The 110 sequence detector module:

One of the objectives of this lab was to program a Verilog description of finite state machine. In this particular case the FSM in question was a sequence detector whose job was to output a logic 1 every time the sequence 110 was recognised. The solution implemented had to also make use of a D type Flip-Flop in order to synchronise the state with the clock signal. Sequential logic was to be implemented and the output had to be a function of the present state as well as of the input, in other words the FSM was a Mealy type machine. On the second part of the exercise the output was also to be synchronised with the clock signal.
In the solution described on this lab report the following modules were used:

- The sequence detector module was implemented using case logic and a D flip-Flop module; it generated an output based on the current state of the FSM machine and on the input.
- The test bench module was arranged as to test for 4 different circumstance as follows:
    - Stings of zeros
    - Strings of ones
    - 10101 sequence
    - Sequences that would cycle the FSM through all its states in order
    - Test response to the reset signal

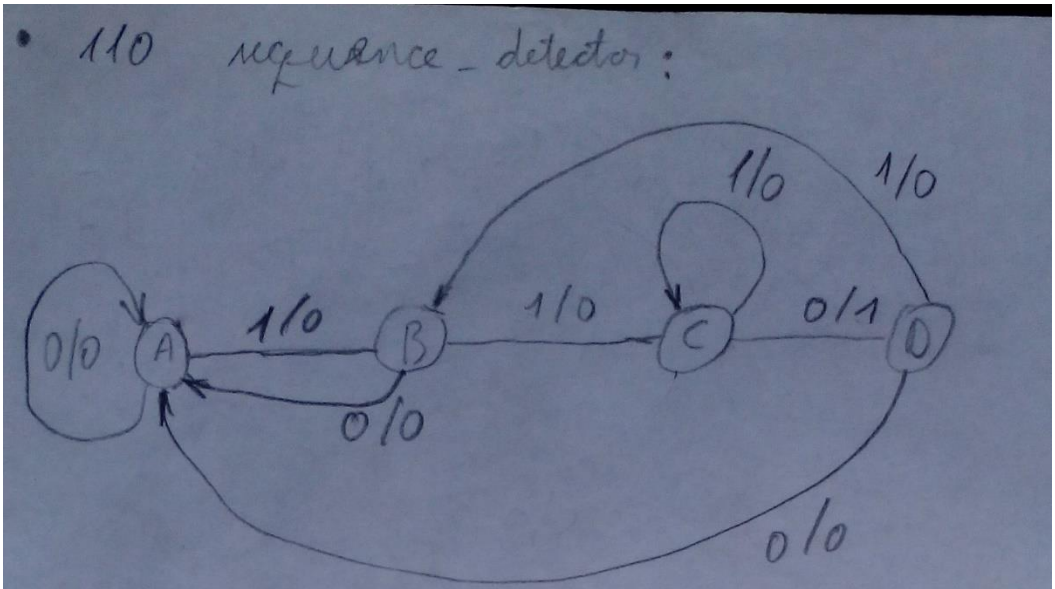
The timing of the bit stream was designed in such a way as for every clock cycle only one bit to be read in as input.
On the second part of the exercise the machine become fully synchronous with both the state and the output updating at the rising edge of every clock cycle.
The unit under test has behaved as expected, the results, code and waveforms being included next.

110 sequence detector minimisation

Due to the fact that the sequence detector machine had only a few number of states the equivalent states could be identified by inspection and there was no need for employing the formal minimisation method by using an implication chart.

The states A and D were found to be equivalent and in consequence state D was eliminated and in state C as next state for input 0, state D was substituted with state A.

The figures [] next present the Diagram and State Table for the FSM both before and after the minimisation.

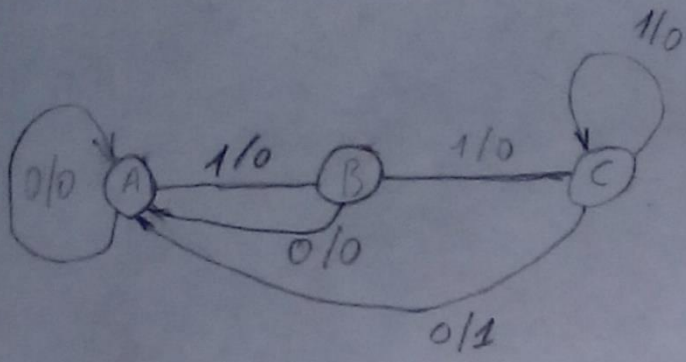


• State table for sequence detector:

Curr State	Next State Input 0	Next State Input 1	Output Input 0	Output Input 1
A	A	B	0	0
B	A	C	0	0
C	D	C	1	0
D	A	B	0	0

Figure [] Diagram and State table for FSM

- 110 sequence_detector revised diagram:



- State table for sequence detector:

Curr State	Next State Input 0	Next State Input 1	Output Input 0	Output Input 1
A	A	B	0	0
B	A	C	0	0
C	~~D~~ A	C	1	0
~~D~~	A	B	0	0

