Verilog HDL – Part II

044252 - Digital Systems and Computer Structure

Topics Covered Last Time

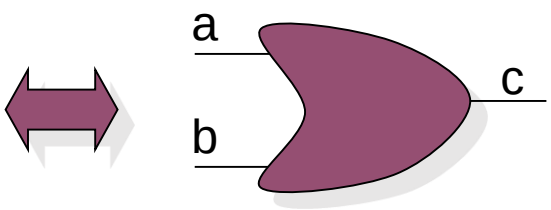
- Hardware Description Languages and coding styles
- Verilog Building Blocks:
  - Keywords
  - Modules
  - Values / Number representation
  - Data Types
  - Operators
- Assignments and procedural blocks
- Test benches and simulation
- Combinational example

Continuous Assignments - Recap

Wire/Vector assignment are “continuous assignments”

- Connect combinational logic block or other wire to wire input
- When right-hand-side changes, it immediately flows through to left
- **Order of statements not important to Verilog**, executed totally in parallel
- Designated by the keyword **assign**

```
wire c;
assign c = a | b;
wire c = a | b; //same thing
```



- Common errors:
  - Not assigning a wire a value
  - Assigning a wire a value more than once

Procedural Blocks

- Two Procedural Constructs
  - initial Statement - Executes only once
  - always Statement - Executes in a loop
- Sequential: All statements within the block are executed sequentially
- Assign variables using procedural assignments
- Example

```
...
initial begin
  Sum = 0;
  Carry = 0;
end
...
```

```
...
always ... begin
  Sum = A ^ B;
  Carry = A & B;
end
...
```

Agenda

- Procedural Execution Control
- Procedural Statement Constructs
- Understanding Simulator Behavior
- Final example - FSM

Execution Control

- Initial – on simulation start. May be delayed using # (more to come)
- Always – execution triggered on:
  - Occurrence of an event in the sensitivity list
  - Event: Change in the logical value

```
always @ (a or b or sel)
begin
   if (sel) out = a;
   else out = b;
end
```

Executes whenever signals in sensitivity list change.

Sensitivity list should include conditional (sel) and right side (a, b) assignment variables.

  - Sensitivity list could also be: always @(*)

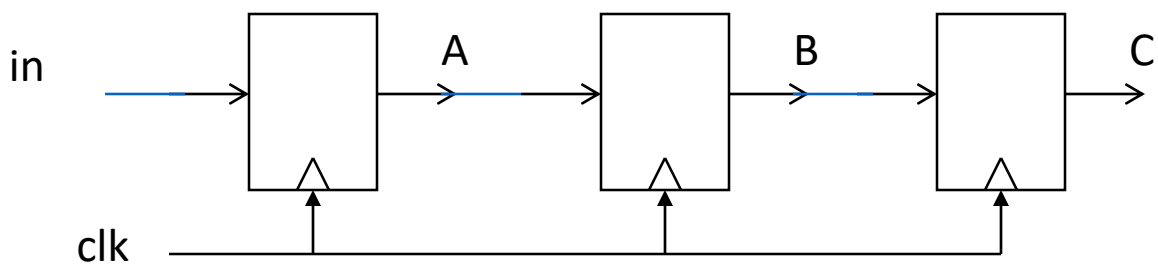
Procedural Assignments

Drive values onto registers (vector and scalar)

- Occur within procedures such as *always* and *initial*
- Triggered when the flow of execution reaches them
- Blocking/Non-blocking
  - **Blocking** assignment statement (**=** operator) acts much like in traditional programming languages. The whole statement is done before control passes on to the next statement
  - **Non-blocking** assignment statement (**<=** operator) evaluates all the right-hand sides for the current time unit and assigns the left-hand sides at the end of the time unit

