IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

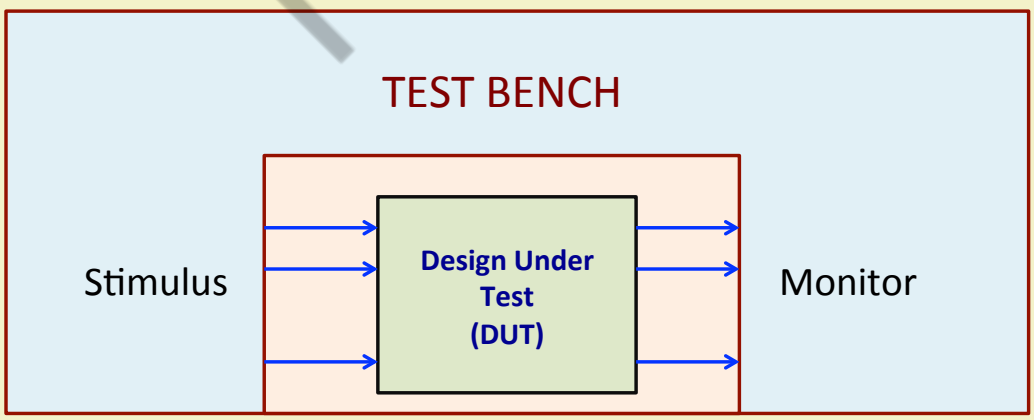
Lecture 21: VERILOG TEST BENCH

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Verilog Test Bench

- What is test bench?
  - A Verilog procedural block that executes only once.
  - Used for simulation.
  - Test bench generates clock, reset, and the required test vectors for a given *design-under-test* (DUT).
  - The test bench can monitor the DUT outputs and present them in a way as specified by the creator.
    - Print the values of the signal lines.
    - Dump the values in a file from where waveforms can be viewed.

- Basic requirements:
  - The inputs of the DUT need to be connected to the test bench.
  - The outputs of the DUT needs also to be connected to the test bench.
- Points to note:
  - Test benches use the "*initial*" procedural block that executes only once.
  - Can also use "*always*" for generating some test inputs, like a clock signal.



A Simple Example

```
module example (A,B,C,D,E,F,Y);
  input A,B,C,D,E,F;
  output Y;
  wire  t1, t2, t3, Y;
  nand  #1 G1 (t1,A,B);
  and   #2 G2 (t2,C,~B,D);
  nor   #1 G3 (t3,E,F);
  nand  #1 G4 (Y,t1,t2,t3);
endmodule
```

```
module  testbench;
  reg  A,B,C,D,E,F;  wire Y;
  example DUT(A,B,C,D,E,F,Y);

  initial
    begin
      $monitor ($time," A=%b, B=%b, C=%b,
        D=%b, E=%b, F=%b, Y=%b",
        A,B,C,D,E,F,Y);
      #5 A=1; B=0; C=0; D=1; E=0; F=0;
      #5 A=0; B=0; C=1; D=1; E=0; F=0;
      #5 A=1; C=0;
      #5 F=1;
      #5 $finish;
    end
endmodule
```


How to write test benches?

- Create a dummy template
  - Declare inputs to the design-under-test (DUT) as "*reg*", and the outputs as "*wire*".
    - Because we have to initialize the DUT inputs inside procedural block(s), typically "*initial*", where only "*reg*" type variables can be assigned.
  - Instantiate the DUT.
- Initialization and Monitoring
  - Assign some known values to the DUT inputs.
  - Monitor the DUT outputs for functional verification.

- For synchronous sequential circuits:
  - We need some clock generation logic.
  - Various ways to specify clock signal.
- Test bench can include various simulator directives:
  - *$display*, *$monitor*, *$dumpfile*, *$dumpvars*, *$finish*, etc.
- Important point:
  - We do not need test bench when we are synthesizing a design.
  - Required only during simulation.

The Simulator Directives

- `$display ("<format>", expr1, expr2, …);`
  - Used to print the immediate values of text or variables to stdout.
  - Syntax is very similar to "printf" in C.
  - Additional format specifiers are supported, like "b" (binary), "h" (hexadecimal), etc.
- `$monitor ("<format>", var1, var2, …);`
  - Similar in syntax to *$display*, but does not print immediately.
  - It will print the value(s) whenever the value of some variable(s) in the given list changes.
  - Has the functionality of *event-driven* print.