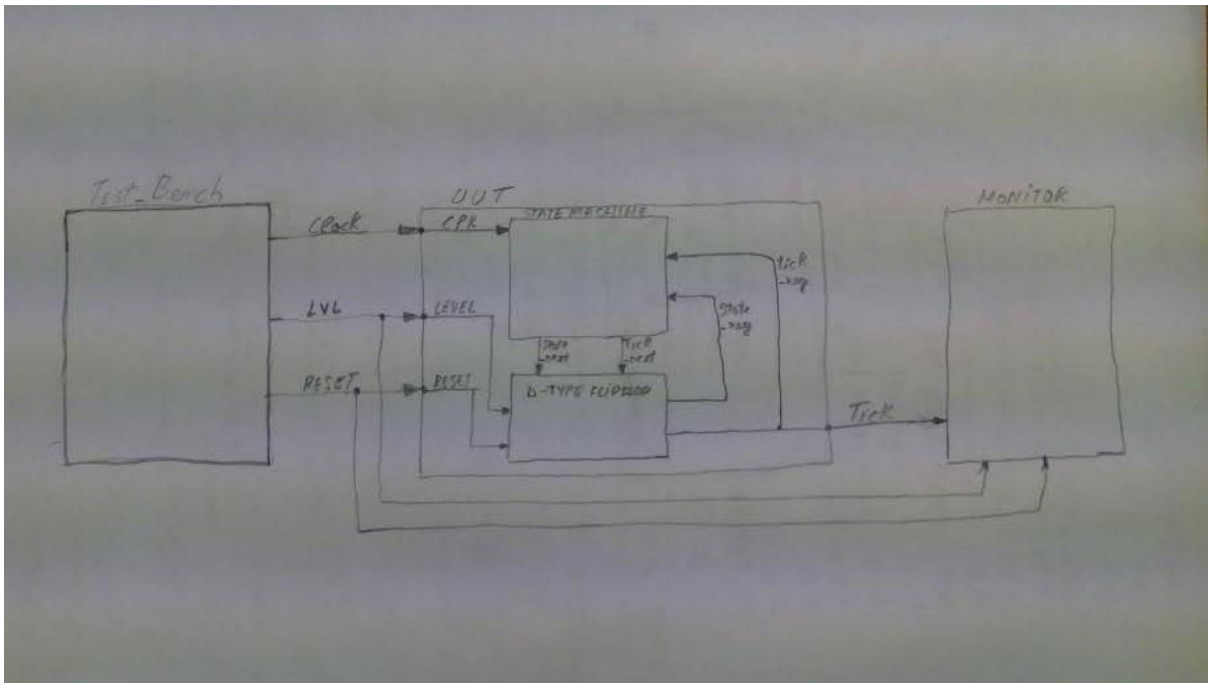
Equivalent states
A ≡ D

- State table after minimisation:

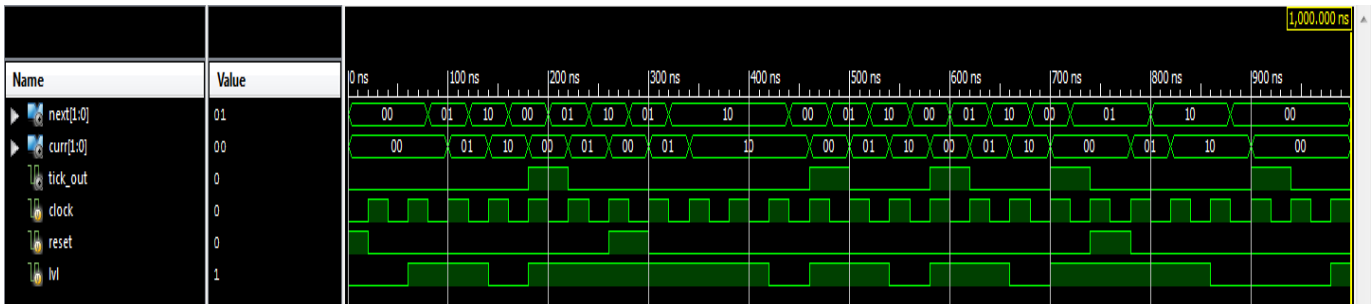
Curr State	Next State Input 0	Next State Input 1	Output Input 0	Output Input 1
A	A	B	0	0
B	A	C	0	0
C	A	C	1	0

Figure [] Revised diagram and State table after minimisation for FSM

Block diagram of the FSM for the 110 sequence detector:



The waveforms for the 110 sequence detector module



Code for 110 sequence detector module

```
`timescale 1ns / 1ps
module sequence_detect
(
input wire clk, reset,
input wire level,
output wire tick,
output wire [1:0] curr_s,next_s
);
// symbolic state declaration
localparam capital_a = 2'b00, capital_b= 2'b01, capital_c = 2'b10, zero = 1'b0, one = 1'b1 ;
// signal declaration
reg [1:0] state_reg, state_next;
reg  tick_reg,tick_next;
// state register
always @(posedge clk, posedge reset)
        if (reset)
                begin
                state_reg <= capital_a;
                tick_reg <= zero;
                end
        else
                begin
                state_reg <= state_next;
                tick_reg <= tick_next;
                end
// next-state logic and output logic
always @(negedge clk)
```

```
begin
state_next = state_reg; // this line makes the output depend on the present state
tick_next = zero; // default output: 0
//the following logic makes the output depend on the value of the input
case (state_reg)
capital_a:
        if (level)
        begin
                state_next = capital_b;
        end
        else
                state_next = capital_a;
capital_b:
        if (level)
        begin
                state_next = capital_c;
        end
        else
                state_next = capital_a;


capital_c:
        if (level)
        begin
                state_next = capital_c;
        end
        else
        begin
                state_next = capital_a;
```

```
                    tick_next = one;
                    end

default:
            begin
                    state_next = capital_a;
                    //tick_next = zero;
            end
        endcase
        end
assign curr_s = state_reg;
assign next_s = state_next;
assign tick = tick_reg;
endmodule
```

Code for 110 sequence detector test bench:

```
`timescale 1ns / 1ps
module sequence_detect_tb;
//declaring inputs and outputs
        reg clock, reset, lvl;          //signal corresponds to 'level'
        wire[1:0] curr, next;
        wire tick_out;
        //instatiation of circuit under test!
        sequence_detect uut
                (.clk(clock),  .reset(reset),  .level(lvl),  .tick(tick_out),  .curr_s(curr),
.next_s(next));

        initial
        begin                           //initialise clock
```

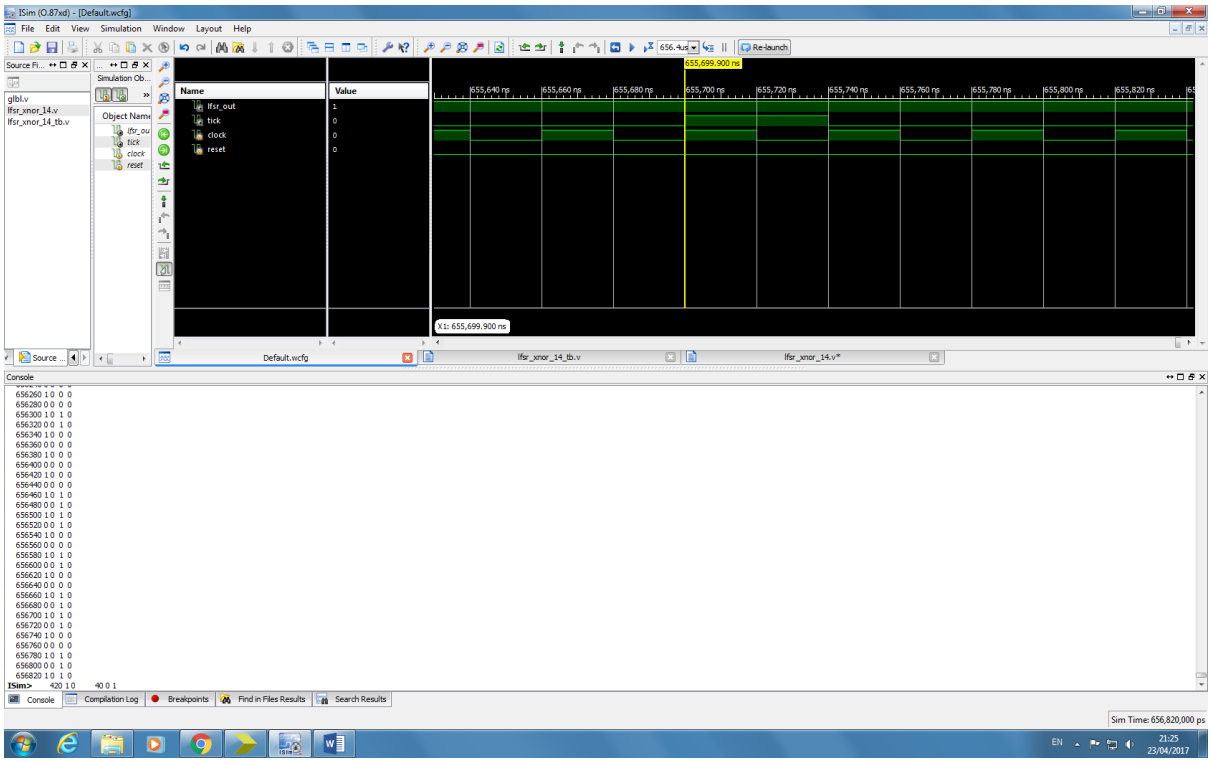
```
        clock = 1'b0;
end

initial
begin                       //define signal


        reset = 1'b1;
        lvl = 1'b0;
        #20
        reset = 1'b0;
        lvl = 1'b0;
        #40
        lvl = 1'b1;
        #40
        lvl = 1'b1;
        #40
        lvl = 1'b0;
        #40
        lvl = 1'b1;
        #40
        lvl = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b0;
        #40
        lvl = 1'b1;
        #40
```

```
lvl = 1'b1;
#40
lvl = 1'b0;
#40
lvl = 1'b1;
#40
lvl = 1'b1;
#40
lvl = 1'b0;
#40
lvl = 1'b1;
#40
lvl = 1'b1;
#40
lvl = 1'b0;
#40
lvl = 1'b1;
#40
lvl = 1'b1;
reset = 1'b1;
#40
reset = 1'b0;
#40
lvl = 1'b1;
#40
lvl = 1'b0;
#40
lvl = 1'b0;
#40
```

