CND 111: Introduction to Digital Design

Assignment #: 6

Section #: 16

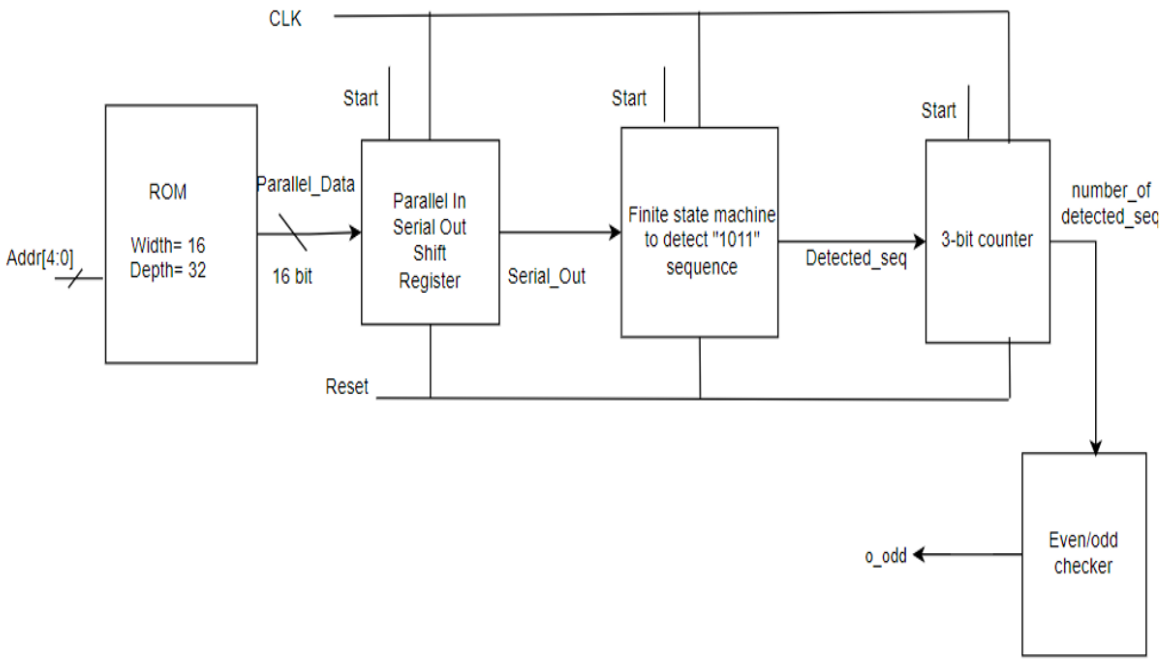
Submitted by:

Student Name	ID
Aya Ahmed Abdelrahman	23010284
Karim Mahmoud Kamal	V23010174
Tarek salah abdalhafeez	V23010337

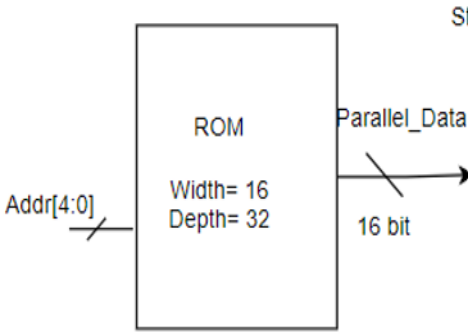
Submitted to TA: Mohamed Elshafey

Date: 3/10/2023

➢ Assignment 6



ROM (read only memory) design



E:/AUC assignments/Digital/Karim_LAB6/Karim_LAB6/shifter.v (/Top/Rom_Instance) - Default

```
Ln#
 1  module ROM_code#(parameter DEPTH=32,WIDTH=16)( addr , out, );
 2
 3          input[3:0] addr;
 4          output[WIDTH-1:0] out;
 5
 6          reg [WIDTH-1:0] out;
 7          reg [WIDTH-1:0] ROM[0:DEPTH-1];
 8
 9          always @(*) begin
10                  $readmemb("sequences.txt",ROM);
11                  out=ROM[addr];
12          end
13
14  endmodule
```

