MODELLING OF FINITE STATE MACHINES

- Combinational and Sequential Circuits
  - In a combinational circuit, the outputs depend only on the applied input values and not on the past history.
  - In a sequential circuit, the outputs depend not only on the applied input values but also on the internal state.
    - The internal states also change with time.
    - The number of states is finite, and hence a sequential circuit is also referred to as a Finite State Machine (FSM).
- Most of the practical circuits are sequential in nature.

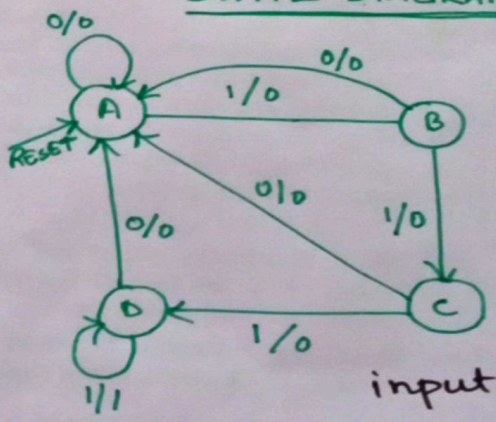
Finite State Machine (FSM)

- A FSM can be represented either in the form of a ~~[illegible]~~ state table or in the form of a state transition diagram.
  - Variations exists, eg. Algorithmic State Machine (ASM) chart.
- Example:
  - A circuit to detect 3 or more 1's in a serial bit stream.
  - The bits are applied serially in synchronism with a clock.
  - The outputs will become 1 whenever it detects 3 or more consecutive 1's in the stream.

STATE TABLE

Reset	PS	Input	NS	Output
1	-	-	A	0
0	A	0	A	0
0	A	1	B	0
0	B	0	A	0
0	B	1	C	0
0	C	0	A	0
0	C	1	D	1
0	D	0	A	0
0	D	1	D	1

STATE DIAGRAM



MEALY AND MOORE FSM TYPES

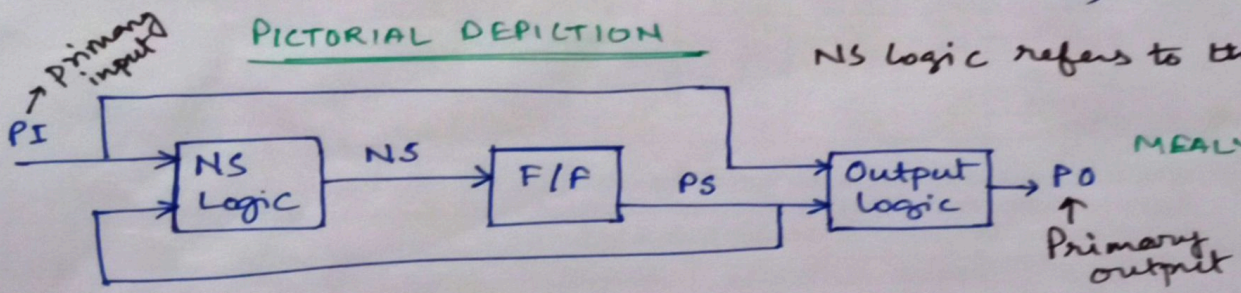
- A deterministic FSM can be mathematically defines as 5-tuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$

  where Σ is the set of input combinations, Γ is the set of output combinations, S is a finite set of states, s_0 belongs to S, called as the initial state, δ is the ~~state function~~ state-transition function, ω is the output function.

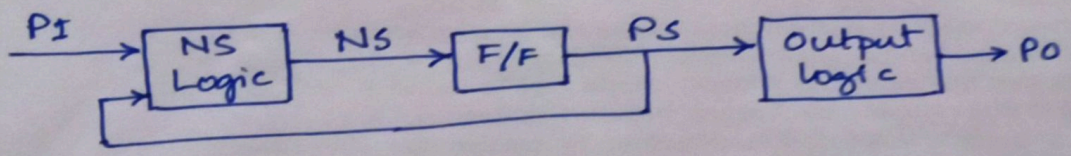
- Here, $\delta : S \times \Sigma \rightarrow S$ → cartesian product
  - Present state (PS) and present input determines the next state (NS).
- For Mealy machine, $\omega : S \times \Sigma \rightarrow \Gamma$ (output depends on state + inputs)
- For Moore machine, $\omega : S \rightarrow \Gamma$ (output depends only on the state)