Example - Blocking

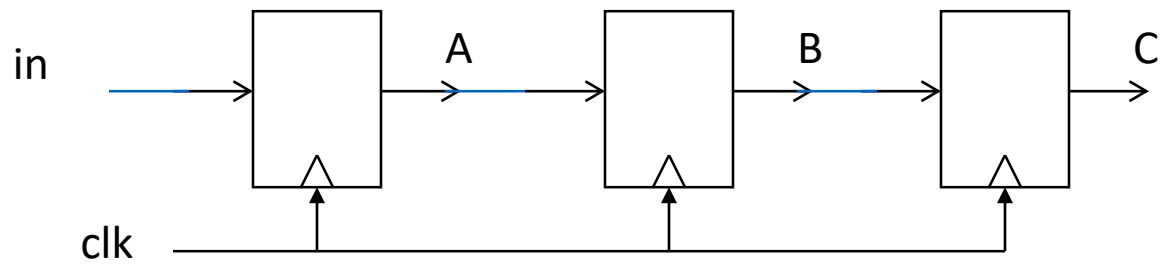
```
module shifter (in, A,B,C,clk);
  input in, clk;
  input A,B,C;
  reg A, B, C;
  always @ (posedge clk) begin
    C = B;
    B = A;
    A = in;
  end
endmodule
```



- Block interpreted sequentially, but action happens “at once”

Example – Non blocking

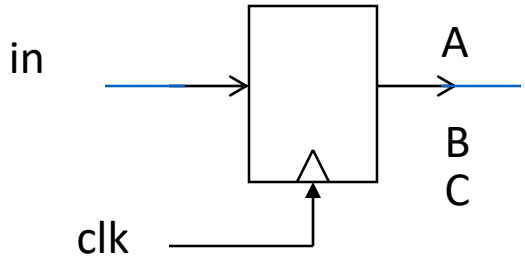
```
module shifter (in, A,B,C,clk);
  input in, clk;
  input A,B,C;
  reg A, B, C;
  always @ (posedge clk) begin
    B <= A;
    A <= in;
    C <= B;
  end
endmodule
```



- Non-blocking: all statements interpreted in parallel
  - Everything on the RHS evaluated,
  - Then all assignments performed

Example – Blocking Reordered

```
module shifter (in, A,B,C,clk);
  input in, clk;
  input A,B,C;
  reg A, B, C;
  always @ (posedge clk) begin
    A = in;
    B = A;
    C = B;
  end
endmodule
```



“Complete” Assignments

- If an `always` block executes, and a variable is *not* assigned
  - Variable keeps its old value (think implicit state!)
  - *NOT* combinational logic $\Rightarrow$ latch is inserted (implied memory)
  - This is usually *not* what you want: dangerous for the novice!
- Any variable assigned in an `always` block should be assigned for any (and every!) execution of the block.

Incomplete Triggers

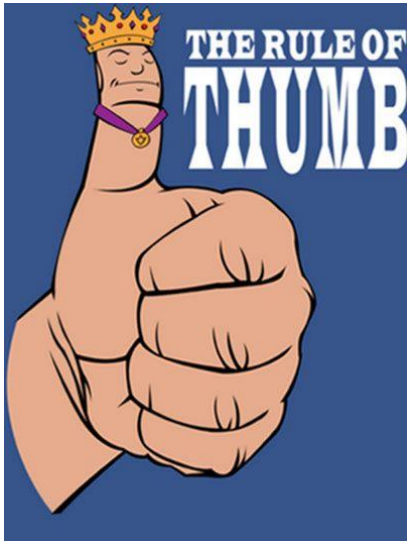
- Leaving out an input trigger usually results in a sequential circuit
- Example: The output of this “and” gate depends on the input history

```
module and_gate (out, in1, in2);
  input    in1, in2;
  output       out;
  reg          out;

  always @(in1) begin
     out = in1 & in2;
  end
endmodule
```

Procedural Assignments Summary

- **Combinational logic**: Use **blocking statements** inside always blocks with @(*) to mimic logic flow of combinational logic.
- **Sequential logic**: Use **non-blocking statements** with always blocks sensitive to rising clock edge to mimic parallel sequential logic.



Agenda

- Procedural Execution Control
- Procedural Statement Constructs
- Understanding Simulator Behavior
- Final example - FSM

Loop Statements

- Loop Statements
  - Repeat
  - While
  - For

- Repeat Loop
  - Example:

```
repeat (Count)
  sum = sum + 5;
```

  - If the condition is evaluated to be ‘X’ or ‘Z’ it is treated as if it were ‘0’.