- `$finish;`
  - Terminates the simulation process.
- `$dumpfile (<filename>);`
  - Specifies the file that will be used for storing the values of the selected variables so that they can be graphically visualized later.
  - The file typically has an extension *.vcd (Value Change Dump)*, and contains information about any value changes on the selected variables.
- `$dumpoff;`
  - This directive stops the dumping of variables. All variables are dumped with "x" values and the next change of variables will not be dumped.
- `$dumpon;`
  - This directive starts previously stopped dumping of variables.

- `$dumpvars (level, list_of_variables_or_modules);`
  - Specifies which variables should be dumped to the *.vcd* file.
  - Both the parameters are optional; if both are omitted, all variables are dumped.
  - If *level=0*, then all variables within the modules from the list will be dumped. If any module from the list contains module instances, then all variables from these modules will also be dumped.
  - If *level=1*, then only listed variables and variables of listed modules will be dumped.
- `$dumpall;`
  - The current values of all variables will be written to the file, irrespective of whether there has been any change in their values or not.
- `$dumplimit (filesize);`
  - Used to set the maximum size of the .vcd file.

A Complete Example :: 2-bit equality checker

```
`timescale 1ns / 100ps
module comparator (x, y, z);
  input [1:0] x, y;  output z;
  assign z = (x[0]&y[0]&x[1]&y[1])    |
             (~x[0]&~y[0]&x[1]&y[1])   |
             (~x[0]&~y[0]&~x[1]&~y[1]) |
             (x[0]&y[0]&~x[1]&~y[1]);
endmodule
```



```
`timescale 1ns / 100ps
module testbench;
  reg [1:0] x, y;  wire z;
  comparator C2 (.x(x), .y(y), .z(z));
  initial
  begin
    $dumpfile ("comp.vcd");
    $dumpvars (0, testbench);
    x = 2'b01; y = 2'b00;
    #10 x = 2'b10; y = 2'b10;
    #10 x = 2'b01; y = 2'b11;
  end
  initial
    begin
      $monitor ("t=%d x=%2b y=%2b z=%d", $time, x, y, z);
    end
endmodule
```

END OF LECTURE 21

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Lecture 22: WRITING VERILOG TEST BENCHES

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Writing Test Benches

- We shall be illustrating the process of writing test benches through a number of examples.
- We shall be looking at how to:
  - Write test benches for combinational designs.
  - Write test benches for sequential designs.
  - Generate clock and synchronize the applied inputs.
  - Automatically verifying the outputs generated by the design under test.
  - Generating random test vectors.

Example 1: Full Adder

```
module full_adder (s, co, a, b, c);
  input a, b, c;
  output s, co;
  assign  s = a ^ b ^ c;
  assign  co = (a & b) | (b & c) | (c & a);
endmodule
```

```
module  testbench;
  reg  a, b, c;  wire sum, cout;
  full_adder FA (sum, cout, a, b, c);

  initial
    begin
      $monitor ($time," a=%b, b=%b, c=%b, sum=%b, cout=%b",
                  a, b, c, sum, cout);
      #5 a=0; b=0; c=1;
      #5 b=1;
      #5 a=1;
      #5 a=0; b=0; c=0;
      #5 $finish;
    end
endmodule
```

```
 0 a=x, b=x, c=x, sum=x, cout=x
 5 a=0, b=0, c=1, sum=1, cout=0
10 a=0, b=1, c=1, sum=0, cout=1
15 a=1, b=1, c=1, sum=1, cout=1
20 a=0, b=0, c=0, sum=0, cout=0
```



```
module  testbench;
  reg  a, b, c;  wire sum, cout;
  full_adder FA (sum, cout, a, b, c);

  initial
    begin
      a=0; b=0; c=1; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b",$time,a,b,c,sum,cout);
      b=1; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b",$time,a,b,c,sum,cout);
      a=1; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b",$time,a,b,c,sum,cout);
      a=0; b=0; c=0; #5;
      $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b",$time,a,b,c,sum,cout);
      #5 $finish;
    end
endmodule
```

```
T= 5, a=0, b=0, c=1, sum=1, cout=0
T=10, a=0, b=1, c=1, sum=0, cout=1
T=15, a=1, b=1, c=1, sum=1, cout=1
T=20, a=0, b=0, c=0, sum=0, cout=0
```