PICTORIAL DEPICTION

NS logic refers to the δ function



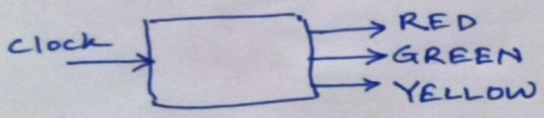
MEALY MACHINE

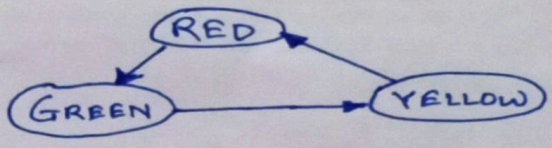


MOORE MACHINE

EXAMPLE-1

- There are three lamps, RED, GREEN and YELLOW, that should glow cyclically with a fixed time interval (say, 1second).
- Some observations:
  - The FSM will have three states, corresponding to the glowing state of the lamps.
  - The input set is null; state transition will occur whenever clock signal comes.
  - This is a Moore Machine, since the Lamp that will glow only depends on the state and not on the inputs (here null).

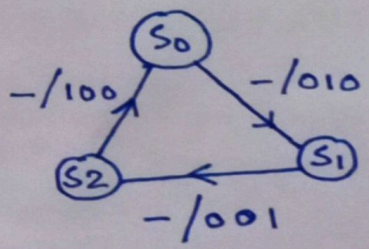





```
module cyclic_lamp (clock, light);
    input clock;
    output reg [0:2] light;
    parameter S0 = 0, S1 = 1, S2 = 2;
    parameter RED = 3'b100, GREEN = 3'b010, YELLOW = 3'b001;
    reg [0:1] state;
    always @(posedge clock)
        case (state)
            S0: begin    // S0 means RED
                    light <= GREEN;
                    state <= S1;
                end
            S1: begin    // S1 means GREEN
                    light <= YELLOW;
                    state <= S2;
                end
            S2: begin
                    light <= RED;
                    state <= S0;
                end
```

00 S0 → RED
01 S1 → GREEN
10 S2 → YELLOW

RGY



```
        default: begin
                    light <= RED;
                    state <= S0;
                 end
    endcase
endmodule
```

TestBench

```
module test_cyclic_lamp;
    reg clk;
    wire [0:2]  light;
    cyclic_lamp  LAMP (clock, light);

    always #5  clk = ~clk;
    initial
        begin
            clk =1'b0;
            #100  $finish;
        end
    initial
        begin
            $dumpfile ("cyclic.vcd");
            $ dumpvars(0, test_cyclic_lamp);
            $monitor ($time, "RGY: %b", light);
        end
endmodule
```

```
0   RGY : xxx
5   RGY : 100
15  RGY : 010
25  RGY : 001
35  RGY : 100
45  RGY : 010
55  RGY : 001
65  RGY : 100
75  RGY : 010
85  RGY : 001
95  RGY : 100
```

- Some comments on the solution:
  - The synthesis tool will generate five flipflops – 2 for state, and 3 for light.
  - The three output lines are also getting stored in flipflops.
    - We have used non-blocking assignment triggered by clock edge.
  - But actually we do not need separate flipflops for the outputs, as the outputs can be directly generated from the state.
  - How to achieve this?
    - Modify the Verilog code such that all assignments to light

is made in a separate "always" block.

- Use blocking assignment triggered by state change, and not by clock.

```
module cyclic_lamp (clock, light);
    input clock;
    output reg [0:2] light;
    parameter S0=0, S1=1, S2=2;
    parameter RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
    reg [0:1] state;
    always @(posedge clock)
        case (state)
            S0: state <= S1;
            S1: state <= S2;
            S2: state <= S0;
            default: state <= S0;
        endcase
    always @(state)
        case (state)
            S0: light = RED;
            S1: light = GREEN;
            S2: light = YELLOW;
            default: light = RED;
        endcase
endmodule
```

- The synthesis tool will be generating only 2 flipflops corresponding to the first clock-triggered "always" block.
- The second "always" block will be generating a combinational circuit that takes state as input and produces light as the output.