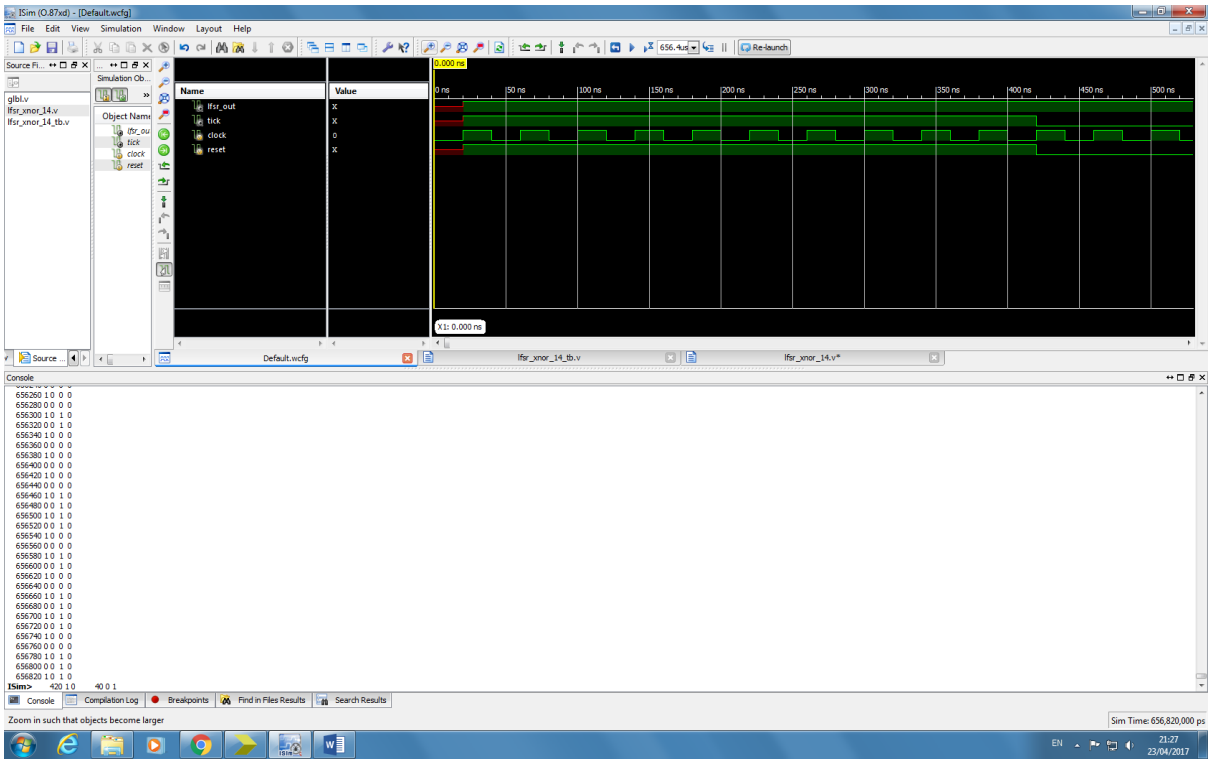
```
		lvl = 1'b0;
		#40
		lvl = 1'b1;
		#40
		lvl = 1'b1;
		#40
		lvl = 1'b0;
		#40
		$stop;

	end

	always
	begin                                   //tick the clock/
	#20 clock = ~clock;      //flip. note the period
	end

	initial
		$monitor($stime,, clock,, reset,, lvl,,, tick_out,,, curr,,, next);
endmodule
```

LFSR_XNOR_14 module:

Tick waveform



10 cycles reset waveform



Timing for it to circle back

The time expected for the LFSR to circle back to the initial seed value is dependent on the clock period (40 ns) and the size of the LFSR (14 bits).The following expression was used to calculate it:

$$Time = Clock * ((2^{Size})-1)$$

$$Time = 40ns * ((2^{14})-1)$$

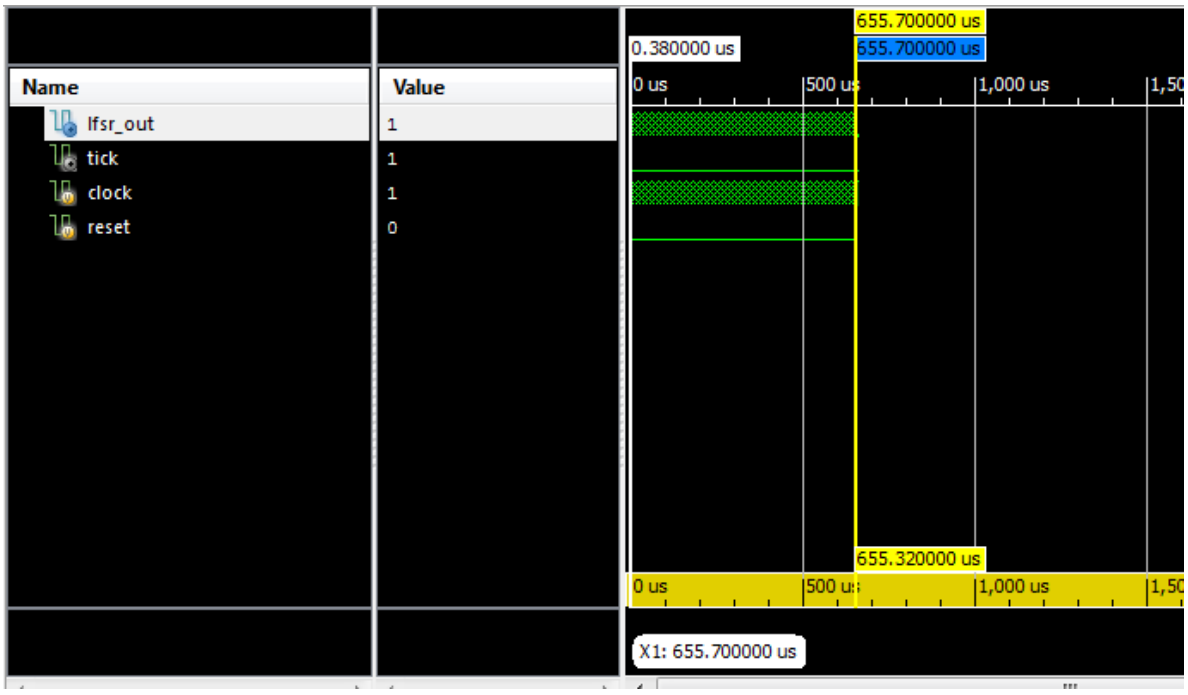
$$Time = 40ns * 16383$$

$$Time = 655.320ns$$

As can be seen on the figure below the theoretical time and the experimental time was exactlly as calculated :

$$Time\ (experimental) = 655700ns-380ns = 655320ns$$

The experimental time period required by the LFSR to return to its initial seed value vas measure vit the aid of cursors in the ISim enviroment. The first cursor was set at the rising edge of the last reset signal, while the second cursor was set at the rising edge of the tick signal. The setting of the two cursors can be seen on the figures [] [] bleow, while the figure [] shows the time period between the two cursors.