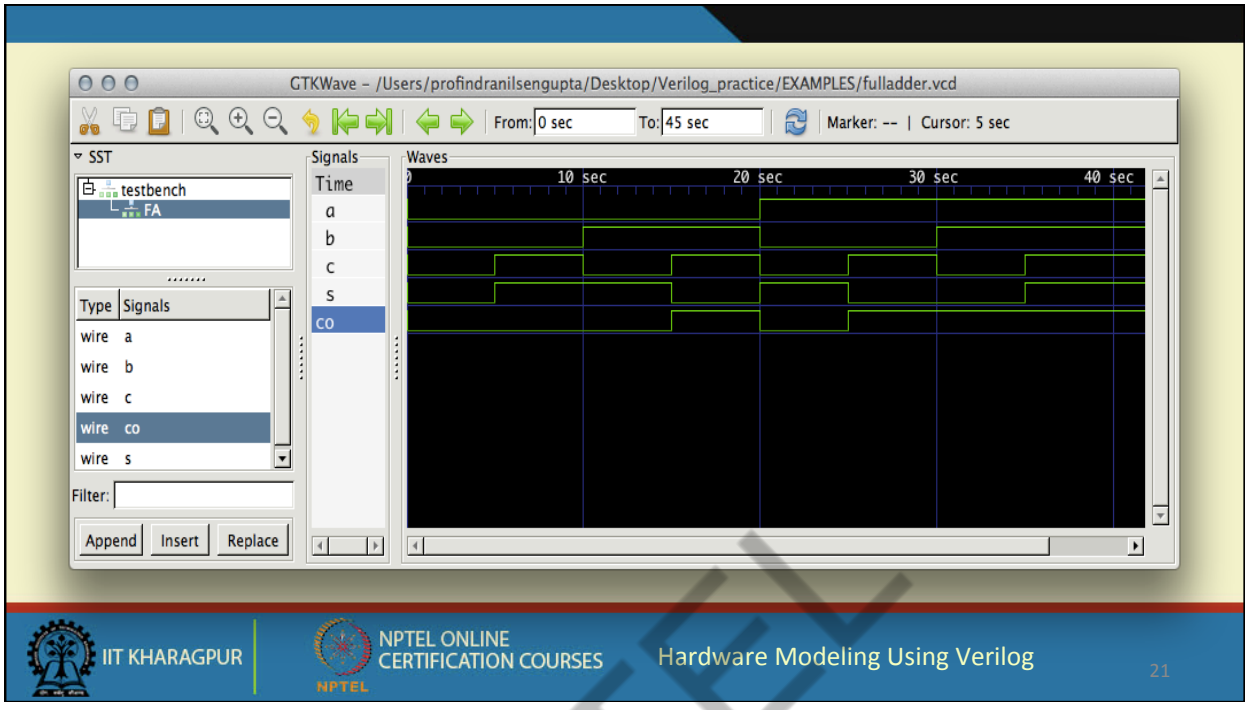
```
module  testbench;
  reg  a, b, c;  wire sum, cout;
  integer i;
  full_adder FA (sum, cout, a, b, c);

  initial
    begin
      for (i=0; i<8; i=i+1)
       begin
         {a,b,c} = i;  #5;
         $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b",
                        $time, a, b, c, sum, cout);
       end
      #5 $finish;
    end
endmodule
```

```
T= 5, a=0, b=0, c=0, sum=0, cout=0
T=10, a=0, b=0, c=1, sum=1, cout=0
T=15, a=0, b=1, c=0, sum=1, cout=0
T=20, a=0, b=1, c=1, sum=0, cout=1
T=25, a=1, b=0, c=0, sum=1, cout=0
T=30, a=1, b=0, c=1, sum=0, cout=1
T=35, a=1, b=1, c=0, sum=0, cout=1
T=40, a=1, b=1, c=1, sum=1, cout=1
```



```
module  testbench;
  reg  a, b, c;  wire sum, cout;
  integer i;
  full_adder FA (sum, cout, a, b, c);

  initial
    begin
      $dumpfile ("fulladder.vcd");
      $dumpvars (0, testbench);
      for (i=0; i<8; i=i+1)
       begin
         {a,b,c} = i;  #5;
         $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b",
                        $time, a, b, c, sum, cout);
       end
      #5 $finish;
    end
endmodule
```

