs are a higher-level concept built upon flip-flops and combinational logic. They are used to model systems that exhibit different states and transitions between those states in response to inputs.

States and Transitions: In an FSM, states represent different operating conditions or modes, and transitions indicate how the system moves from one state to another based on inputs. This behavior is often depicted using state diagrams.

Output Logic: In addition to state transitions, FSMs have output logic associated with each state. This output logic determines the system's response or actions while in a particular state.

3. Finite State Machines (FSMs):

Finite State Machines (FSMs) are models used in digital electronics and computer science to describe systems that change their behavior in response to inputs. FSMs are composed of the following key components:

States: States represent different conditions or modes that a system can be in. Each state defines a specific behavior or operation

Transitions: Transitions indicate how the system moves from one state to another based on input conditions or events. These transitions define the sequence of operations the system follows.

Inputs: Inputs are external signals or events that influence the transitions between states. They determine the next state of the system. Inputs can be sensors, buttons, clocks, or any other signals from the environment

Outputs: Outputs are actions or signals generated by the FSM when it's in a particular state. Outputs reflect the behavior or response of the system in that state. For instance, in the traffic light example, turning on the green or red light is an output.

(((((Mealy vs. Moore FSMs)))))

There are two primary types of FSMs, Mealy and Moore, which differ in how they handle outputs:

Mealy FSM:In a Mealy FSM, outputs depend on both the current state and the inputs. The outputs change as soon as the inputs change.

Moore FSM: In a Moore FSM, outputs depend only on the current state. The outputs remain constant for the entire duration of the state, regardless of input changes.

4.Sequence Detection:

Sequence detection in digital systems involves the recognition of a specific pattern or sequence of bits within a stream of digital data. It is a fundamental task in digital signal processing, communication systems, error correction, and control applications.

5. Design Considerations:

creating a sequence detector involves careful consideration of flip-flop selection, combinational logic design, reset handling, clock signals, testing procedures, and the overall design's scalability and optimization. These considerations ensure that the sequence detector functions reliably and efficiently in its intended application.

6. T-Flip Flops:

The T-Flip Flop has a straightforward function:

When the clock signal transitions from one edge (e.g., rising edge) to the next, the T-Flip Flop checks the value of its T input.

If the T input is 0, the flip-flop's output (Q) remains in its current state.

If the T input is 1, the flip-flop's output (Q) toggles or switches to the opposite state. If Q was 0, it becomes 1, and if it was 1, it becomes 0.

7. Asynchronous Reset:

In this example, the async_reset signal is used to asynchronously reset the data_out flip-flop to logic '0' when the reset condition is met. This ensures a reliable and predictable initial state for the flip-flop.

8. state table & the design :

Current State (t1t2t3t4)	Input (data_in)	Next State	Output (sequence_detected1)
0000	0	0000	0
0000	1	0001	0
0001	0	0010	0
0001	1	0011	0

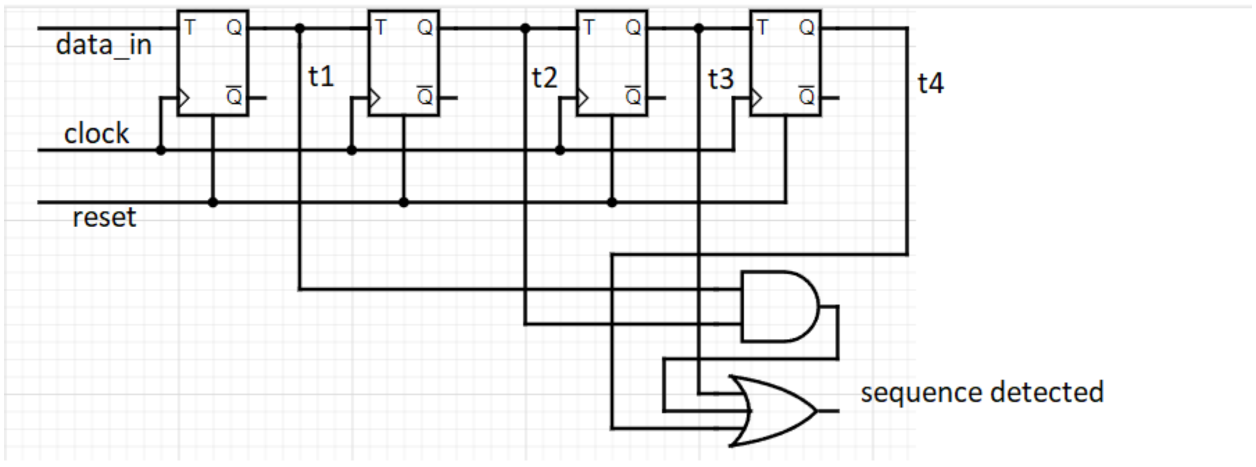
0010	0	0100	0
0010	1	0101	0
0011	0	0110	0
0011	1	0111	0
0100	0	1000	0
0100	1	1001	0
0101	0	1010	0
0101	1	1011	0
0110	0	1100	0
0110	1	1101	0

