LAB 3: Simple Finite State Machine

Nate Lannan
Oklahoma State University
Electrical and Computer Engineering Department
Stillwater, OK 74078 USA
nate.lannan@okstate.edu

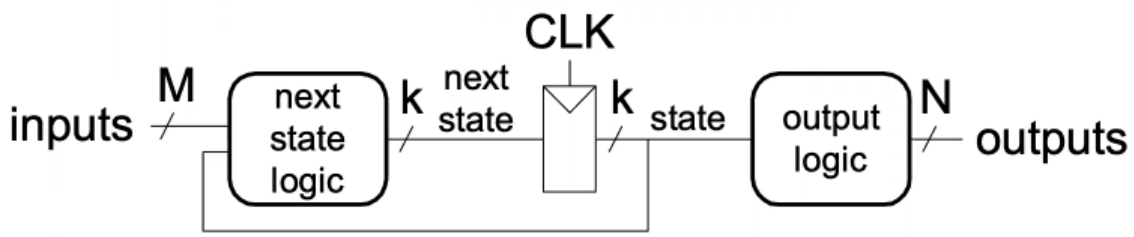
Objective

- This lab has several key objectives:
  - See what sequential logic and how to implement it in HDL.
  - Learn how HDL implementations are different than their design on paper.
  - Understand the difference between clock within digital logic and how it differs from real time.
  - Implement a simple Finite State Machine (FSM)
    - Thunderbird FSM.
    - Add hazards
    - Understand debugging of FSMs

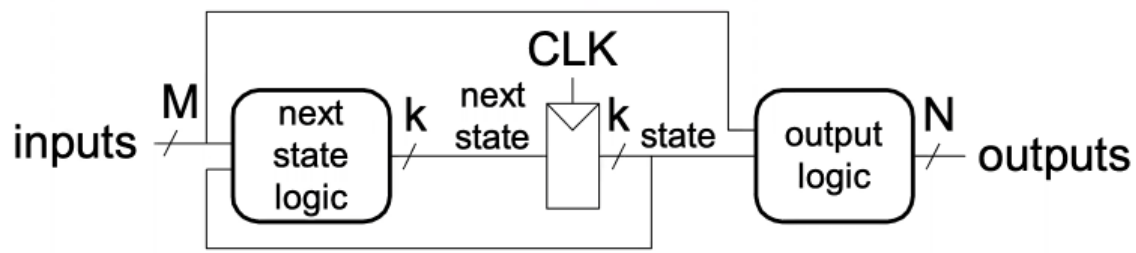
FSMs

- **Next state** determined by current state and inputs
- Two types of finite state machines differ in **output logic**:
  - **Moore FSM:** outputs depend only on current state
  - **Mealy FSM:** outputs depend on current state *and* inputs