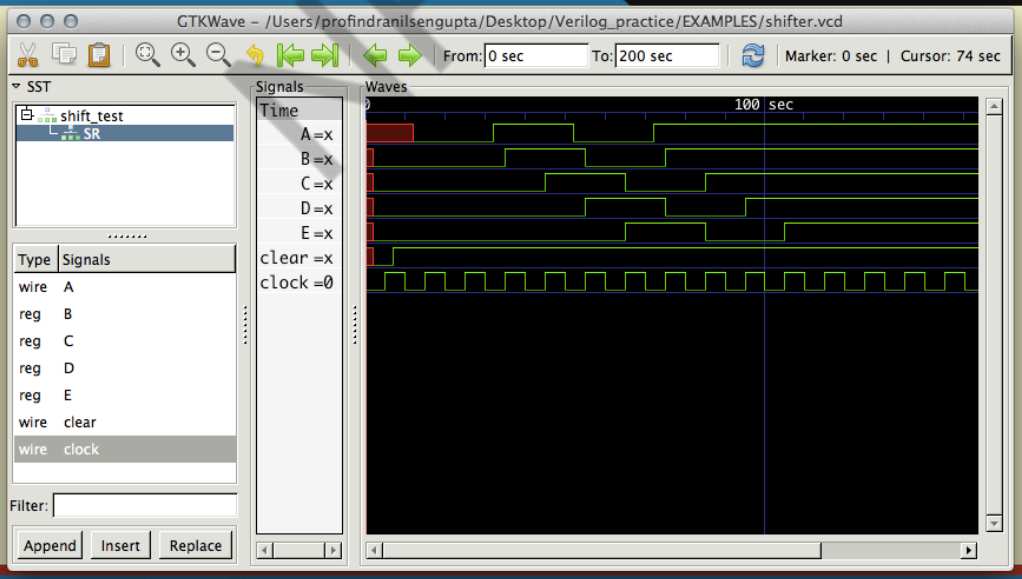
```
T= 5, a=0, b=0, c=0, sum=0, cout=0
T=10, a=0, b=0, c=1, sum=1, cout=0
T=15, a=0, b=1, c=0, sum=1, cout=0
T=20, a=0, b=1, c=1, sum=0, cout=1
T=25, a=1, b=0, c=0, sum=1, cout=0
T=30, a=1, b=0, c=1, sum=0, cout=1
T=35, a=1, b=1, c=0, sum=0, cout=1
T=40, a=1, b=1, c=1, sum=1, cout=1
```



Example 2: 4-bit shift register

```
module shiftreg_4bit (clock, clear, A, E);
  input clock, clear, A;
  output reg E;
  reg B, C, D;
  always @(posedge clock or negedge clear)
    begin
      if (!clear) begin B<=0; C<=0; D<=0; E<=0; end
      else begin
             E <= D;
             D <= C;
             C <= B;
             B <= A;
           end
    end
endmodule
```

```
module  shift_test;
  reg  clk, clr, in;    wire out;   integer i;
  shiftreg_4bit SR (clk, clr, in, out);

  initial
    begin clk = 1'b0; #2 clr = 0; #5 clr = 1;  end

  always  #5 clk = ~clk;

  initial  begin  #2;
      repeat (2)
        begin #10 in=0; #10 in=0; #10 in=1; #10 in=1; end
    end

  initial
    begin
      $dumpfile ("shifter.vcd");
      $dumpvars (0, shift_test);
      #200 $finish;
    end
endmodule
```