Loop Statements (cont.)

- While Loop
  - Example:

```
while (Count < 10) begin
  sum = sum + 5;
  Count = Count +1;
end
```

  - If the condition is evaluated to be ‘X’ or ‘Z’ it is treated as if it were ‘0’.

- For Loop
  - Example:

```
for (Count = 0; Count < 10; Count = Count + 1) begin
  sum = sum + 5;
end
```

Conditional Statements

- if Statement
- Format:

```
if (condition)
   procedural_statement
else if (condition)
   procedural_statement
else
   procedural_statement
```

- Example:

```
if (Clk)
  Q = 0;
else
  Q = D;
```

Conditional Statements (cont.)

- Case Statement
- Example 1:

```
case (X)
  2'b00: Y = A + B;
  2'b01: Y = A – B;
  2'b10: Y = A / B;
endcase
```

Prone to partial assignment issues

- Example 2:

```
case (3'b101 << 2)
  3'b100: A = B + C;
  4'b0100: A = B – C;
  5'b10100: A = B / C; //This statement is executed
endcase
```

Conditional Statements (cont.)

- Variants of case Statements:
  - casex and casez
- casez – z is considered as a don't care
- casex – both x and z are considered as don't cares
- Example:

```
casez (X)
  2'b1z: A = B + C;
  2'b11: A = B / C;
endcase
```

Delay based Timing Control

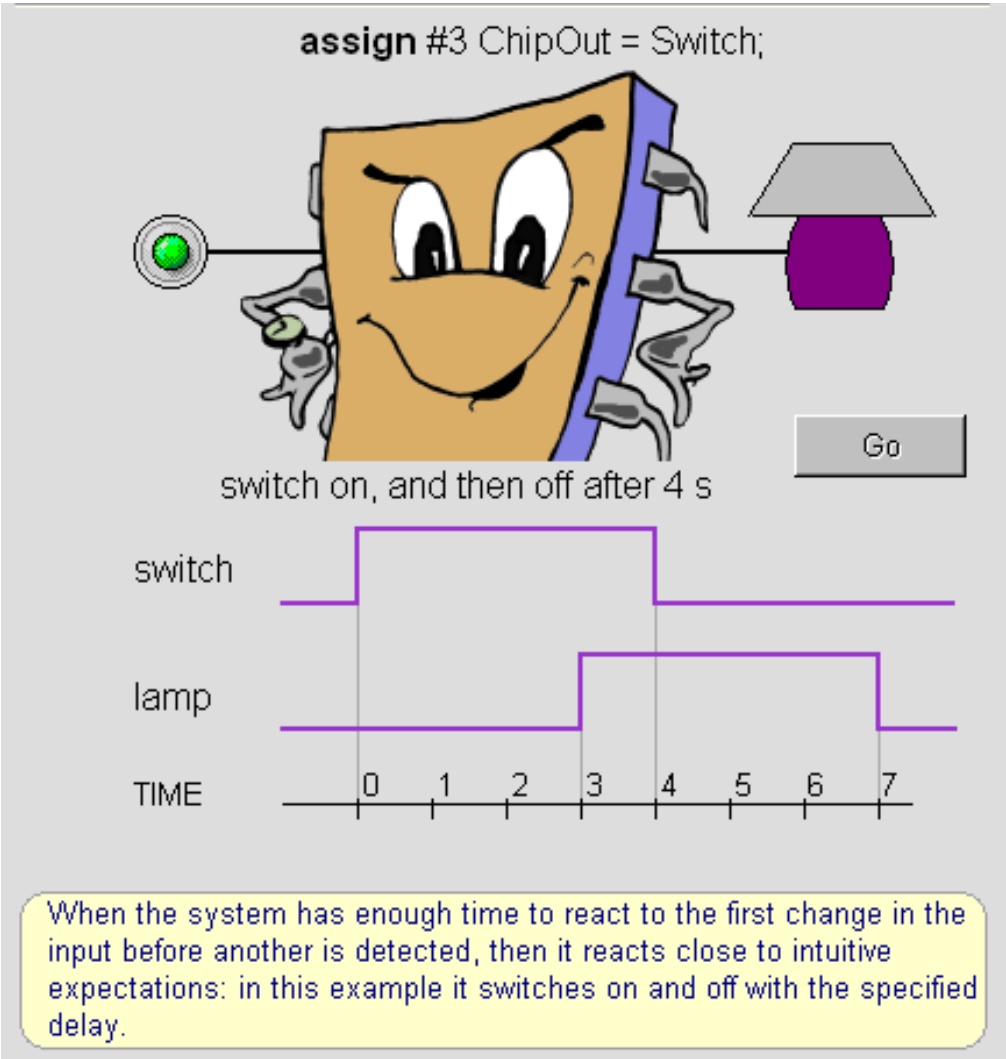
- **Delay Control (#)**
  - Expression specifies the time duration between initially encountering the statement and when the statement actually executes.
  - Delay in Procedural Assignments
    - Inter-Statement Delay
    - Intra-Statement Delay
  - For example:
    - Inter-Statement Delay

```
#10 A = A + 1;
```

