Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ENCS 331 Advanced Digital Course Project

# MOOORE FSM

Prepared by:

Name: Robert Bannoura Number #: 1210574

Instructor: Dr. Abdallatif Abuissa

Date: 2/9/2023

Sec#: 1

## Introduction:

Finite State Machines (FSMs) are important for digital circuit design, helping us control sequential logic systems. So, here in this report, we're diving into creating a Moore Finite State Machine (FSM). Our mission to create a sequential circuit to spot a specific binary sequence which is '1011.' Now, you might think, 'What's the big deal?' Well, it turns out this little task has big implications, from improving data transmission and communication to catching errors and even recognizing patterns in data.

We'll start by crafting a circuit that uses T-Flip Flops module and a of logic that is intended to spot the '1011' sequence in all sorts of input data streams. Once that's in place, we want to learn how to design the circuit so we use the Moore state diagram to get the design and dive into the nitty-gritty of validation and verification. Our mission here is crystal clear: make sure our circuit behaves just as we want it to. To get the job done, we're putting together a thorough testbench. It's like our circuit's personal obstacle course, where we'll throw all sorts of input situations at it. We want to catch any slip-ups and make darn sure it's working like a charm. And just to keep things interesting, we're tossing in an asynchronous reset feature. That way, we can swiftly get ready for the next '1011' sequence that comes our way.

## Table Of Contents:

## Contents

## Table Of Figures:

## Table of Tables:

## Theory:

### Cracking the Sequence Detector Code

In learning of digital circuits concepts, Finite State Machines (FSMs) is vital in designing sequential logic. Our mission? Crafting a Moore FSM to recognize the '1011' binary sequence. But why does it matter? Well, consider data transmission or pattern recognition – having a system that spots specific sequences in a data stream is pure gold.

Our toolkit includes T Flip-Flops and Active HDL as software. T Flip-Flops are like data holders, while logic operations help us decide what to do with that data. The trick? Crafting state transitions that ensure our FSM detects '1011' amidst a sea of data. It's like training a machine to spot a secret code in a noisy room – complex but fascinating.

### Validation and the Trusty Test Bench

Building is one thing; making sure it works is the real challenge. Our test bench is the quality control center. We throw a barrage of data at our FSM, testing its '1011' detection skills under various conditions. It's not just about '1011' – we want it to handle real-world messiness. So, here we are, college students learning the art of turning theory into practical solutions, and that's the theory behind our project – a mix of logic, creativity, and the pursuit of technical excellence."

## Procedure:

first, we want to build a project to spot the '1011' binary sequence. Our system is like a puzzle with T Flip-Flops and clever logic pieces that fit together just right, no matter what kind of data it's checking. To build it,; I started by considering the states that the machine will be in before the sequence is detected which are stated at the table below, then plan how it would change states and recognize '1011' exactly. It's like crafting a well-tuned engine for your car – it needs to work perfectly to get you where you want to go.

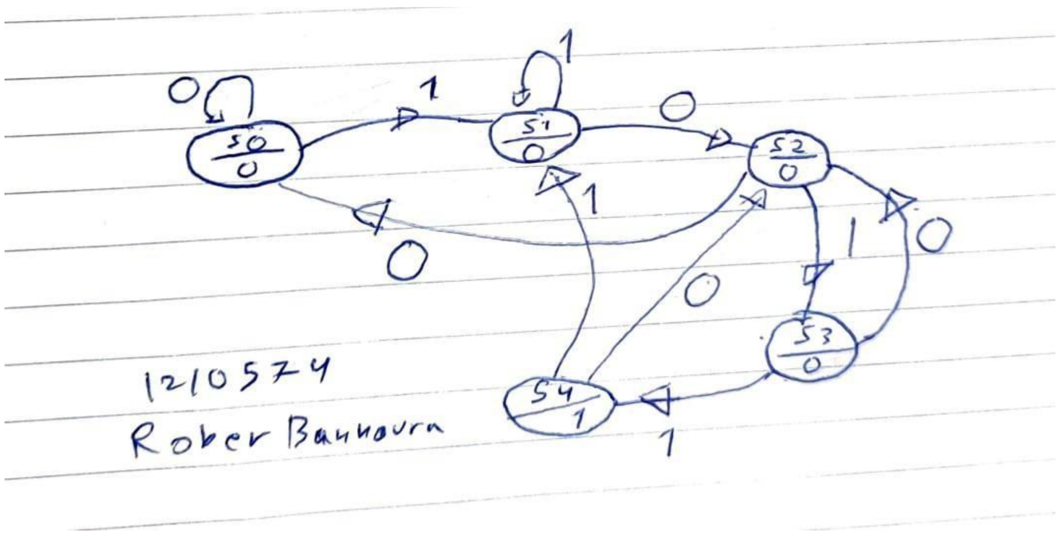


*Figure 1: Moore State Diagram(To get the sequence)*

*Table 1: State Definition*

| State | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|---|---|---|---|---|---|---|---|---|
| In Binary | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