Example 3: 7-bit binary counter

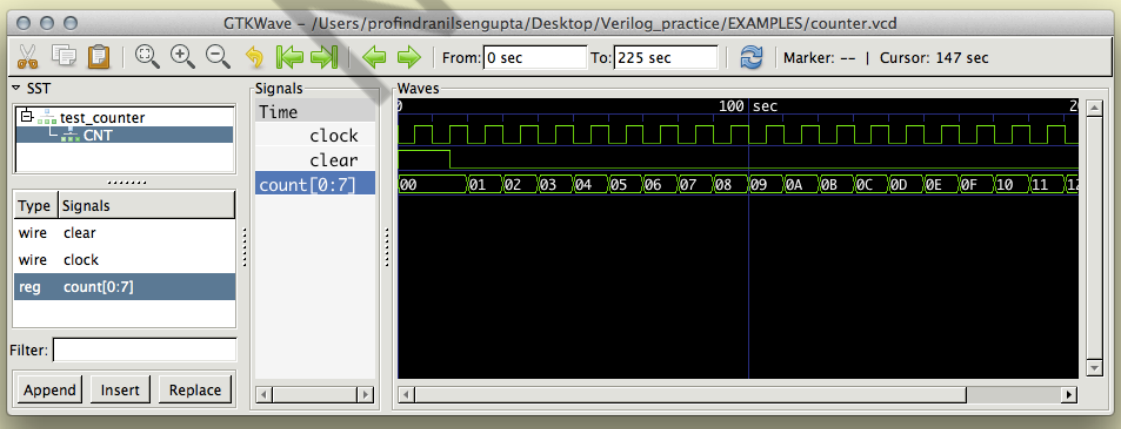
```
module  counter (clear, clock, count);
    parameter  N = 7;
    input  clear, clock;
    output reg [0:N] count;

    always @(negedge  clock)
       if (clear)
          count <= 0;
       else
          count <= count + 1;
endmodule
```



```
module test_counter;
  reg clk, clr;
  wire [7:0] out;

  counter CNT (clr, clk, out);

  initial  clk = 1'b0;

  always  #5 clk = ~clk;

  initial
    begin
      clr = 1'b1;
      #15 clr = 1'b0;
      #200 clr = 1'b1;
      #10 $finish;
    end
```

```
  initial
    begin
      $dumpfile ("counter.vcd");
      $dumpvars (0, test_counter);
      $monitor ($time, " Count: %d", out);
    end

endmodule
```

```
  0 Count:   0          140 Count:  13
 20 Count:   1          150 Count:  14
 30 Count:   2          160 Count:  15
 40 Count:   3          170 Count:  16
 50 Count:   4          180 Count:  17
 60 Count:   5          190 Count:  18
 70 Count:   6          200 Count:  19
 80 Count:   7          210 Count:  20
 90 Count:   8          220 Count:   0
100 Count:   9
110 Count:  10
120 Count:  11
130 Count:  12
```

Example 4: Automatic verification of output

```
module fulladder (a, b, c, s, cout);
  input a, b, c;
  output s, cout;

  assign s = a ^ b ^ c;

  assign cout = (a&b) | (b&c) | (c&a);

endmodule
```



```
module fulladder_test;

reg a,b,c;
wire s, cout;
integer correct;

fulladder FA (a,b,c,s,cout);

initial
  begin
    correct = 1;

    #5 a=1; b=1; c=0; #5;
       if ((s != 0) || (cout != 1))
          correct = 0;
```

```
    #5 a=1; b=1; c=1; #5;
       if ((s != 1) || (cout != 1))
           correct = 0;

    #5 a=0; b=1; c=0; #5;
       if ((s != 1) || (cout != 0))
           correct = 0;

    #5 $display ("%d", correct);
  end

endmodule
```

Shall display 1 if outputs are correct; and display 0 otherwise.

Example 5: Generating random test vectors

```
module adder (out, cout, a, b);
  input [7:0] a, b;
  output [7:0] out;
  output cout;

  assign #5 {cout,out} = a + b;
endmodule
```

- The system task *$random* can be used to generate a random number.
- It is called as : *$random (<seed>)*
  - The value of *<seed>* is optional and is used to ensure that the same sequence of random numbers are generated each time the test is run.


```
module test_adder;
  reg [7:0] a, b;
  wire [7:0] sum;  wire cout;
  integer myseed;
  adder ADD (sum, cout, a, b);
  initial  myseed = 15;
  initial
    begin
      repeat (5)
        begin
          a = $random(myseed);
          b = $random(myseed); #10;
          $display ("T: %3d, a: %h, b: %h, sum: %h", $time, a, b, sum);
        end
    end
endmodule
```

```
T:  10, a: 00, b: 52, sum: 52
T:  20, a: ca, b: 08, sum: d2
T:  30, a: 0c, b: 6a, sum: 76
T:  40, a: b1, b: 71, sum: 22
T:  50, a: 23, b: df, sum: 02
```

END OF LECTURE 22

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Lecture 23: MODELING FINITE STATE MACHINES

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Introduction

- 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.