    - Intra-Statement Delay

```
A = #10 A + 1;
```

Delay Statement



Agenda

- Procedural Execution Control
- Procedural Statement Constructs
- Understanding Simulator Behavior
- Final example - FSM

Simulation Behavior

- Scheduled using an event queue
- Non-preemptive, no priorities
- Events at a particular time unordered

- Scheduler runs each event at the current time, possibly scheduling more as a result

Simulation Behavior

- Concurrent processes (initial, always) run until they stop at one of the following

- #42
  - Schedule process to resume 42 time units from now
- wait(cf & of)
  - Resume when expression “cf & of” becomes true
- @(a or b or y)
  - Resume when a, b, or y changes
- @(posedge clk)
  - Resume when clk changes from 0 to 1

Simulation Behavior

- Infinite loops are possible and the simulator does not check for them
- This runs forever: no context switch allowed, so ready can never change

```
while (~ready)
  count = count + 1;
```

- Instead, use

```
wait(ready);
```

Simulation Behavior

- Race conditions abound in Verilog

- These can execute in either order: final value of a undefined:

```
always @(posedge clk) a = 0;
always @(posedge clk) a = 1;
```

Agenda

- Procedural Execution Control
- Procedural Statement Constructs
- Understanding Simulator Behavior
- Final example - FSM