State		Light		
S_1	S_0	R	G	Y
0	0	1	0	0
0	1	0	1	0
1	0	0	0	1
1	1	x	x	x

Logic expression after minimization is:

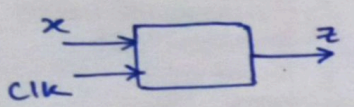
$R = \overline{S_0} \cdot \overline{S_1}$

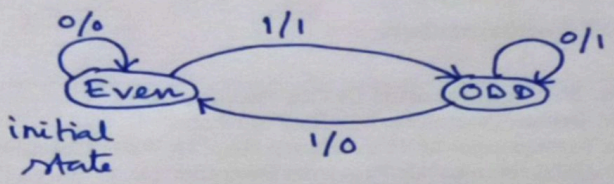
$G = S_0$

$Y = S_1$

Example - 2

- Design of a serial parity detector.
  - A continuous stream of bits is fed to a circuit in synchronous with a clock. The circuit will be generating a bit stream as output, where a 0 will indicate "even number of 1's seen so far" and a 1 will indicate "odd number of 1's seen so far".
  - Also a Moore Machine



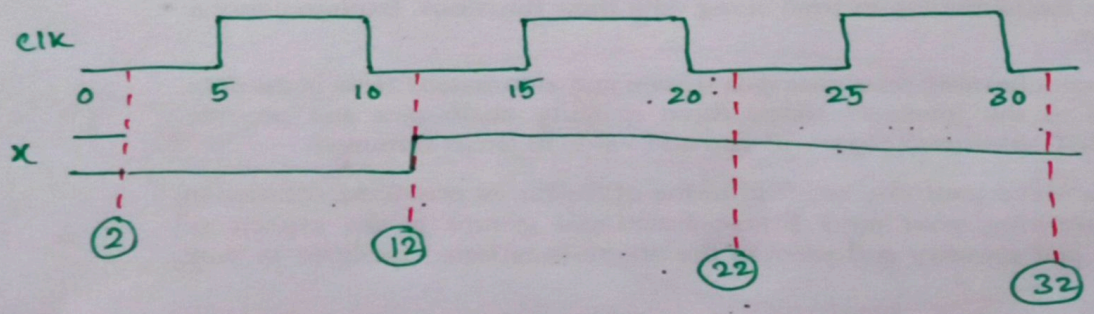


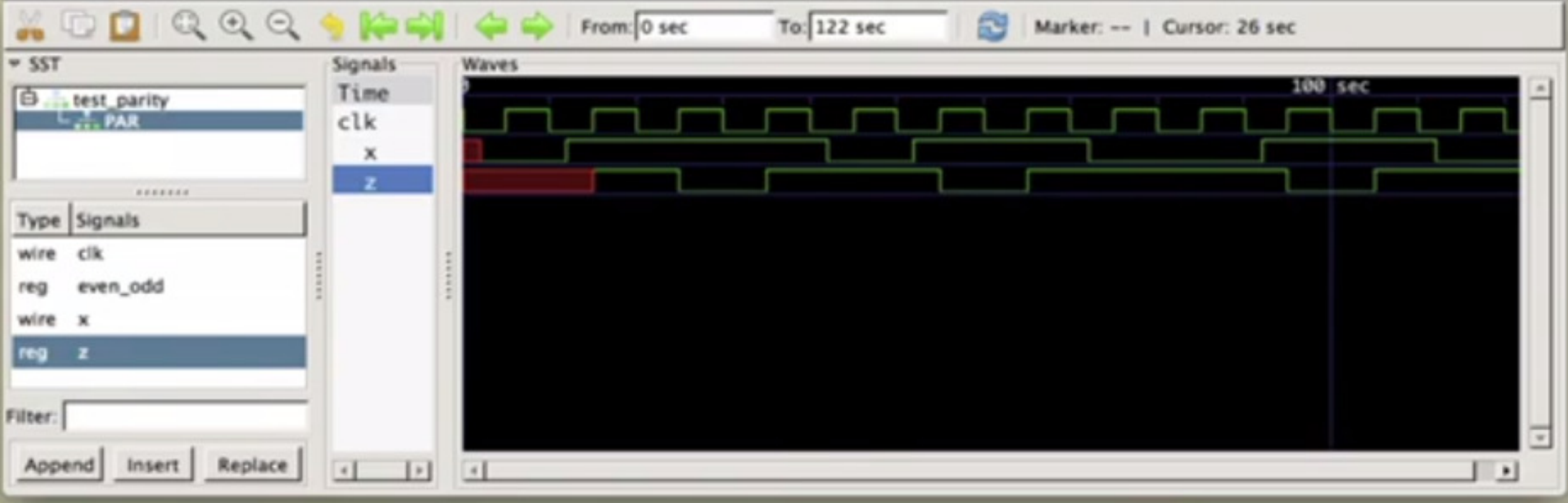

```
module parity_gen (x, clk, z);
    input x, clk;
    output reg z;
    reg even_odd; // The machine state
    parameter EVEN = 0, ODD = 1;

    always @(posedge clk)
        case (even_odd)
            EVEN: begin
                    z <= x ? 1 : 0;
                    even_odd <= x ? ODD : EVEN;
                  end
            ODD : begin
                    z <= x ? 0 : 1;
                    even_odd <= x ? EVEN : ODD;
                  end
            default : even_odd <= EVEN;
        endcase
endmodule
```

This design will cause the synthesis tool to generate a latch for the output "even_odd".