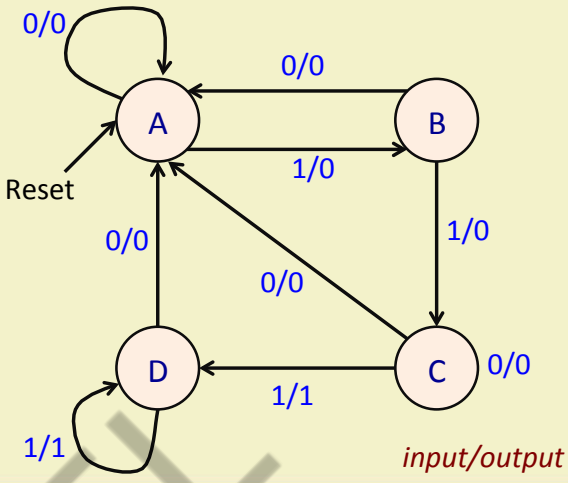
Finite State Machine (FSM)

- A FSM can be represented either in the form of a *state table* or in the form of a *state transition diagram*.
  - Variations exist, e.g. *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 output 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

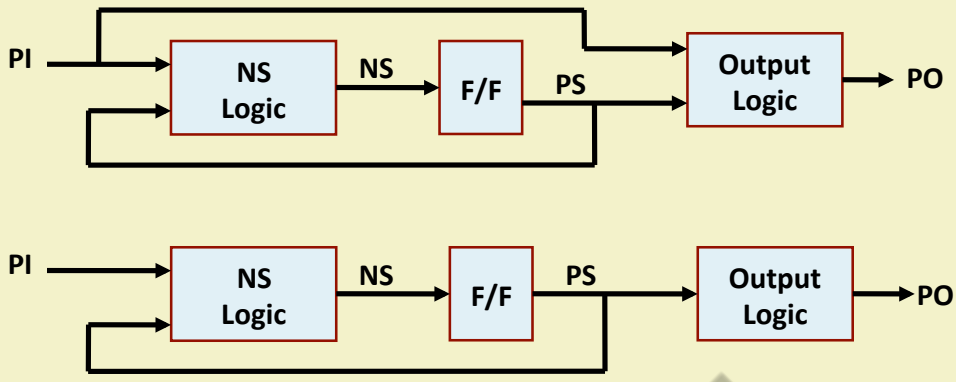
State Diagram



Mealy and Moore FSM Types

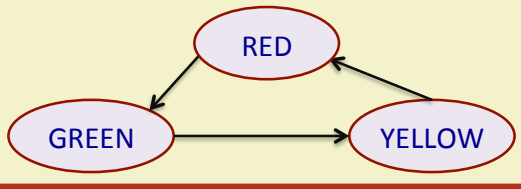
- A deterministic FSM can be mathematically defines as a 5-tuple
  $(\Sigma, \Gamma, S, s_0, \delta, \omega)$
  where Σ is the set of input combinations, Γ is the set of output combinations, S is a finite set of states, $s_0 \in S$ is the initial state, δ is the state-transition function, and ω is the output function.
- Here, $\delta : S \times \Sigma \rightarrow S$
  - 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



Example 1

- There are three lamps, RED, GREEN and YELLOW, that should glow cyclically with a fixed time interval (say, 1 second).
- 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  clk;
  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          // S2 means YELLOW
            light <= RED;  state <=  S0;
          end
```

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