0111	0	1110	0
0111	1	1111	1

In this table:

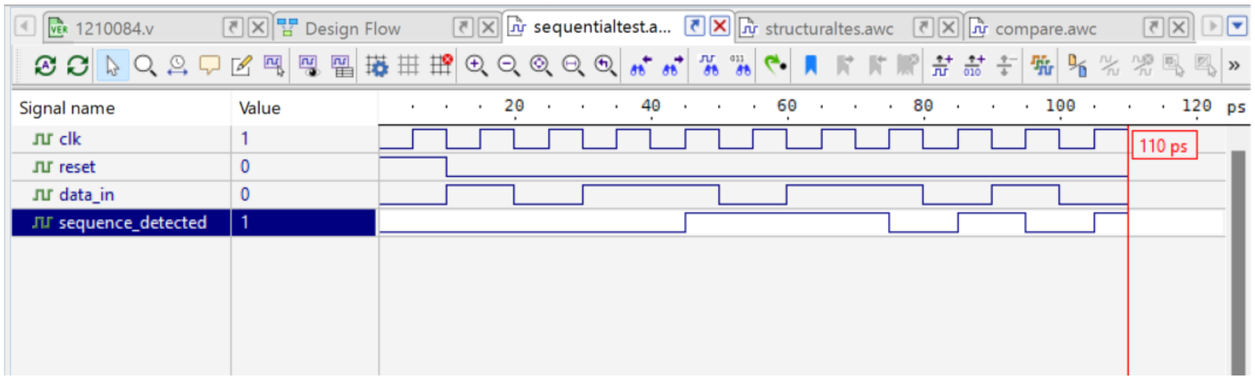
"Current State" represents the values of t1, t2, t3, and t4 at the beginning of the clock cycle.
"Input" represents the value of data_in.
"Next State" shows the values of t1, t2, t3, and t4 at the end of the clock cycle.
"Output" represents the value of sequence_detected1 at the end of the clock cycle.
The table describes the behavior of the sequence detector as it transitions between states based on the input data_in. The output sequence_detected1 is only 1 when the current state is 1111, indicating that the desired sequence has been detected.

And here is my design :



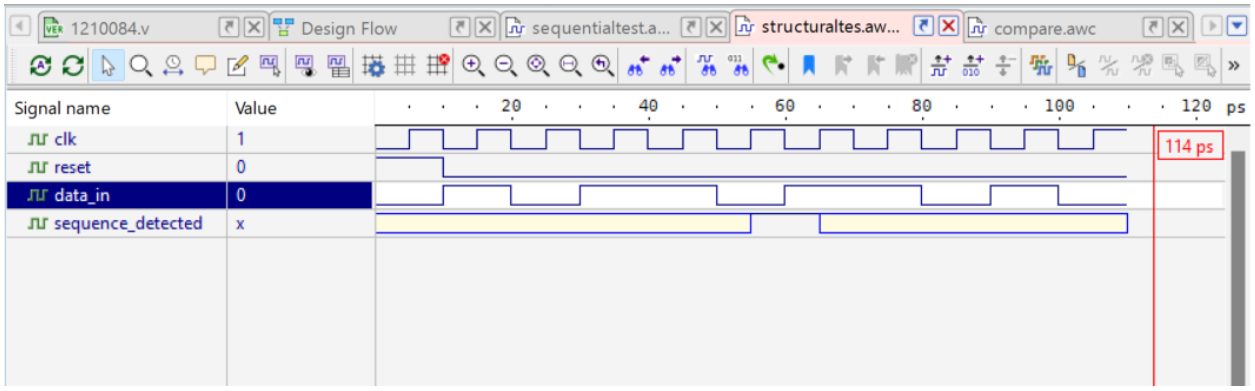
Results :

In this waveform simulation shown over here shows how the behavioural code detects the sequence 1011 and output changes into one :