Testbench

```
module test_parity;
    reg clk, x;
    wire z;
    parity_gen PAR(x, clk, z);
    initial
        begin
            $dumpfile ("parity.vcd");   $dumpvars(0, testparity);
            clk = 1'b0;
        end
    always #5 clk = ~clk;
    initial
        begin
            #2  x=0; #10 x=1; #10x=1; #10 x=1;
            #10  x=0; #10 x=1; #10x=1; #10 x=0;
            #10 x=0; #10 x=1; #10 x=1; #10 x=0;
            #10 $finish
        end
endmodule
```




From: 0 sec
To: 122 sec
Marker: -- | Cursor: 26 sec
SST
test_parity
PAR
Type
Signals
wire clk
reg even_odd
wire x
reg z
Filter:
Append
Insert
Replace
Signals
Time
clk
x
z
Waves
100 sec

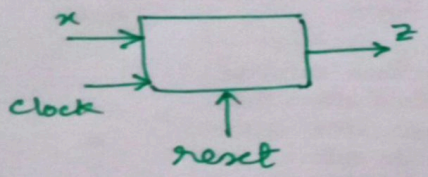
- Design of a sequence detector
  - A circuit accepts a serial bit stream "x" as input and produces a serial bit stream "z" as output.
  - Whenever the bit pattern "0110" appears in the input stream, it outputs z = 1; at all other times, z = 0.
  - Overlapping occurences of the pattern are also detected.
  - This is a Mealy machine

x: 0 1 1 0 1 1 0

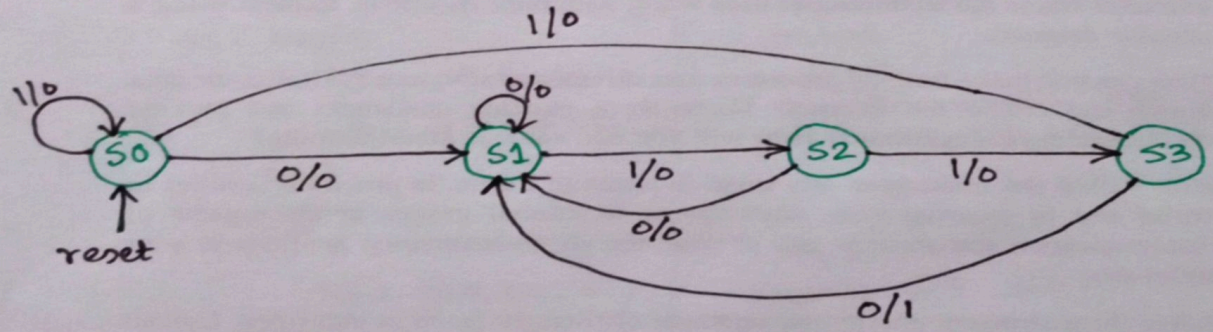
z: 0 0 0 1 0 0 1

- Example: x → 0 1 0 1 0 1 1 0 1 1 0 0 1 0 1 1 0 1 1 1 0

  z → 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0

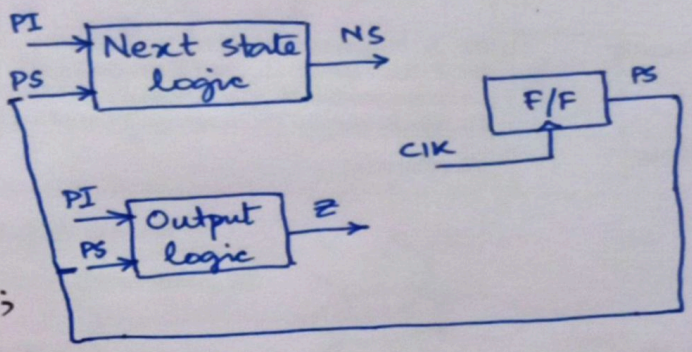


The inputs bits "x" are applied in synchronism with the clock.