➢ Comment

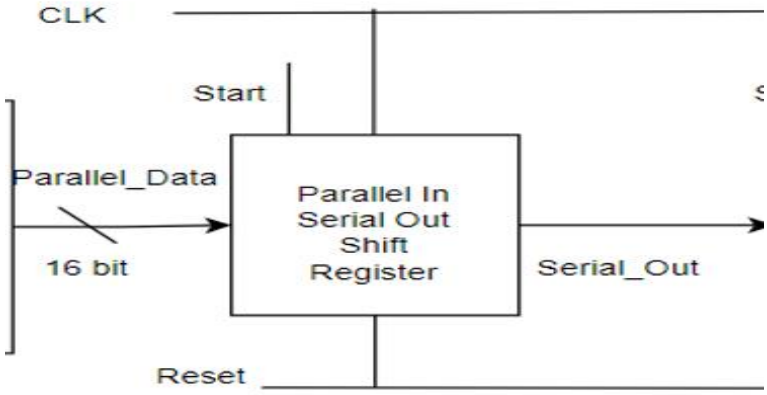
The module has two ports:

- addr is a 4-bit input representing the address to read from.
- out is a WIDTH-bit output representing the data read from the ROM at the specified address.

Inside the module, there is a 2-dimensional array ROM of size DEPTH by WIDTH. It represents the memory contents of the ROM. Each element in the ROM array is a WIDTH-bit value. it reads the contents of a file called "sequences.txt" using the $readmemb system function and assigns the values to the ROM array. The $readmemb function reads binary data from the file and initializes the ROM array.

Finally, the out signal is assigned the value from the ROM array at the address specified by the addr input.

PISO (parallel input serial output) block design



First design

```
Ln#
 1  module PISO_SHREG#(parameter WIDTH=16)(
 2
 3          input[WIDTH-1:0] parallel_in,
 4          input start,
 5          input CLK,
 6          input RST,
 7          output reg serial_out);
 8
 9          integer i;
10
11          always @(posedge CLK or negedge RST) begin
12
13                  if(!RST) begin
14                          serial_out<=1'b0;
15                          i<='b0;
16                  end
17                  else if(start && i<WIDTH) begin
18                          serial_out<=parallel_in[i];
19                          i<=i+1;
20                  end
21                  else begin
22                       serial_out<=1'b0;
23                       i<=1'b0;
24                  end
25
26          end
27
28  endmodule
```