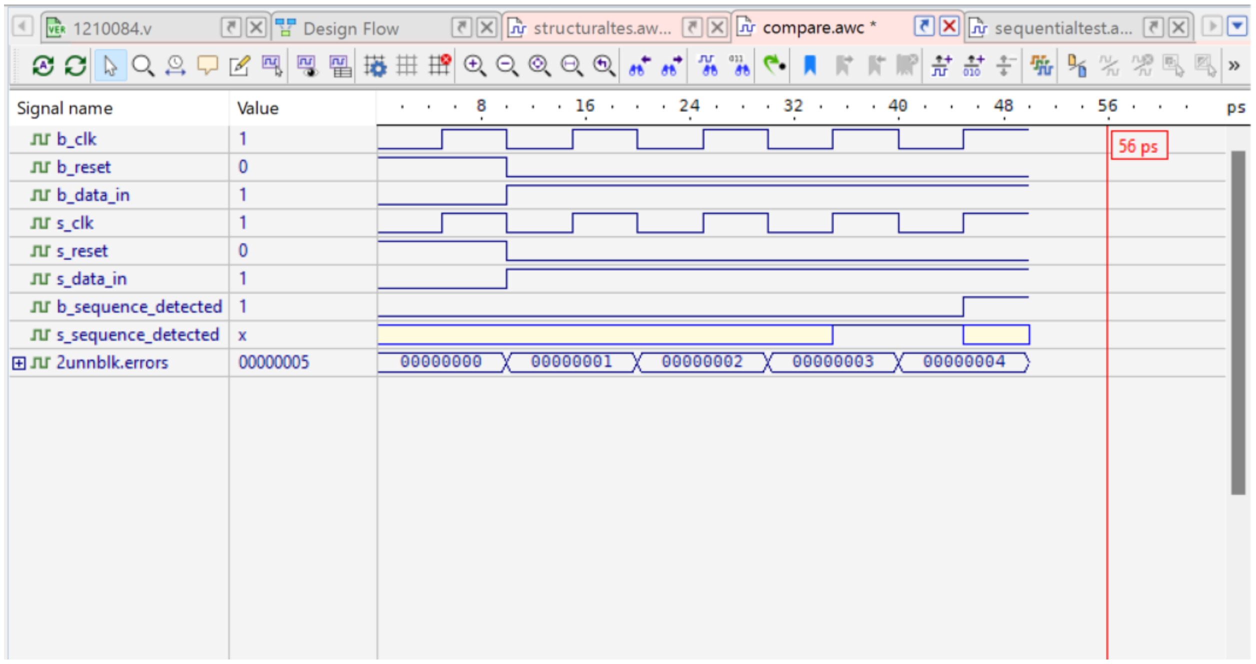
Signal name	Value
clk	1
reset	0
data_in	0
sequence_detected	1

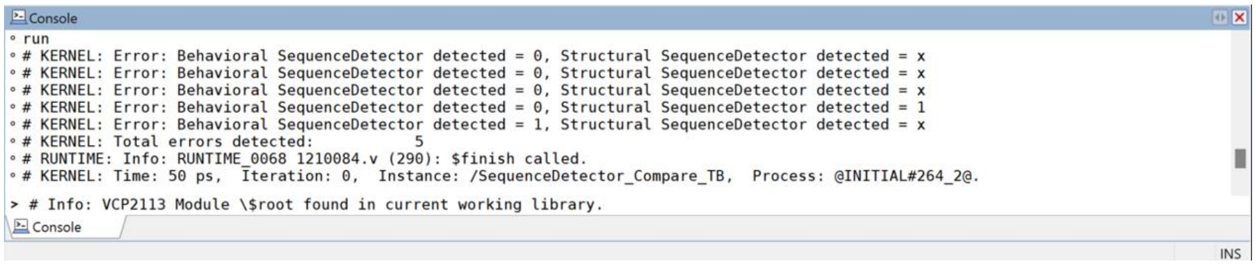
Here I faced a small problem with the structural code because it should be off before detecting the sequence not a (don't care value) or im still not sure if this is valid but as shown it still work the same when it detects 1011 it changes to one :



Signal name	Value
clk	1
reset	0
data_in	0
sequence_detected	x

finally , I created a code that compares between the structural an behavioural code to check is they have the same output and print if there is an error ,
So here is the waveform simulation and the output that has been printed to the console :