```
module test_cyclic_lamp;
  reg clk;
  wire [0:2] light;
  cyclic_lamp LAMP (clk, 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 flip-flops – 2 for *state*, and 3 for *light*.
  - The three output lines are also getting stored in flip-flops.
    - We have used non-blocking assignment triggered by clock edge.
  - But actually we do not need separate flip-flops 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  clk;
  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  clk)
    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
```

- Comment on the solution:
  - The synthesis tool will be generating only 2 flip-flops 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 outputs.

state (s_1s_0)	Light (RGY)
S0: 00	1 0 0
S1: 01	0 1 0
S2: 10	0 0 1
11	x x x

Logic expressions after minimization:

$R = s_0'.s_1'$

$G = s_0$

$Y = s_1$

END OF LECTURE 23

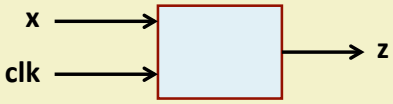
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

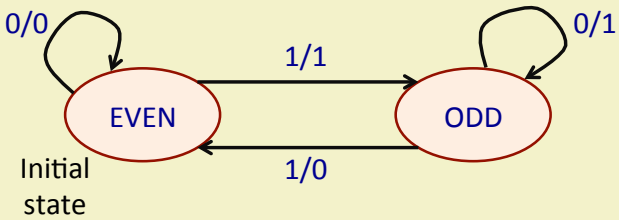
Lecture 24: MODELING FINITE STATE MACHINES (contd.)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Example 2

- Design of a serial parity detector.
  - A continuous stream of bits is fed to a circuit in synchronism 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*.

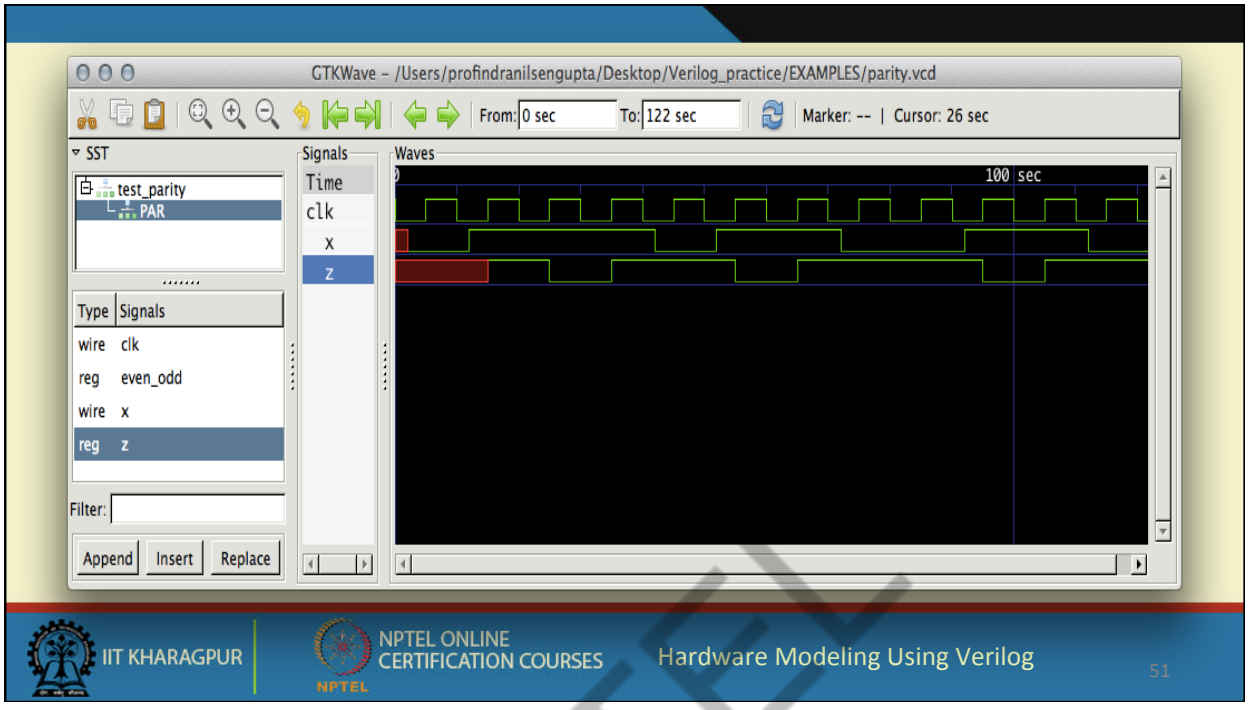




```
module parity_gen (x, clk, z);
  input  x, clk;
  output reg z;
  reg  even_odd;       // The machine state
  parameter  EVEN=0, ODD=1;
  always @(posedge  clk)
    case (even_odd)
      EVEN: begin
              z <= x ? 1 : 0;
              even_odd <= x ? ODD : EVEN;
            end
      ODD:  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*".


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



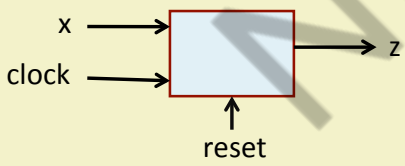

```
module parity_gen (x, clk, z);
  input  x, clk;   output reg z;
  reg  even_odd;      // The machine state
  parameter  EVEN=0, ODD=1;
  always @(posedge  clk)
    case (even_odd)
      EVEN: even_odd <= x ? ODD : EVEN;
      ODD:  even_odd <= x ? EVEN : ODD;
      default : even_odd <= EVEN;
    endcase
  always @(even_odd)
    case (even_odd)
      EVEN: z = 0;
      ODD:  z = 1;
    endcase
endmodule
```

This design will not cause the synthesis tool to generate a latch for the output "*z*".