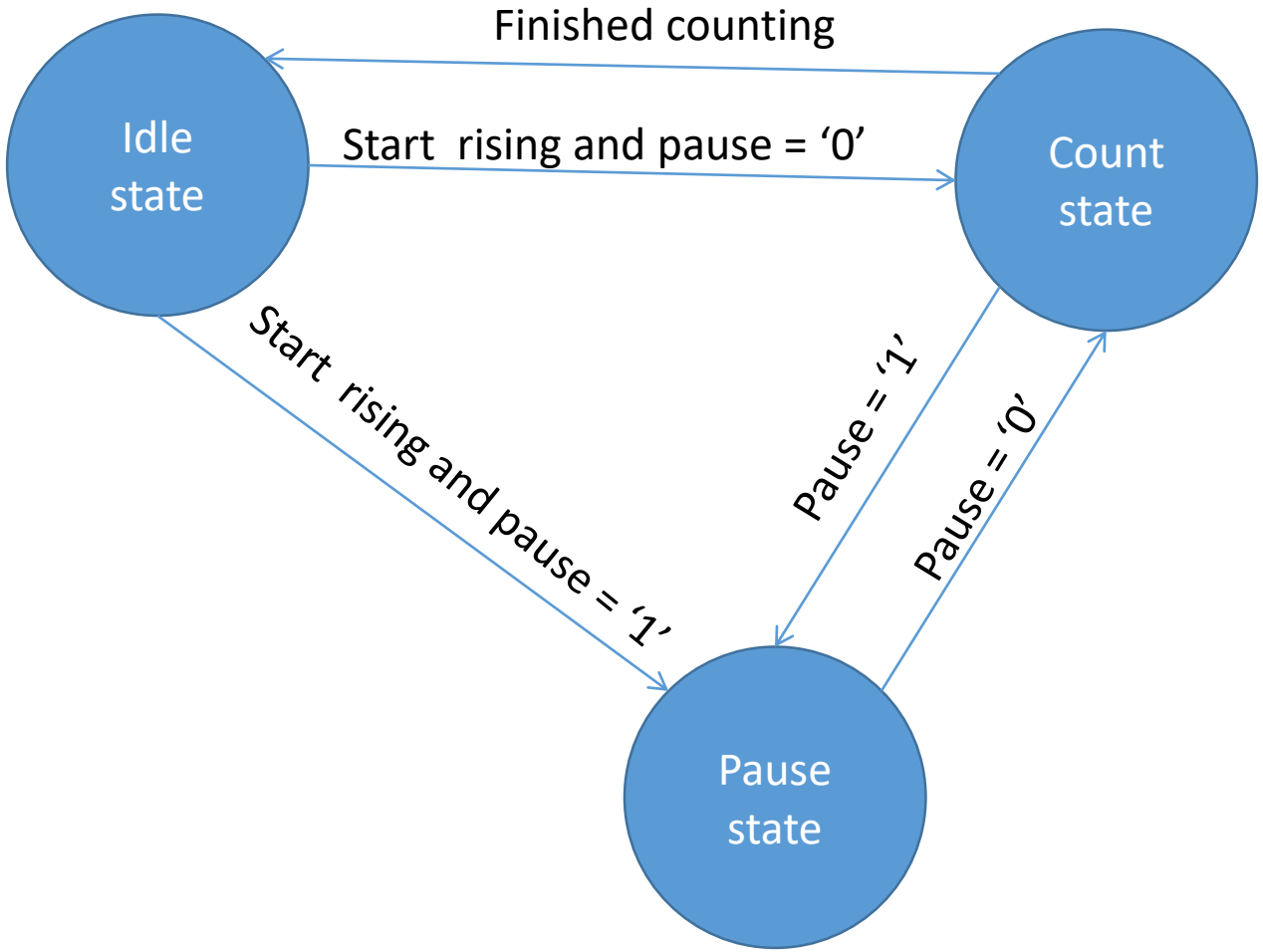
Final Example - FSM

- Designing a smart counter
- Requirements
  - Receives a start signal.
  - When start signal goes high, starts to count from zero: reset value – 0, first clock after start – Out = 1, second clock after start- 2 etc.
  - Counts until a user defined value. When finished, issues a Finish pulse and waits for a new start.
  - Can receive a pause signal active high. In this case, will wait until pause goes down, and then continue

FSM-Top Level

```
7   module smart_cnt(clk,rst,start,pause,data_in,data_out,finish_cnt);
8
9   //-----------------------------------------------------------------------------
10  // parameter list
11  //-----------------------------------------------------------------------------
12      parameter G_DATA_WIDTH = 8;
13
14  //-----------------------------------------------------------------------------
15  // port list
16  //-----------------------------------------------------------------------------
17  // Clock & Reset
18      input                                   clk;
19      input                                   rst;
20  // Inputs
21      input                                   start;
22      input                                   pause;
23      input   [G_DATA_WIDTH-1:0]              data_in;
24  // Outputs
25      output  [G_DATA_WIDTH-1:0]              data_out;
26      output  reg                             finish_cnt;
27
```

FSM - Design


Finished counting
Idle state
Start rising and pause = '0'
Count state
Start rising and pause = '1'
Pause = '1'
Pause = '0'
Pause state

FSM – Writing Code

- Declare states used for FSM
- FSM divided into two processes:
  - Synchronous – saves next state
  - Asynchronous – decides where to go, control signals
- We use other processes for help

FSM – Declaration and Sync Process

```
//-----------------------------------------------------------------------------
// Signal Declerations
//-----------------------------------------------------------------------------
    localparam IDLE_ST  = 2'd0;
    localparam CNT_ST   = 2'd1;
    localparam PAUSE_ST = 2'd2;

    reg                                     start_d;
    reg [1:0]                               nxt_st, cur_st;
    reg [G_DATA_WIDTH-1:0]                  cnt, cnt_ff, max_cnt;
```

```
// Synchronous Process
    always @(posedge clk, posedge rst)
    begin: fsm_sync_proc
        if (rst == 1'b1)
            cur_st <= IDLE_ST;
        else
            cur_st <= nxt_st;
    end // end fsm_sync_proc
```