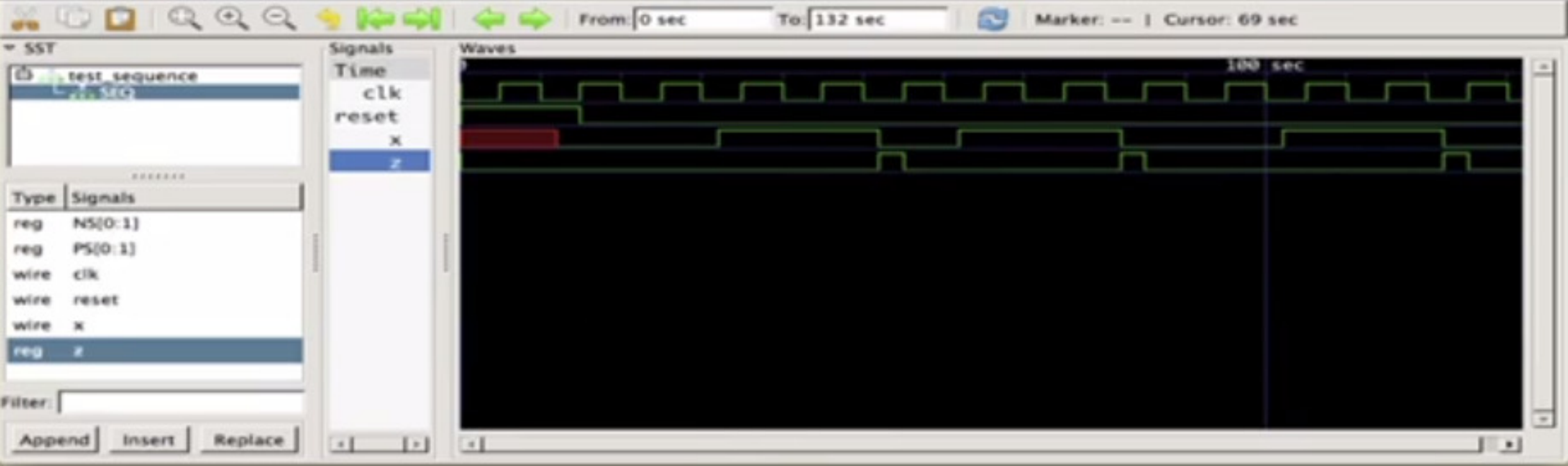
Sequence detector for pattern "0110"

```
module seq_detector (x, clk, reset, z);
    input x, clk, reset;
    output reg z;
    parameter S0=0, S1=1, S2=2, S3=3;
    reg [0:1] PS, NS;
    always @(posedge clk or posedge reset)
        if (reset)
            PS <= S0;
        else
            PS <= NS;
    always @(PS, x)
        case (PS)
            S0: begin
                z = x ? 0 : 0;
                NS = x ? S0 : S1;
                end
            S1: begin
                z = x ? 0 : 0;
                NS = x ? S2 : S1;
                end
            S2: begin
                z = x ? 0 : 0;
                NS = x ? S3 : S1;
                end
            S3: begin
                z = x ? 0 : 1;
                NS = x ? S0 : S1;
                end
        endcase
endmodule
```



Test bench

```
module test_sequence;
    reg clk, x, reset; wire z;
    seq_detector SEQ (x, clk, reset, z);
    initial
      begin
        $dumpfile("sequence.vcd");
        $dumpvars(0, test_sequence);
        clk = 1'b0; reset = 1'b1;
        #15 reset = 1'b0;
      end
    always #5 clk = ~clk;
  initial
    begin
      #12 x=0; #10 x=0; #10 x=1; #10 x=1;
  #10 x=0; #10 x=1; #10 x=1; #10 x=0;
  #10 x=0; #10 x=1; #10 x=1; #10 x=0;
  #10 $finish;
    end
endmodule
```


From: 0 sec
To: 132 sec
Marker: -- | Cursor: 69 sec
SST
test_sequence
Type
Signals
reg NS[0:1]
reg PS[0:1]
wire clk
wire reset
wire x
reg z
Filter:
Append
Insert
Replace
Signals
Time
clk
reset
x
z
Waves
100 sec

Example 4 (Homework)

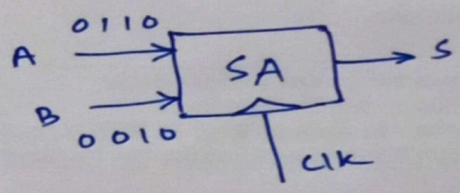
- Design a sequence detector for the bit pattern "101010".
  - Work out the state diagram in a similar way.
  - Then code the state diagram in Verilog.

Serial Adder:

A: 0 1 1 0

B: 0 0 1 0

S: 0 1 0 0 0



State: CARRY (1 bit)

0	1
1	0
1	1

1	1
0	1
1	1
0	1