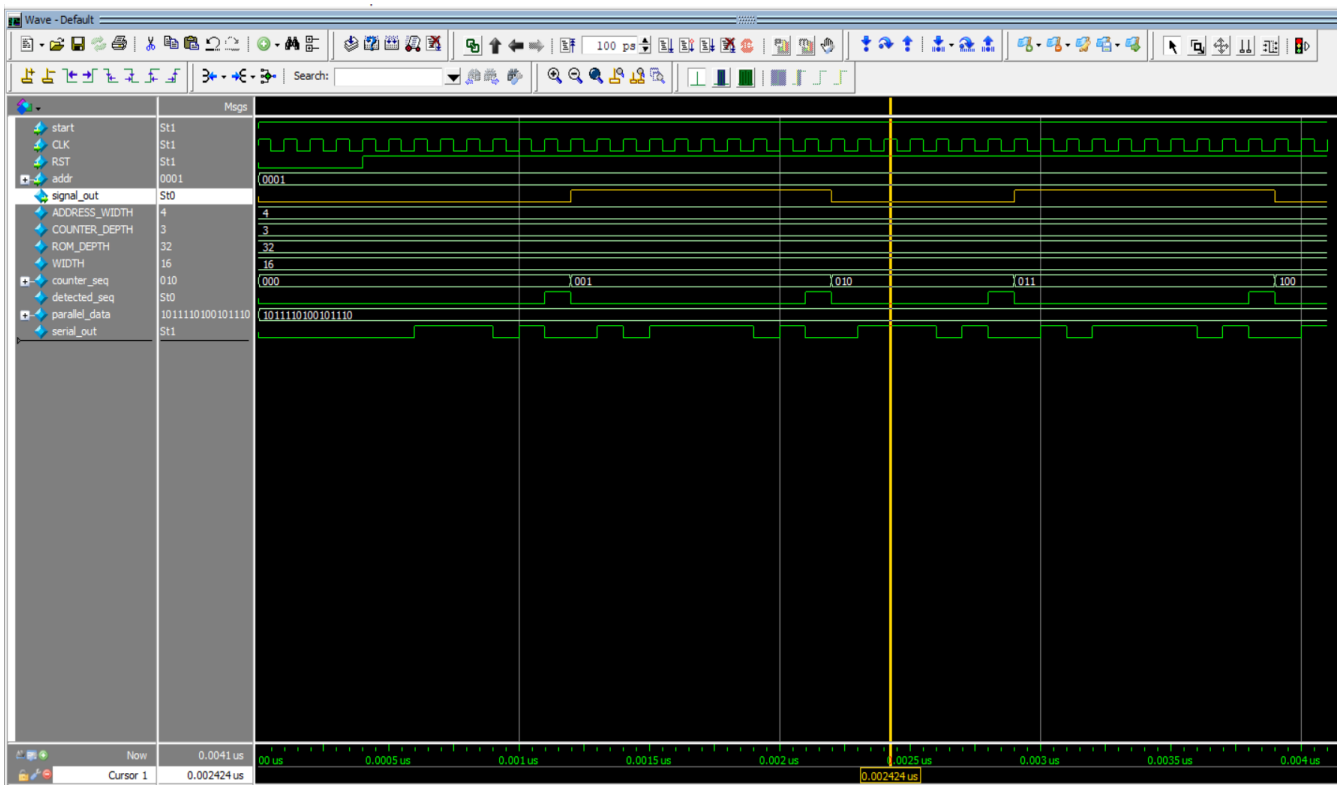
➢ comments

The module has five ports:

- parallel_in is a WIDTH-bit input representing the parallel data to be loaded into the shift register.
- start is a control input that triggers the shifting operation.
- CLK is the clock input.
- RST is the reset input.
- serial_out is a 1-bit output representing the serial data output from the shift register.

Inside the module, there is an integer variable i used as a counter to keep track of the current bit position being shifted. The always block is a sequential logic block that executes on the positive edge of the clock (posedge CLK) or the negative edge of the reset (negedge RST). This block controls the shifting operation of the shift register. If the reset signal is high (deasserted), the block checks if the start signal is asserted and if the counter i is less than WIDTH. If both conditions are true, it loads the parallel_in data at bit position i into the serial_out signal and increments the counter i by 1. This means that when the start signal is asserted and the counter is within the valid range, data is shifted out serially.

If we use the design above the output of the final system will be as shown in the following figure



The disadvantage of the above design is that when the counter i reaches to the WIDTH value (number of bits stored in the ROM memory) the counter will be 0 again and hence will return back to count from the start of the value stored in rom and will continue that more than one time and will count the sequences exist in the register once the start value ,clock are on and reset signal is deasserted to the system so this design isn't efficient.

Second design

```
Ln#
 1  module PISO_SHREG#(parameter WIDTH=16)(
 2
 3          input[WIDTH-1:0] parallel_in,
 4          input start,
 5          input CLK,
 6          input RST,
 7          output reg serial_out);
 8          reg [WIDTH-1:0]temp;
 9          always @(posedge CLK or negedge RST) begin
10
11                  if(!RST) begin
12                          serial_out<=1'b0;
13                          temp<=parallel_in;
14                  end
15                  else if(start) begin
16                  serial_out<=temp[0];
17                  temp <= {1'b0, temp[15:1]};
18                  end
19                  else begin
20                       serial_out<=1'b0;
21                  end
22
23          end
24
25
26
27  endmodule
28
29
30
31
32
```