FSM – Aiding process

```
52  // Aiding Process
53      always @(posedge clk, posedge rst)
54      begin: ff_proc
55          if (rst == 1'b1) begin
56              start_d <= 1'b0;
57              cnt_ff  <= {G_DATA_WIDTH{1'b0}};
58              max_cnt <= {G_DATA_WIDTH{1'b1}};
59          end
60          else begin
61              start_d <= start;
62              cnt_ff  <= cnt;
63              if( (start == 1'b1) && (start_d == 1'b0) )
64                  max_cnt <= data_in;
65          end
66      end // end ff_proc
67
```

FSM- Async Process

```
68  // Asynchronous Process
69      always @(*)
70      begin: fsm_async_proc
71          cnt = cnt_ff;
72          finish_cnt = 1'b0;
73          case(cur_st)
74              IDLE_ST: begin
75                  if ( (start == 1'b1) && (start_d == 1'b0) )
76                      if (pause == 1'b1)
77                          nxt_st = PAUSE_ST;
78                      else
79                          nxt_st = CNT_ST;
80                  else
81                      nxt_st = IDLE_ST;
82              end // end IDLE_ST
83
84              PAUSE_ST: begin
85                  if (pause == 1'b1)
86                      nxt_st = PAUSE_ST;
87                  else begin
88                      nxt_st = CNT_ST;
89                      cnt = cnt_ff+1;
90                  end
91              end // end PAUSE_ST
```

Defaults

Assignment at every branch

FSM- Async Process-continued

```
93              CNT_ST: begin
94                  if (pause == 1'b1)
95                      nxt_st = PAUSE_ST;
96                  else if (cnt_ff == max_cnt) begin
97                      cnt = {G_DATA_WIDTH{1'b0}};
98                      nxt_st = IDLE_ST;
99                      finish_cnt = 1'b1;
100                 end
101                 else begin
102                     cnt = cnt_ff+1;
103                     nxt_st = CNT_ST;
104                 end
105             end // end CNT_ST
106         endcase
107     end // end fsm_async_proc
108
```

- Assigning a reg value to a wire:

```
109     assign data_out = cnt_ff;
110
111
112 endmodule
```