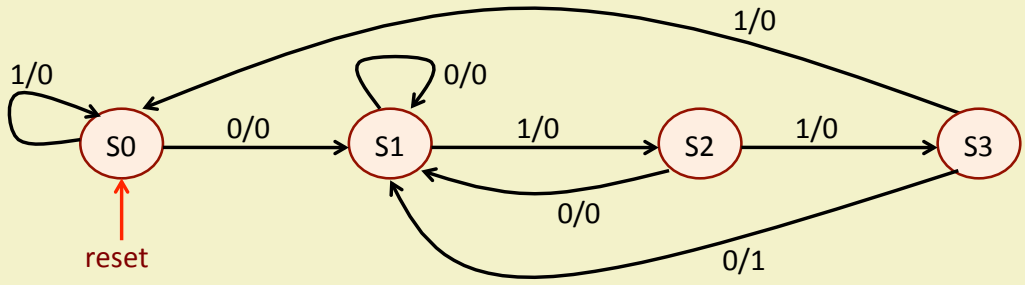
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
Hardware Modeling Using Verilog
52

Example 3

- 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 occurrences of the pattern are also detected.
  - This is a *Mealy Machine*.
  - 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



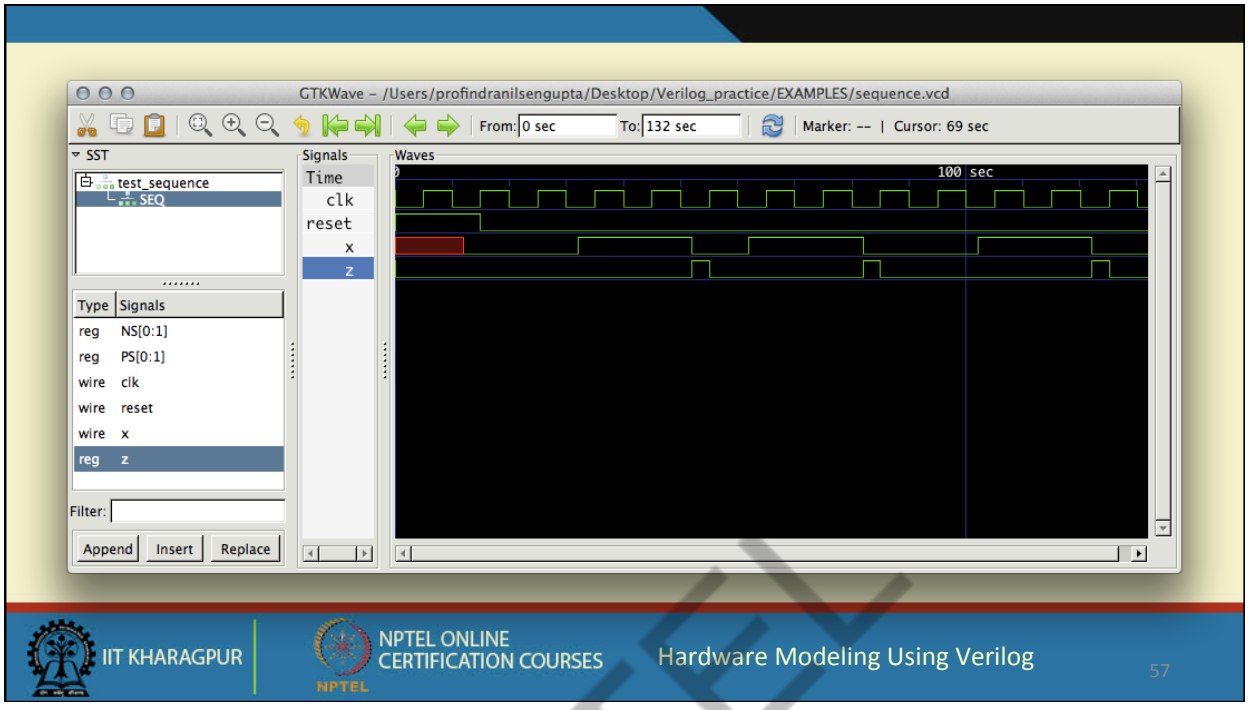
The input bits "x" are applied in synchronism with the clock.



```
// 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
```



```
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
```



Example 4

- 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.

END OF LECTURE 24