| STATE | INPUT | NEXT STATE | OUTPUT |
|---|---|---|---|
| S0 | 0,1 | S0,S1 | 0,0 |
| S1 | 0,1 | S2,S3 | 0,0 |
| S2 | 0,1 | S4,S5 | 0,0 |
| S3 | 0,1 | S6,S7 | 0,0 |
| S4 | 0,1 | S0,S1 | 0,0 |
| S5 | 0,1 | S2,S3 | 0.1 |
| S6 | 0,1 | S4,S5 | 0,0 |
| S7 | 0,1 | S6,S7 | 0,0 |

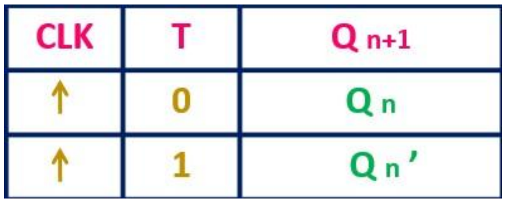
*Table 2: State Table Translation*

As we can notice the state Diagram has less states than the state table, this is because it was minimized because of the next states that just keeps on repeating, and the machine takes inputs bit by bit till it reaches 4 bits, and if the sequence “1011” appears the output will turn to 1, else there will be no change ‘0’.

And then started by designing T-flipflop module, and the circuit module based on the Moore state diagram and a test bench. It's like our system's personal playground, where we test it with all sorts of data to make sure it's sharp and error-free. Whether the data follows the '1011' pattern or not, our system can handle it. The design and testing parts of this project go hand in hand. We're sharing our journey from idea to a reliable sequence detector, mixing creativity with precision every step of the way.

## T-flip flop module:

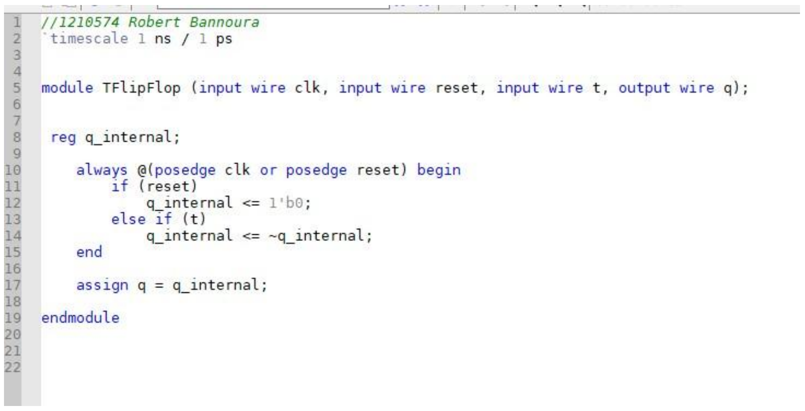
T-flip flop module works based on its truth table, which compliments the input as follows:



| CLK | T | $Q_{n+1}$ |
|---|---|---|
| ↑ | 0 | $Q_n$ |
| ↑ | 1 | $Q_n'$ |

*Figure 2: T-flipflop truth table*

Thus the Module in Verilog is as follows:

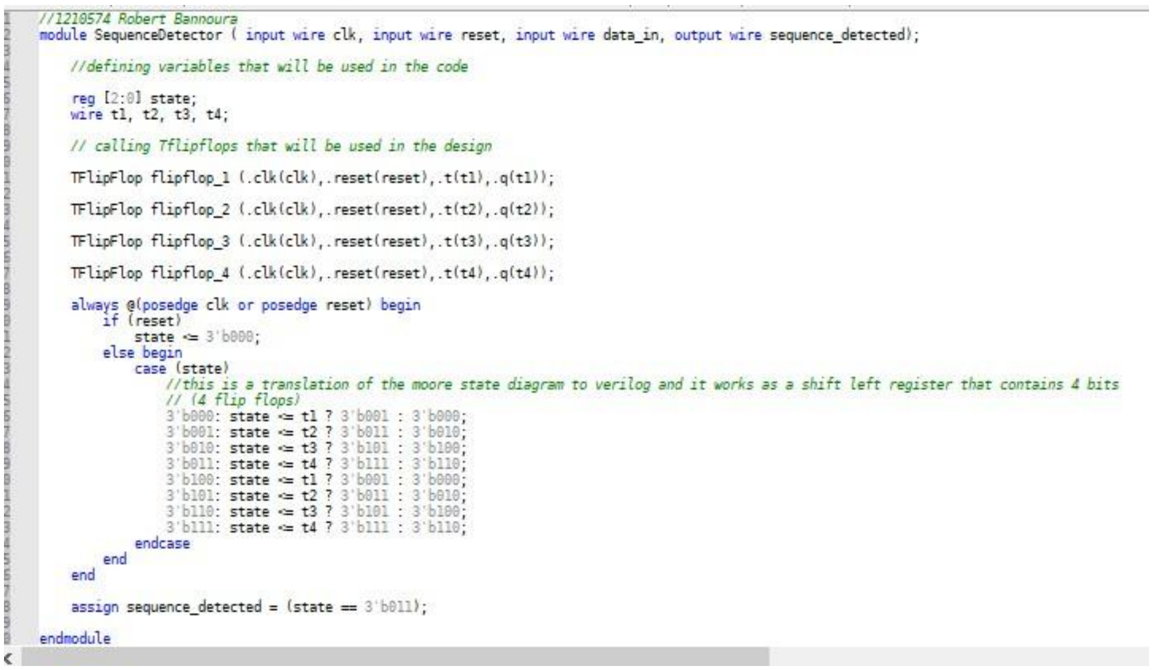


```
//1210574 Robert Bannoura
`timescale 1 ns / 1 ps


module TFlipFlop (input wire clk, input wire reset, input wire t, output wire q);


 reg q_internal;

    always @(posedge clk or posedge reset) begin
        if (reset)
            q_internal <= 1'b0;
        else if (t)
            q_internal <= ~q_internal;
    end

    assign q = q_internal;

endmodule
```

*Figure 3: T-flipflop Module*

## Circuit Design:

The Circuit Design is based on the Moore state Diagram (the minimized version), which uses 4 t flip flops as a memory element to check for the sequence, and it works as a 4 bit shift left register, here is the module in Verilog:



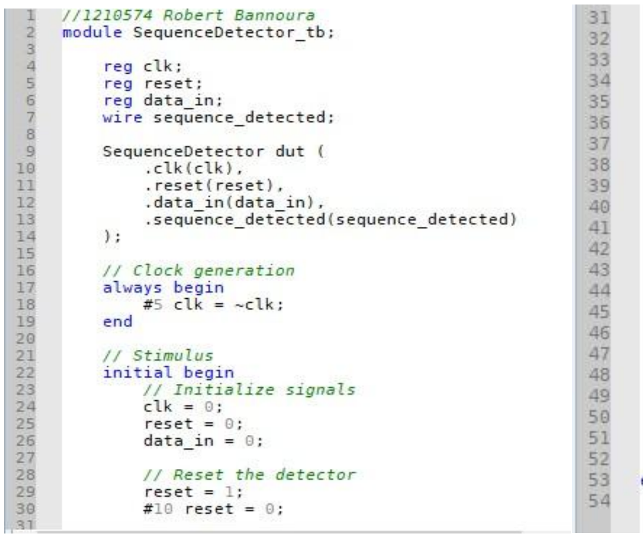
```verilog
//1210574 Robert Bannoura
module SequenceDetector ( input wire clk, input wire reset, input wire data_in, output wire sequence_detected);

    //defining variables that will be used in the code

    reg [2:0] state;
    wire t1, t2, t3, t4;

    // calling Tflipflops that will be used in the design

    TFlipFlop flipflop_1 (.clk(clk),.reset(reset),.t(t1),.q(t1));

    TFlipFlop flipflop_2 (.clk(clk),.reset(reset),.t(t2),.q(t2));

    TFlipFlop flipflop_3 (.clk(clk),.reset(reset),.t(t3),.q(t3));

    TFlipFlop flipflop_4 (.clk(clk),.reset(reset),.t(t4),.q(t4));

    always @(posedge clk or posedge reset) begin
        if (reset)
            state <= 3'b000;
        else begin
            case (state)
                //this is a translation of the moore state diagram to verilog and it works as a shift left register that contains 4 bits
                // (4 flip flops)
                3'b000: state <= t1 ? 3'b001 : 3'b000;
                3'b001: state <= t2 ? 3'b011 : 3'b010;
                3'b010: state <= t3 ? 3'b101 : 3'b100;
                3'b011: state <= t4 ? 3'b111 : 3'b110;
                3'b100: state <= t1 ? 3'b001 : 3'b000;
                3'b101: state <= t2 ? 3'b011 : 3'b010;
                3'b110: state <= t3 ? 3'b101 : 3'b100;
                3'b111: state <= t4 ? 3'b111 : 3'b110;
            endcase
        end
    end

    assign sequence_detected = (state == 3'b011);

endmodule
```