➢ comment

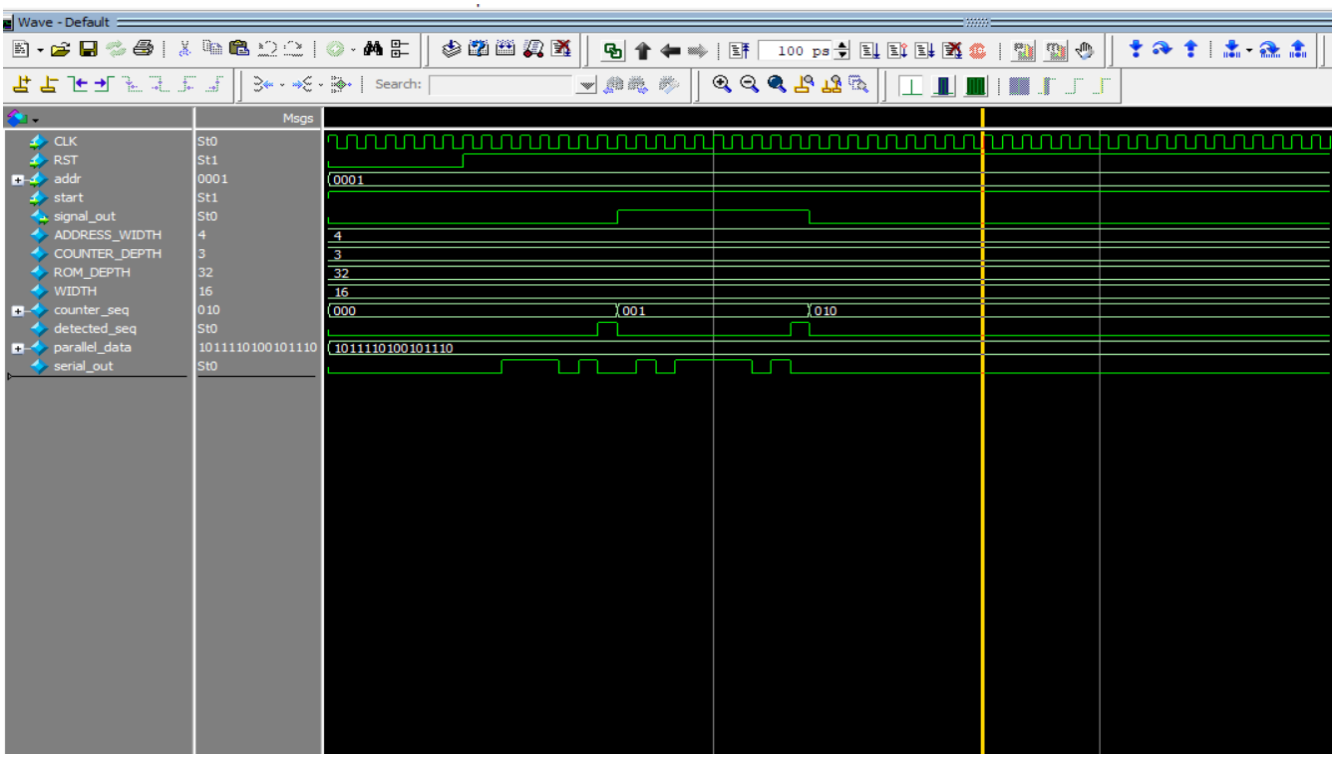
The module has five ports:

- parallel_in is a WIDTH-bit input representing the parallel data to be loaded into the shift register.
- start is a control input that triggers the shifting operation.
- CLK is the clock input.
- RST is the reset input.
- serial_out is a 1-bit output representing the serial data output from the shift register.

Inside the module, there is a reg array temp of size WIDTH used to store the parallel input data. If the reset signal RST is low (asserted), the block sets the serial_out signal to 0 and stores the parallel_in data into the temp array. This means that when the reset signal is active, the shift register is cleared, and the temp array is loaded with the parallel input data. If the reset signal is high (deasserted), the block checks if the start signal is asserted. If the start signal is asserted, it assigns the most significant bit (temp[0]) to the serial_out signal, and shifts the temp array to the left by one position. The new bit shifted in from the start input is 0. This means that when the start signal is asserted, the parallel input data is shifted out serially. If the start signal is not asserted, meaning the shifting operation is not active, the block sets the serial_out signal to 0.