Figure[] Cursors used to measure the time period

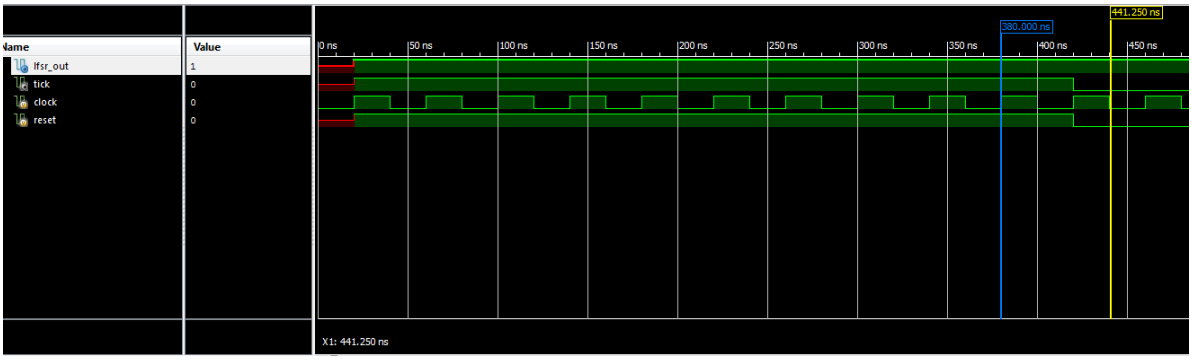


Figure [] Set up of the first cursor

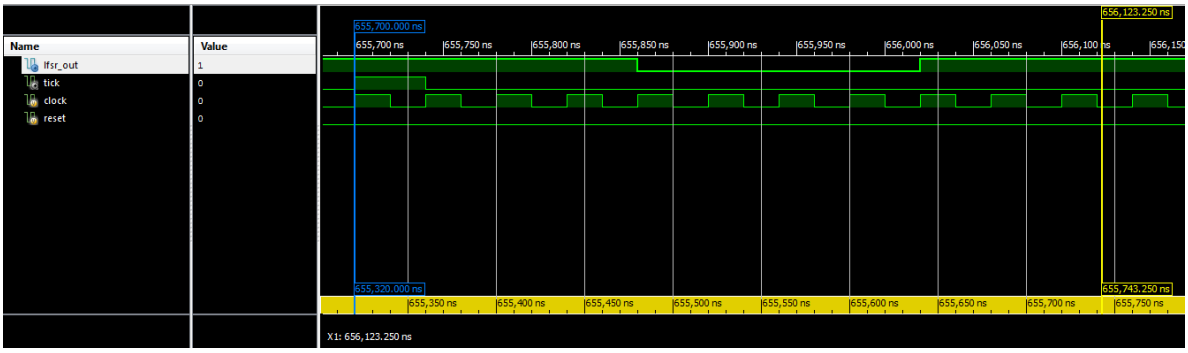


Figure [] Set up of the second cursor

Code for the LFSR_XNOR_14 module:

```
`timescale 1ns / 1ps
module lfsr_xnor_14
#(parameter N=14)
 (
 input wire clk, reset,
 output wire lfsr_out, max_tick
 );

 //signal declaration
 reg [N-1:0] lfsr_reg; // register storage
 reg [N-1:0] lfsr_next; // next value
 reg lfsr_tap; // to hold feedback
```

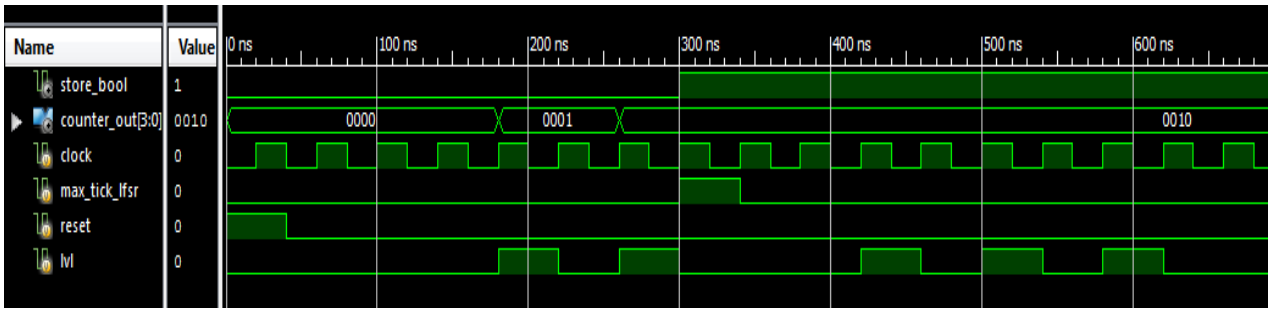
```
// constnd declaration
localparam seed = 14'b11110000111100, N1=N-1; // local parameter to hold the seed value
// body

// register
always @(posedge clk, posedge reset)
if (reset)
lfsr_reg <= seed; // a seed value
else
lfsr_reg <= lfsr_next;

// next-state logic
always @*
begin
// generate the feedback by XNOR of tap 13,4,2 and 0
lfsr_tap = lfsr_reg[N1] ~^ lfsr_reg[4]~^ lfsr_reg[2]~^ lfsr_reg[0];
// feedback goes into 0 position. Other bits shift up
lfsr_next = {lfsr_reg[N1-1:0],lfsr_tap };
end // next state logic

// output logic
assign lfsr_out = lfsr_reg[N1];
assign max_tick = (lfsr_reg==seed) ? 1'b1 : 1'b0; // output 1 when the initial seed value is reached agin after (2**N)-1 cycles


endmodule
```

Code for the LFSR_XNOR_14_tb module:

```
`timescale 1ns / 1ps
module lfsr_xnor_14_tb ();
//declaring inputs and outputs
        reg clock, reset;
        wire lfsr_out,tick;

//instantiation of circuit under test!
        lfsr_xnor_14 uut
                (.clk(clock), .reset(reset), .lfsr_out(lfsr_out), .max_tick(tick));
        Initial

//initialise clock
        begin

                clock = 1'b0;
        end

//define signal
        initial
        begin

                #20
                reset = 1'b1;
                #40
                reset = 1'b1;
                #40
                reset = 1'b1;
                #40
```

```
                reset = 1'b1;
                #40
                reset = 1'b1;
                #40
                reset = 1'b1;
                #40
                reset = 1'b1;
                #40
                reset = 1'b1;
                #40
                reset = 1'b1;
                #40
                reset = 1'b1;
                #40
                reset = 1'b0;

                $stop;
        end
        always
//tick the clock
        begin
        #20 clock = ~clock;      //flip. note the period
        end

        initial
                $monitor($stime,, clock,, reset,,,lfsr_out,,,tick);
endmodule
```

Counter_fsm module



Counter_fsm module code:

```
`timescale 1ns / 1ps
module counter_fsm
#(parameter N=4) // used to make the code easier to reuse
 (
 input wire clock, tick_fsm, reset, max_tick_lfsr,
 output wire  store_bool,
 output wire [N-1:0] q;
 );
 //signal declaration
 reg stop;
 reg [N-1:0] r_reg, r_next; //used to store the current value and generat the next one
```

```
// body
// register update
always @( posedge reset, posedge tick_fsm )
begin
        if (reset)// this condition makes sure that this module can reset when the general detection system resets
                begin
                        r_reg <= 0;
                        stop  <=0;
                end
        else
                r_reg <= r_next;// this line update the register to the next value
end
// next state logic
always @ *
        begin
        if ( max_tick_lfsr || stop == 1)// this condition is used to stop the counterfrom updting the value in r_reg when the sequence on the LFSR module
         begin                  //returnes to the initial seed value
                stop <= 1;
                r_next = r_reg;
         end
        else
        r_next = r_reg + 1'b1 ;
        end
// output logic
assign store_bool = stop;
assign q = r_reg;
endmodule
```