FSM- Test Bench

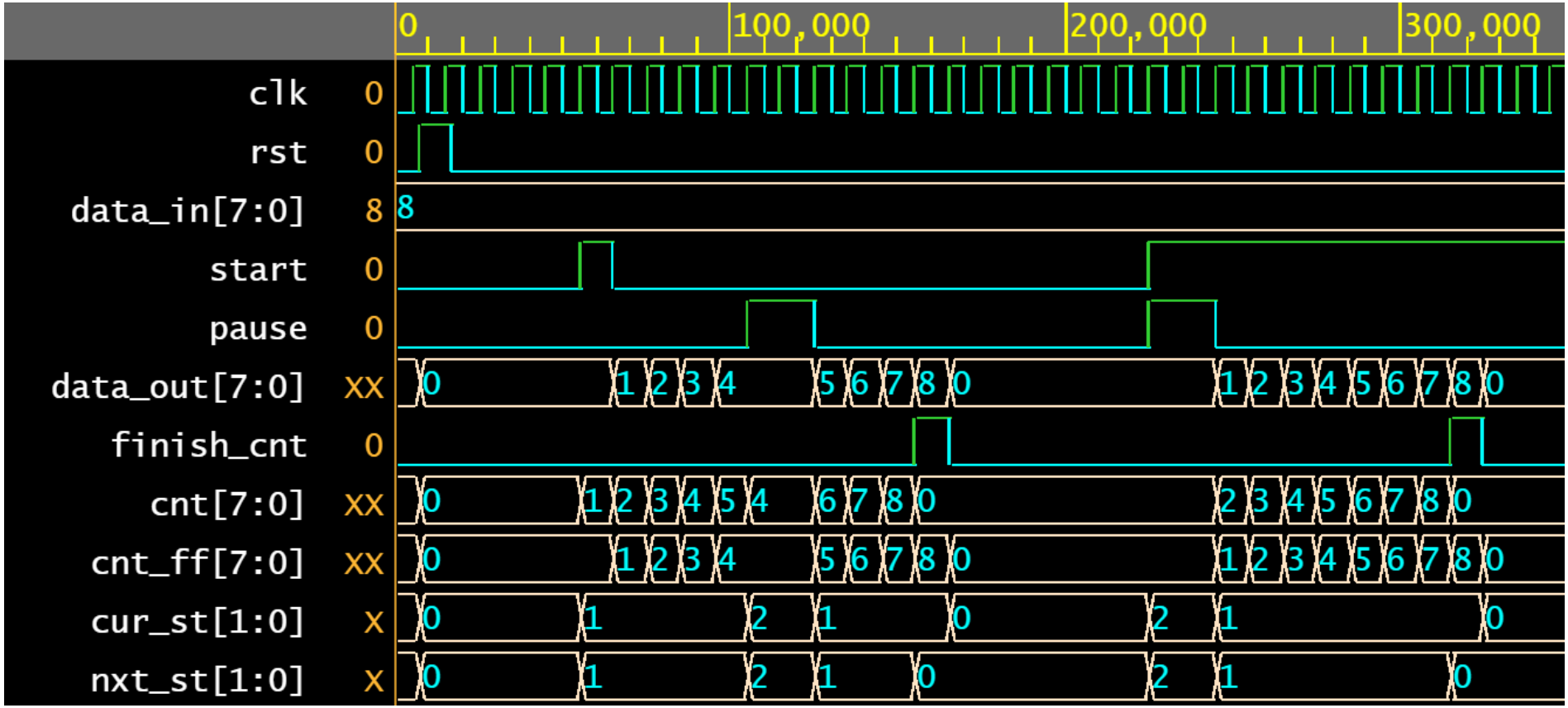
```
7   `timescale 1ns/100ps
8
9   module smart_cnt_tb();
10
11  //-----------------------------------------------------------------------------
12  // Signal Declerations
13  //-----------------------------------------------------------------------------
14      reg                                                     CLK;
15      reg                                                     RST;
16      reg                                                     START;
17      reg                                                     PAUSE;
18
19  //-----------------------------------------------------------------------------
20  // DUT Instantiation
21  //-----------------------------------------------------------------------------
22
23      smart_cnt #(.G_DATA_WIDTH(8))   DUT(.clk(CLK),
24                                          .rst(RST),
25                                          .start(START),
26                                          .pause(PAUSE),
27                                          .data_in(8'b0001000),
28                                          .data_out(),
29                                          .finish_cnt());
30
31
```

FSM – Test Bench - continued

```
32  //----------------------------
33  // Stimuli Generation
34  //----------------------------
35
36  // Clock Generation
37      initial begin
38          CLK = 0;
39      end
40
41      always begin
42          #5 CLK = ~CLK;
43      end
44
```

```
45  // Input Generation
46      initial begin
47          RST = 1'b0;
48          PAUSE = 1'b0;
49          START = 1'b0;
50          #7
51          RST = 1'b1;
52          #10
53          RST = 1'b0;
54          #30
55          @(posedge CLK);
56          START = 1'b1;
57          @(posedge CLK);
58          START = 1'b0;
59          repeat(4)
60              @(posedge CLK);
61          PAUSE = 1'b1;
62          @(posedge CLK);
63          @(posedge CLK);
64          PAUSE = 1'b0;
65          repeat(10)
66              @(posedge CLK);
67          START = 1'b1;
68          PAUSE = 1'b1;
69          repeat(2)
70              @(posedge CLK);
71          PAUSE = 1'b0;
72          #100
73          $finish
74      end
75
76  endmodule
```

FSM- in action



FSM - summary

- Before writing – design
- Divide into two processes
- Define states with meaningful names
- Every signal (wire) must have:
  - An assignment in all possible branches, or
  - Default assignment

More Information

- Online reference - http://www.asic-world.com/verilog/
- Cheat sheet - https://web.stanford.edu/class/ee183/handouts_win2003/VerilogQuickRef.pdf
- Online simulator and examples - https://www.edaplayground.com/
- GIYF