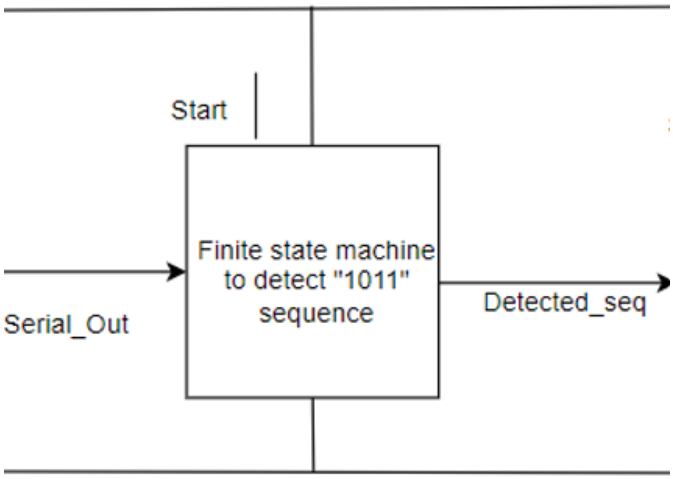
The advantage of this design that when the data bits stored in the rom register finished, the number of times that the sequence is detected will be reduced and it will be counted by the counter one time only, not more than one time and hence, we solved the problem of first design.

Output when we use design 2 of PISO block



As shown in the figure above, the number of detected sequences will be two only as expected so we solved the problem of first design.

Sequence detector




```
Ln#
 1  module SEQ_DETECTOR (
 2          input  wire        start , serial_in ,
 3          input  wire        rst,
 4          input  wire        clk,
 5          output reg         detected_seq
 6  );
 7
 8
 9          localparam  [2:0]    IDLE = 3'b000,
10                                                          S1 = 3'b001,
11                                                          S11 = 3'b011,
12                                                          S011 = 3'b010,
13                                                          S1011 = 3'b110;
14
15          reg    [2:0]         current_state,
16                                                          next_state ;
17
18          // state transition
19          always @(posedge clk or negedge rst) begin
20                      if(!rst) begin
21                              current_state <= IDLE ;
22                      end
23                      else if(start) begin
24                              current_state <= next_state ;
25                      end
26                      else begin
27                              current_state <= IDLE ;
28                      end
29           end
30
31          // next_state logic
32          always @(*) begin
33                  case(current_state)
34                          IDLE: begin
35                                  if(serial_in)
36                                          next_state = S1 ;
37                                  else
38                                          next_state = IDLE ;
39                          end
40                          S1: begin
41                                  if(serial_in)
42                                          next_state = S11 ;
43                                  else
44                                          next_state = IDLE ;
45                          end
```

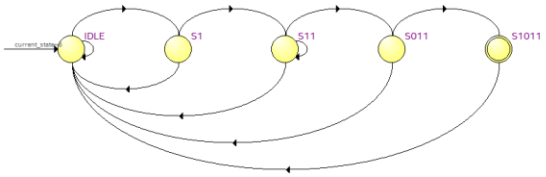
```
                        S11: begin
                                if(!serial_in)
                                        next_state = S011 ;
                                else
                                        next_state = S11 ;
                                end
                        S011: begin
                                if(serial_in)
                                        next_state = S1011 ;
                                else
                                        next_state = IDLE ;
                                end
                        S1011: begin
                                        next_state = IDLE ;
                                end
                                default:
                                        next_state = IDLE ;
                endcase
        end


        // next_state logic
        always @(*) begin
                case(current_state)
                        IDLE: begin
                                detected_seq   =  1'b0 ;
                        end
                        S1: begin
                                detected_seq   =  1'b0 ;
                        end
                        S11: begin
                                detected_seq   =  1'b0 ;
                        end
                        S011: begin
                                detected_seq   =  1'b0 ;
                        end
                        S1011: begin
                                detected_seq=1'b1;
                        end
                        default: begin
                                detected_seq   =  1'b0 ;
                        end
         endcase
        end

endmodule
```