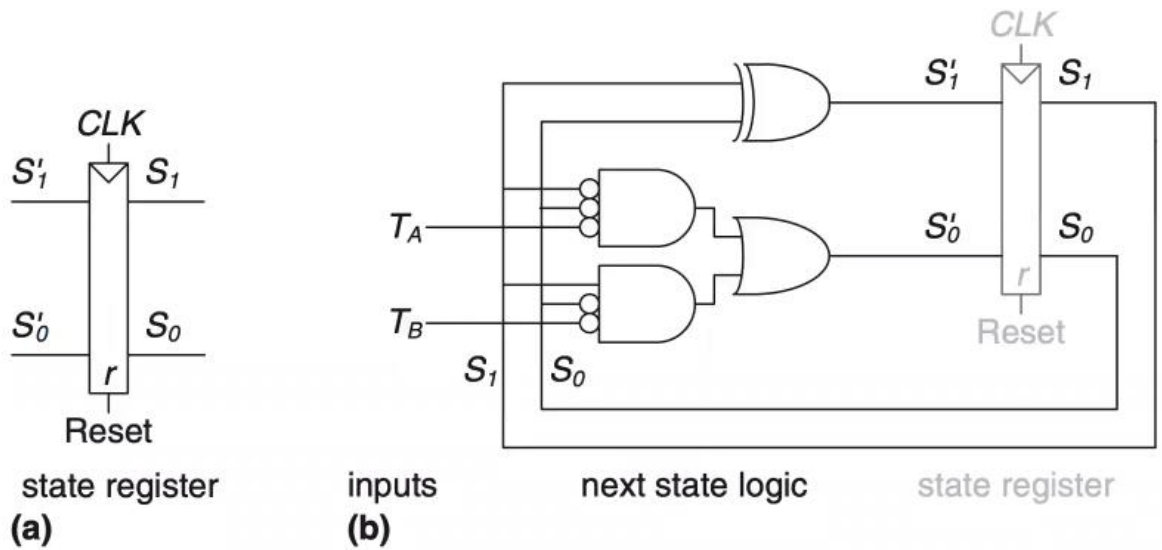
Moore FSM

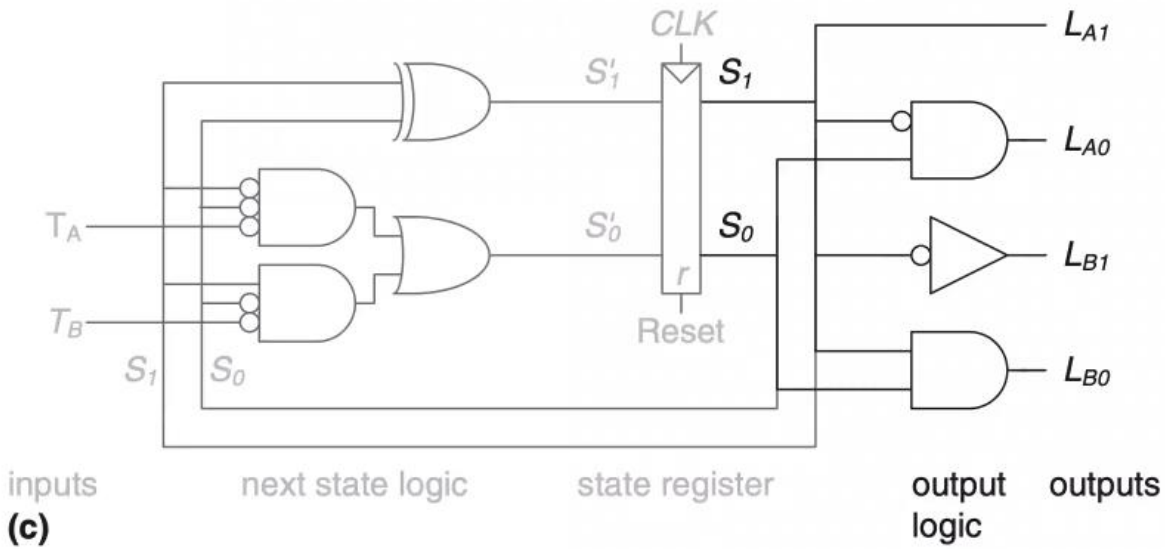


Mealy FSM



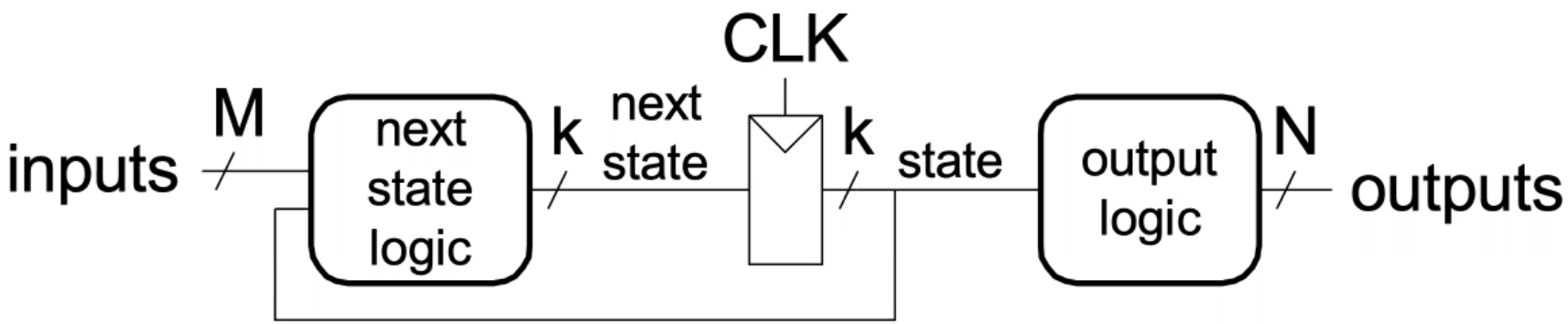
Implementation





How do we implement in HDL

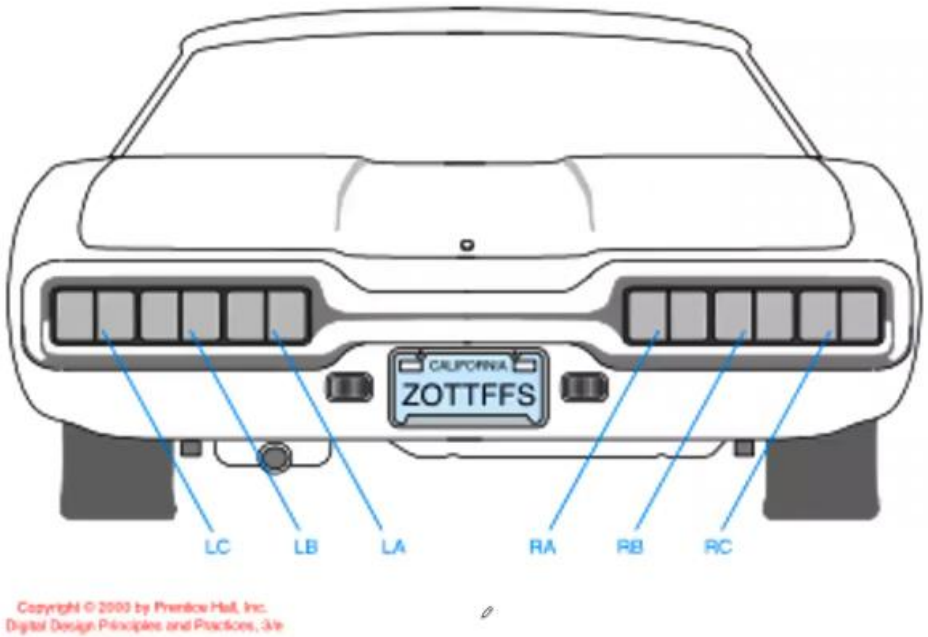
- **Three blocks:**
  - next state logic
  - state register
  - output logic
- FSMs in HDL should always have a next-state and state-register section.
- The output logic can integrated into the state next-state section, if needed.
  - That is, either is acceptable but having three sections saves work as we will see!

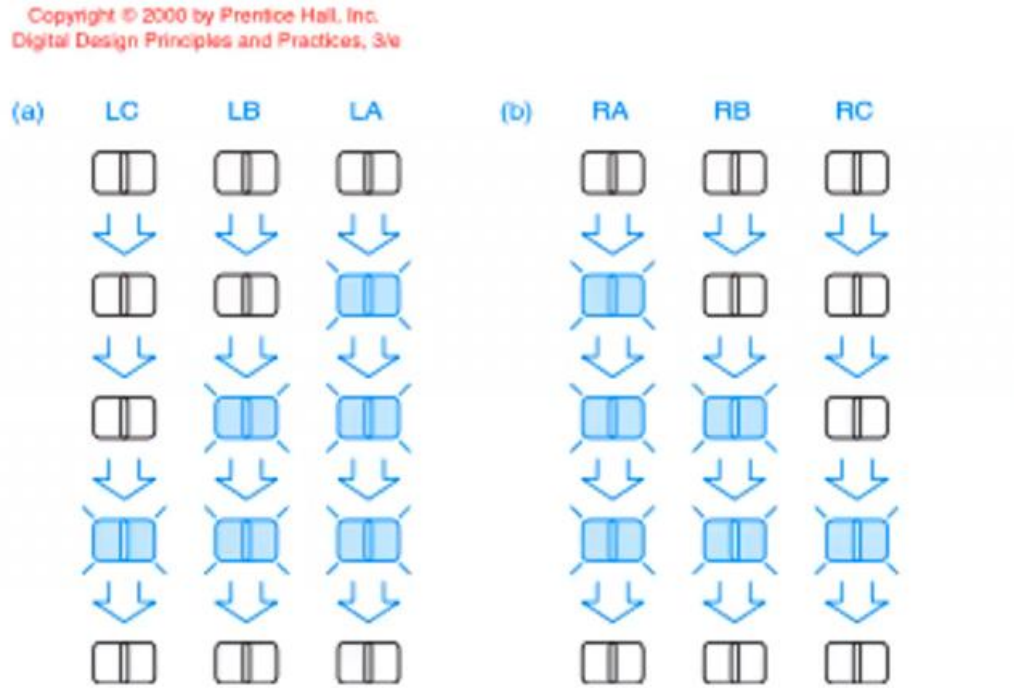


FSM Implementation in SV

```
module divideby3FSM (input  logic clk,
                     input  logic reset,
                     output logic q);

   typedef enum logic [1:0] {S0, S1, S2} statetype;
   statetype [1:0] state, nextstate;

   // state register
   always_ff @ (posedge clk, posedge reset)
      if (reset) state <= S0;
      else       state <= nextstate;

   // next state logic
   always_comb
      case (state)
         S0:
           begin
                nextstate = S1;
                q = 1'b1;
           end
         S1:
           begin
                nextstate = S2;
                q = 1'b0;
           end
         S2:
           begin
                nextstate = S0;
                q = 1'b0;
         default:
           begin
                nextstate = S0;
                q = 1'b0;
           end
      endcase
endmodule
```

What are we implementing?