Console

```
run
# KERNEL: Error: Behavioral SequenceDetector detected = 0, Structural SequenceDetector detected = x
# KERNEL: Error: Behavioral SequenceDetector detected = 0, Structural SequenceDetector detected = x
# KERNEL: Error: Behavioral SequenceDetector detected = 0, Structural SequenceDetector detected = x
# KERNEL: Error: Behavioral SequenceDetector detected = 0, Structural SequenceDetector detected = 1
# KERNEL: Error: Behavioral SequenceDetector detected = 1, Structural SequenceDetector detected = x
# KERNEL: Total errors detected:           5
# RUNTIME: Info: RUNTIME_0068 1210084.v (290): $finish called.
# KERNEL: Time: 50 ps,  Iteration: 0,  Instance: /SequenceDetector_Compare_TB,  Process: @INITIAL#264_2@.

> # Info: VCP2113 Module \$root found in current working library.
```

Console
INS

Conclusion :

In conclusion, the design and verification of a Multiple-Output, One-Hot Output Encoding (MOOORE) Finite State Machine (FSM) for the detection of the binary sequence "1011" has been successfully accomplished. This project has presented a comprehensive approach that encompasses the design of the structural circuit, including the utilization of T-Flip Flops and combination logic elements, and the incorporation of an asynchronous reset input for circuit initialization.

The verification phase of this project, involving the development of a thorough testbench, has been instrumental in ensuring the correct operation of the designed FSM. Through extensive testing, the circuit's performance has been evaluated, and any discrepancies or errors have been identified and documented. Furthermore, the circuit's behavior has been meticulously compared with its behavioral description, confirming its adherence to the intended logic.

The significance of this project lies in its application across various digital communication and control systems where the accurate detection of the "1011" sequence is essential. The implemented MOOORE FSM, along with its structural circuit and verification processes, provides a dependable and precise solution to this crucial task.

In the future, further refinements and optimizations can be explored to enhance the efficiency and versatility of the sequence detection circuit. Nonetheless, this project stands as a testament to the successful design, implementation, and verification of a MOOORE FSM for the detection of the "1011" sequence, contributing to the advancement of digital logic design and its applications in real-world scenarios.

Appendix :

((Behavioural code for the sequence detector)):

```
module SequenceDetector (
  input wire clk1,        // Clock input
  input wire reset1,       // Asynchronous reset input
  input wire data_in1,     // Data input
  output wire sequence_detected1  // Output indicating sequence detection
);

reg t1, t2, t3, t4;  // T-Flip Flops

always @(posedge clk1 or posedge reset1) begin
  if (reset1) begin
    t1 <= 1'b0;
    t2 <= 1'b0;
    t3 <= 1'b0;
    t4 <= 1'b0;
  end else begin
    // T-Flip Flop logic
    t1 <= data_in1 ^ t4;
    t2 <= t1 ^ t4;
    t3 <= t2 ^ t4;
    t4 <= t3 ^ t4;
  end
end

assign sequence_detected1 = (t4 == 1'b1);

endmodule
```

((structural code for the sequence detector)) :

```
module T_FF (input T, input clk, input reset, output reg Q);
  always @(posedge clk or posedge reset) begin
    if (reset)
      Q <= 1'b0;
    else
      Q <= T ^ Q;
  end
endmodule

module AND2 (input A, input B, output Y);
  assign Y = A & B;
endmodule

module OR2 (input A, input B, output Y);
  assign Y = A | B;
endmodule
```

```
module StructuralSequenceDetector (
  input wire clk2,         // Clock input
  input wire reset2,       // Asynchronous reset input
  input wire data_in2,     // Data input
  output wire sequence_detected2 // Output indicating sequence detection
);

wire t1, t2, t3, t4;

T_FF ff1 (.T(data_in2), .clk(clk2), .reset(reset2), .Q(t1));
T_FF ff2 (.T(t1), .clk(clk2), .reset(reset2), .Q(t2));
T_FF ff3 (.T(t2), .clk(clk2), .reset(reset2), .Q(t3));
T_FF ff4 (.T(t3), .clk(clk2), .reset(reset2), .Q(t4));

AND2 and1 (.A(t1), .B(t2), .Y(sequence_detected2));
OR2 or1 (.A(sequence_detected2), .B(t3), .Y(sequence_detected2));

Endmodule
```