➢ comments

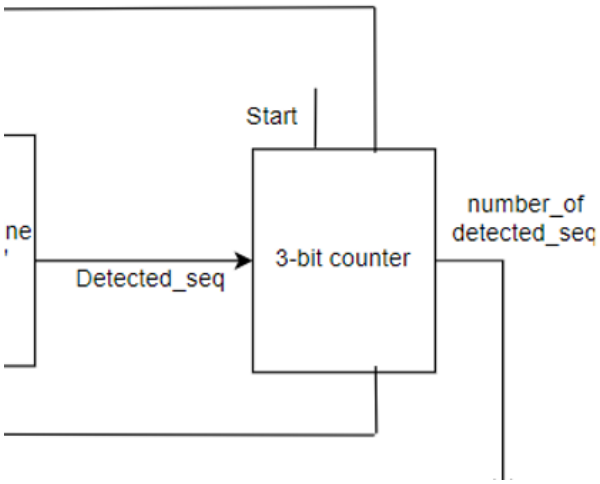
a finite state machine used to detect sequence of 1011 in the given binary number stored in ROM memory.



	Source State	Destination State	Condition
1	IDLE	S1	(serial_in).(start)
2	IDLE	IDLE	(!serial_in) + (serial_in).(!start)
3	S1	IDLE	(!serial_in) + (serial_in).(!start)
4	S1	S11	(serial_in).(start)
5	S11	IDLE	(!start)
6	S11	S11	(serial_in).(start)

Transitions | Encoding

3-bit binary counter

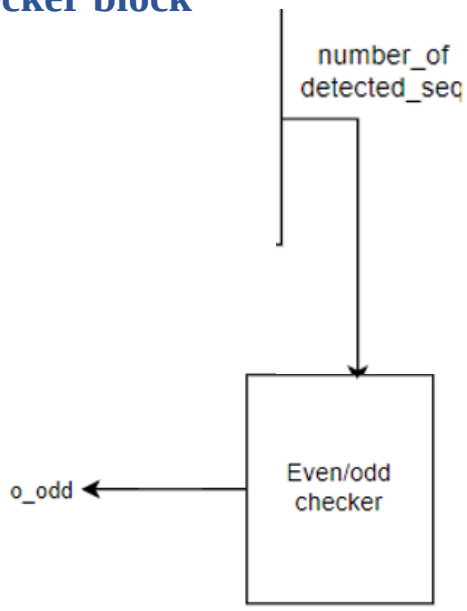



```
Ln#
 1  module counter#(parameter WIDTH=3)(
 2          input detected_seq,
 3          input start,
 4          input CLK,
 5          input RST,
 6          output reg [WIDTH-1:0]counter_seq);
 7
 8
 9          always @(posedge CLK or negedge RST) begin
10
11                  if(!RST) begin
12                          counter_seq<='b0;
13                  end
14                  else if(start && detected_seq) begin
15                          counter_seq<=counter_seq+1'b1;
16                  end
17          end
18
19  endmodule
```

➢ comments

This is a simple counter that counts number of detected sequences 1011 in each value in a specific address in the ROM memory. When detected_sequence signal is high the counter value will increase to say give the value of how many numbers of the detected sequence exits.

Even/odd checker block



E:/AUC assignments/Digital/Karim_LAB6/Karim_LAB6/even_odd_checker.v (/Top/Checker_Instance) - Default