*Figure 4: Circuit Module*

## Test Bench:

The Test bench module Is designed to test if the circuit works as wanted structurally, and I put some test cases to verify that circuit detects the sequence 1011 if found.

Here is the module:



```verilog
//1210574 Robert Bannoura
module SequenceDetector_tb;

    reg clk;
    reg reset;
    reg data_in;
    wire sequence_detected;

    SequenceDetector dut (
        .clk(clk),
        .reset(reset),
        .data_in(data_in),
        .sequence_detected(sequence_detected)
    );

    // Clock generation
    always begin
        #5 clk = ~clk;
    end

    // Stimulus
    initial begin
        // Initialize signals
        clk = 0;
        reset = 0;
        data_in = 0;

        // Reset the detector
        reset = 1;
        #10 reset = 0;

        // Test sequence: 1011
        data_in = 1; #10 data_in = 0;
        data_in = 0; #10 data_in = 1;

        data_in = 1; #10 data_in = 0;
        data_in = 0; #10 data_in = 1;

        data_in = 1; #10 data_in = 0;
        data_in = 1; #10 data_in = 0;

        data_in = 1; #10 data_in = 0;
        data_in = 1; #10 data_in = 0;
        data_in = 0; #10 data_in = 1;


        data_in = 1; #10 data_in = 0;

        // Finish simulation
        $finish;
    end

endmodule
```

*Figure 5: Test Bench for Circuit*

## Waveform for Design:

After creating a Test Bench we want to check if the Results of the testing came back as wanted, and that's where the waveforms comes in handy:

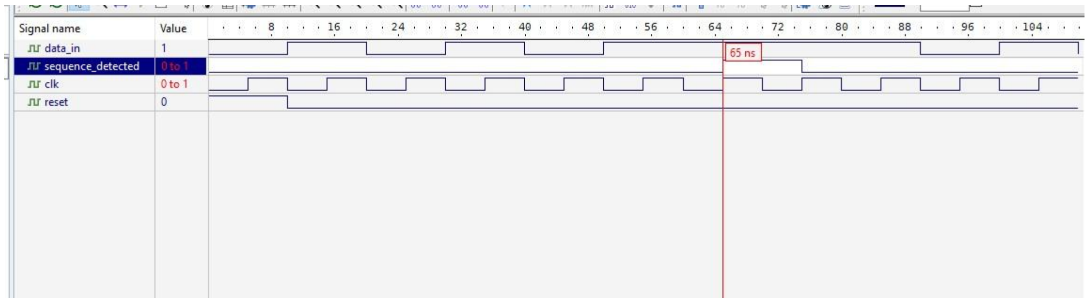


*Figure 6: Waveforms of the Design*

As shown above the "sequence_detected" variable turns to 1, after the data in is 1,0,1,1 else it goes back to 0.

## Conclusion:

In conclusion, our endeavor to construct a '1011' sequence detector has been an enlightening voyage of discovery. We've skillfully melded logical reasoning with technical proficiency to bring this system to fruition. Visualize it as a complex puzzle where every piece needed to seamlessly slot into place. During our testing phase, our system demonstrated unwavering reliability, successfully identifying the '1011' sequence across various test scenarios, underscoring its efficacy.

This project has served as a practical platform for us to apply the theoretical knowledge we've garnered in our academic journey. It represents the process of translating theory into tangible, functional technology. We take immense pride in our achievement and eagerly anticipate future ventures where we can harness these acquired skills to address even more intricate challenges