Counter_fsm_tb module code:

```
`timescale 1ns / 1ps
module counter_fsm_tb
# (parameter N=4)
	();
//declaring inputs and outputs
	reg clock, max_tick_lfsr, reset, lvl;
	reg [N-1:0]	curr;
	wire  store_bool;
	wire [N-1:0] counter_out,  next;

	//instatiation of circuit under test!
	counter_fsm uut
		( .clock (clock), . max_tick_lfsr (max_tick_lfsr), .reset(reset), .tick_fsm (lvl), .q (counter_out), .store_bool (store_bool));
//start clock signal
	initial
		begin
		clock = 1'b0;
		end
	//define signal
	initial
	begin
		reset = 1'b1;
		lvl = 1'b0;
		max_tick_lfsr = 1'b0;
		#40
		reset = 1'b0;
		#20
		lvl = 1'b0;
```

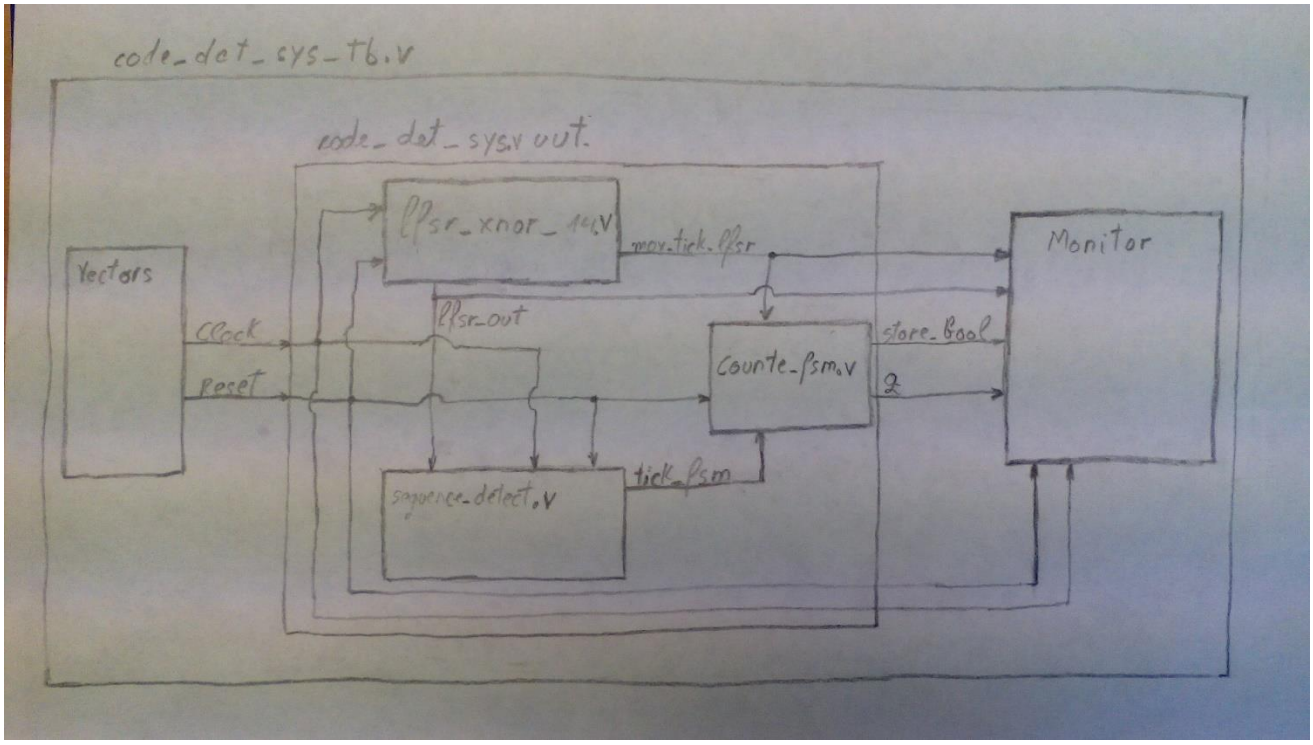
```
#40
lvl = 1'b0;
#40
lvl = 1'b0;
#40
lvl = 1'b1;
#40
lvl = 1'b0;
#40
lvl = 1'b1;
#40
max_tick_lfsr = 1'b1;
lvl = 1'b0;
#40
max_tick_lfsr = 1'b0;
#40
lvl = 1'b0;
#40
lvl = 1'b1;
#40
lvl = 1'b0;
#40
lvl = 1'b1;
#40
lvl = 1'b0;
#40
lvl = 1'b1;
#40
lvl = 1'b0;
```