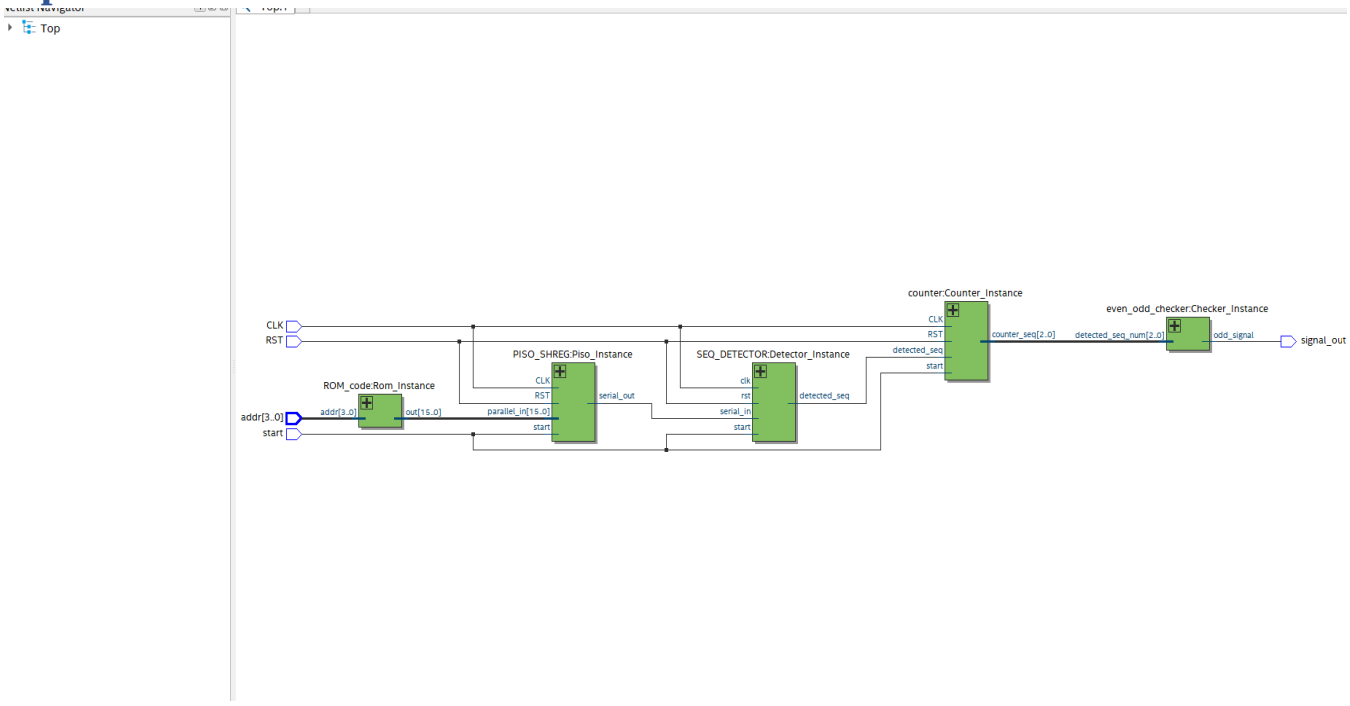
Ln#

```
1  module even_odd_checker#(parameter WIDTH=3)(
2          input [WIDTH-1:0] detected_seq_num,
3          output odd_signal
4          );
5
6          assign odd_signal = (detected_seq_num[0] == 1'b1) ? 1'b1 : 1'b0;
7  endmodule
```

➢ comments

even/odd checker is a simple block which indicate if the detected sequence number value is an even or odd value. the output of this block will ne 1 in case the value of detected sequence entered to it is odd value and equal to 0 if it is an odd value.