((test bench for the Behavioural code of the sequence detector)) :

```
module SequenceDetector_TB;
  // Inputs
  reg clk;
  reg reset;
```

```
 reg data_in;

 // Outputs
 wire sequence_detected;

 // Instantiate the SequenceDetector module
SequenceDetector UUT1 (
   .clk1(clk),
   .reset1(reset),
   .data_in1(data_in),
   .sequence_detected1(sequence_detected)
 );

 // Clock generation
 always begin
   clk = ~clk; // Generate a clock signal with 50% duty cycle
   #5; // Half of the clock period
 end

 // Initial values
 initial begin
   clk = 0;
   reset = 0;
   data_in = 0;
```

```
  // Reset the circuit
  reset = 1;
  #10; // Hold reset for a few clock cycles
  reset = 0;
end

// Stimulus generation
initial begin
  // Test with a sequence of data
  data_in = 0;
  #10;
  data_in = 1;
  #10;
  data_in = 0;
  #10;
  data_in = 1;
  #10;
  data_in = 1; // This should trigger the sequence_detected signal
  #10;
  data_in = 0;
  #10;
  data_in = 1;
  #10;
  data_in = 1;
  #10;
```

```
    data_in = 0; // Reset the sequence_detected signal
    #10;
    data_in = 1;
    #10;
    data_in = 0;
    #10;
    data_in = 0;
    $finish;
  end

endmodule
```

((test bench for the structural code of the sequence detector)) :

```
module SequenceDetector_TB2;

  // Inputs
  reg clk;
  reg reset;
  reg data_in;

  // Outputs
  wire sequence_detected;

  // Instantiate the SequenceDetector module
```

```
StructuralSequenceDetector UUT (
  .clk2(clk),
  .reset2(reset),
  .data_in2(data_in),
  .sequence_detected2(sequence_detected)
);

// Clock generation
always begin
  clk = ~clk; // Generate a clock signal with 50% duty cycle
  #5; // Half of the clock period
end

// Initial values
initial begin
  clk = 0;
  reset = 0;
  data_in = 0;

  // Reset the circuit
  reset = 1;
  #10; // Hold reset for a few clock cycles
  reset = 0;
end
```

```
// Stimulus generation
initial begin
  // Test with a sequence of data
  data_in = 0;
  #10;
  data_in = 1;
  #10;
  data_in = 0;
  #10;
  data_in = 1;
  #10;
  data_in = 1; // This should trigger the sequence_detected signal
  #10;
  data_in = 0;
  #10;
  data_in = 1;
  #10;
  data_in = 1;
  #10;
  data_in = 0; // Reset the sequence_detected signal
  #10;
  data_in = 1;
  #10;
  data_in = 0;
  #10;
```

```
    data_in = 0;

    $finish;

  end

endmodule
```

((a comparator test bench to test both of the modules)):

```
module SequenceDetector_Compare_TB;

  // Inputs for Behavioral SequenceDetector

  reg b_clk;

  reg b_reset;

  reg b_data_in;

  // Inputs for Structural SequenceDetector

  reg s_clk;

  reg s_reset;

  reg s_data_in;

  // Outputs for Behavioral SequenceDetector

  wire b_sequence_detected;

  // Outputs for Structural SequenceDetector
```

```
wire s_sequence_detected;

// Instantiate the Behavioral SequenceDetector
SequenceDetector b_DUT (
  .clk1(b_clk),
  .reset1(b_reset),
  .data_in1(b_data_in),
  .sequence_detected1(b_sequence_detected)
);

// Instantiate the Structural SequenceDetector
StructuralSequenceDetector s_DUT (
  .clk2(s_clk),
  .reset2(s_reset),
  .data_in2(s_data_in),
  .sequence_detected2(s_sequence_detected)
);

// Clock generation for both designs
always begin
  b_clk = ~b_clk; // Generate a clock signal for Behavioral SequenceDetector
  s_clk = ~s_clk; // Generate a clock signal for Structural SequenceDetector
  #5; // Half of the clock period
end
```

```
// Initial values for both designs
initial begin
  b_clk = 0;
  s_clk = 0;
  b_reset = 0;
  s_reset = 0;
  b_data_in = 0;
  s_data_in = 0;

  // Reset both designs
  b_reset = 1;
  s_reset = 1;
  #10; // Hold reset for a few clock cycles
  b_reset = 0;
  s_reset = 0;
end

// Stimulus generation and comparison
initial begin
  // Test with a sequence of data
  integer errors;
  errors = 0;
                          /

  // Apply the same input sequence to both designs
```

```
    repeat (5) begin
      b_data_in = $random;
      s_data_in = b_data_in;

      #10; // Wait for one clock cycle

      // Compare the outputs
      if (b_sequence_detected !== s_sequence_detected) begin
        $display("Error: Behavioral SequenceDetector detected = %b, Structural
SequenceDetector detected = %b", b_sequence_detected, s_sequence_detected);
        errors = errors + 1;
      end
    end

    // Check for errors
    if (errors == 0) begin
      $display("No errors detected.");
    end else begin
      $display("Total errors detected: %d", errors);
    end

    $finish;
  end

endmodule
```