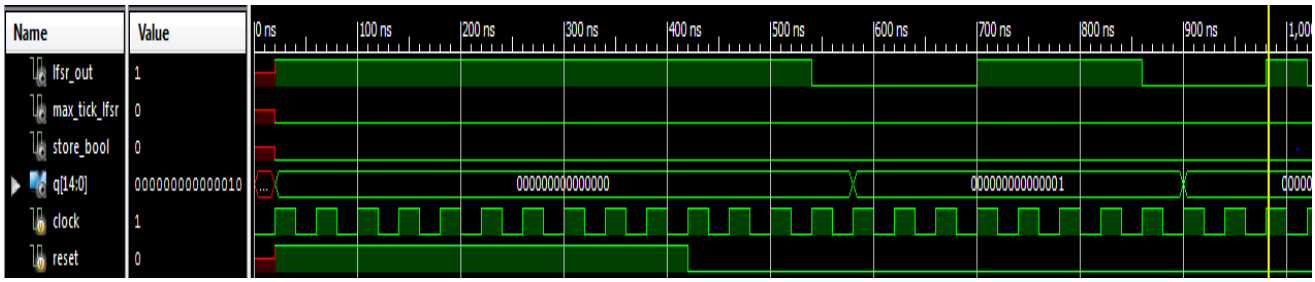
```
        #40
        lvl = 1'b0;
        #40
        lvl = 1'b1;
        #40
        lvl = 1'b0;
        #40
        lvl = 1'b1;
        #40
        lvl = 1'b0;
        #40
        lvl = 1'b0;
        #40
        lvl = 1'b1;
        #40
        lvl = 1'b0;
        #40
        lvl = 1'b0;
        #40
        lvl = 1'b1;
        #40
        $stop;
end
//tick the clock/
always
begin
#20 clock = ~clock;
end
initial
```

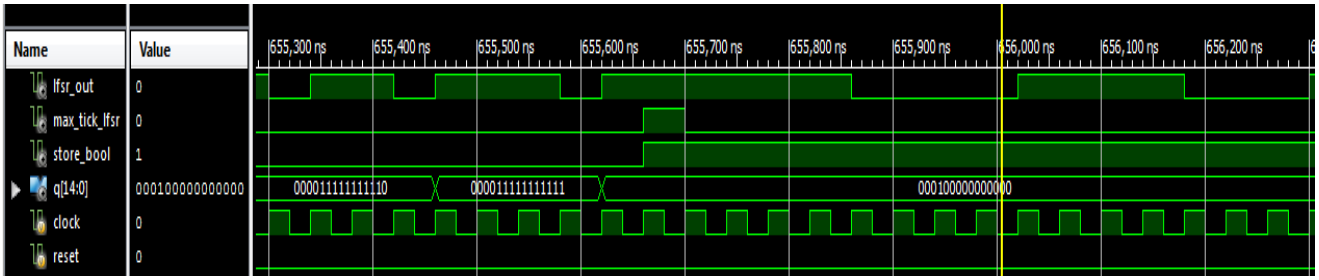
$monitor($stime,, max_tick_lfsr,, reset,, lvl,,, counter_out);

Endmodule

Code_det_sys module







Code_det_sys.v module code:

```
`timescale 1ns / 1ps
module code_det_sys
    (
        input wire clock, reset,
        output wire max_tick_lfsr, lfsr_out, store_bool,
        output wire [10:0] q
     );
    //internal wire declaration
    wire tick_fsm;
    //instantiating the required modules
    lfsr_xnor_14 unit_1 (. clk (clock),. reset (reset),. lfsr_out (lfsr_out),. max_tick (max_tick_lfsr));
    sequence_detect unit_one (. clk (clock), .reset (reset), . level (lfsr_out), . tick (tick_fsm));
    counter_fsm unit_I(.reset (reset), . tick_fsm (tick_fsm), . max_tick_lfsr (max_tick_lfsr), . store_bool (store_bool),.q (q) );
endmodule
```

Code_det_sys_tb.v module code:

```
`timescale 1ns / 1ps
module code_det_sys_tb
    ( );
```

```
//declaring inputs and outputs
reg clock, reset;
wire lfsr_out,max_tick_lfsr, store_bool;
wire [12:0] q;
//instatiation of circuit under test!
code_det_sys uut (.clock(clock), .reset(reset), .max_tick_lfsr(max_tick_lfsr), .lfsr_out(lfsr_out),. store_bool (store_bool), .q(q));
initial
begin                                   //initialise clock
        clock = 1'b0;
end
initial
begin                       //define signal
        #20
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
```

```
        reset = 1'b1;
        #40
        reset = 1'b1;
        #40
        reset = 1'b0;
        $stop;
    end
    always
    begin                                   //tick the clock/
    #20 clock = ~clock;     //flip. note the period
    end
    initial
        $monitor($stime,, clock,, reset,,,max_tick_lfsr,,,lfsr_out,,,store_bool,,,q);
endmodule
```