top module

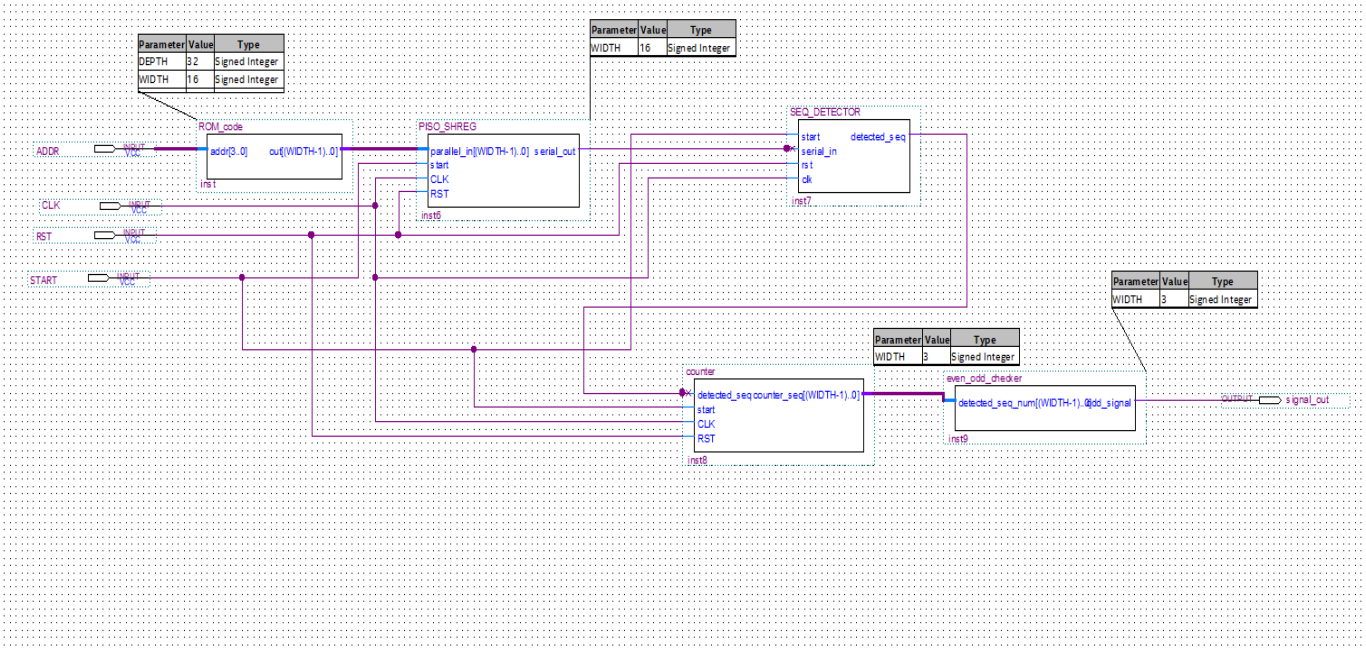



```
Ln#
 1  module Top #( parameter ROM_DEPTH=32 ,  WIDTH=16 , ADDRESS_WIDTH=4 , COUNTER_DEPTH=3 )(
 2          input start,
 3          input CLK,
 4          input RST,
 5          output signal_out,
 6          input[ADDRESS_WIDTH-1:0] addr
 7          );
 8
 9          wire [WIDTH-1:0] parallel_data;
10          wire serial_out;
11          wire detected_seq;
12          wire [COUNTER_DEPTH-1:0] counter_seq;
13          ROM_code  #( ROM_DEPTH , WIDTH ) Rom_Instance (addr, parallel_data );
14
15          PISO_SHREG #(WIDTH) Piso_Instance ( parallel_data , start , CLK , RST , serial_out);
16
17          SEQ_DETECTOR Detector_Instance ( start , serial_out , RST , CLK , detected_seq );
18
19          counter #(COUNTER_DEPTH) Counter_Instance ( detected_seq, start , CLK , RST ,counter_seq);
20
21          even_odd_checker #(COUNTER_DEPTH) Checker_Instance ( counter_seq , signal_out );
22
23  endmodule
```

➢ Comments on top module of the design

1. If we want to apply a new address to detect the number of detected sequences in another position in rom, we first must apply a reset signal to all of the system or make start signal equal to zero first then to be able to count from zero the number of detected sequences in new ROM address.
2. To avoid case 1. we can use done signal from each block to enable blocks to on serially in order, so we be able to finally reset the counter automatic after detected sequences is count for specific place in ROM.

➢ Block diagram of the design using Quartus prime



➢ Test bench to test the design

E:/AUC assignments/Digital/shift_add_cir/TOP_tb.v (/Top_tb) - Default

```
Ln#
1  module Top_tb #( parameter ROM_DEPTH=32 ,  WIDTH=16 , ADDRESS_WIDTH=4 , COUNTER_DEPTH=3 )();
2
3  //Signal declaration
4  reg start,CLK, RST;
5  reg [ADDRESS_WIDTH-1:0] addr;
6  wire signal_out;
7
8  //DUT instantiation
9  Top #( .ROM_DEPTH(ROM_DEPTH) ,  .WIDTH(WIDTH) , .ADDRESS_WIDTH(ADDRESS_WIDTH) , .COUNTER_DEPTH(ADDRESS_WIDTH) ) DUT (.start(start),.CLK(CLK),
10 .RST(RST),.signal_out(signal_out),.addr(addr));
11
12
13 initial begin
14 start=1'b1;
15 CLK=1'b0;
16 RST=1'b0;
17 addr='b0001;
18 #20;
19 RST=1'b1;
20 #200;
21
22 $stop;
23 end
24 always #5 CLK=~CLK;
25 endmodule
26
```