[Wakerly]

https://youtu.be/5azgaPDvPDk

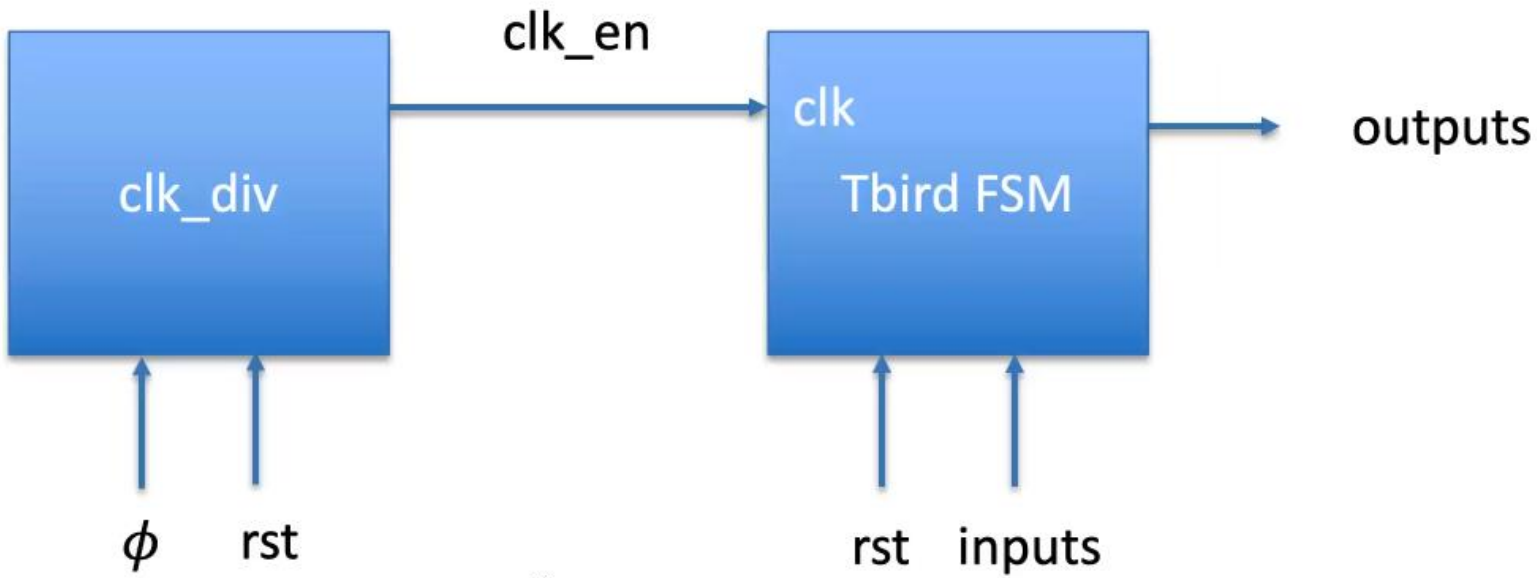
Seems rather straightforward?

- Yes, it is very simple if you follow the SV HDL coding methodology.
- What's the challenge?
  - The challenge is that the clock on the DSDB board operates so much faster than what our eyes see!
  - We need to slow the clock down in order to even see things!
- We use a clock to synchronize, so you can always slow things down in two ways:
  - Use a clock enabler that activates only once per so many iterations.
  - The other option, which we are using here, is a clock divider like the design we created in class but with a counter.

Clock Divider

```
module clk_div (input logic clk, input logic rst, output logic clk_en);

 logic [23:0] clk_count;

 always_ff @(posedge clk)
 //posedge defines a rising edge (transition from 0 to 1)
  begin
   if (rst)
    clk_count <= 24'h0;
   else
    clk_count <= clk_count + 1;
  end
 assign clk_en = clk_count[23];
endmodule
```

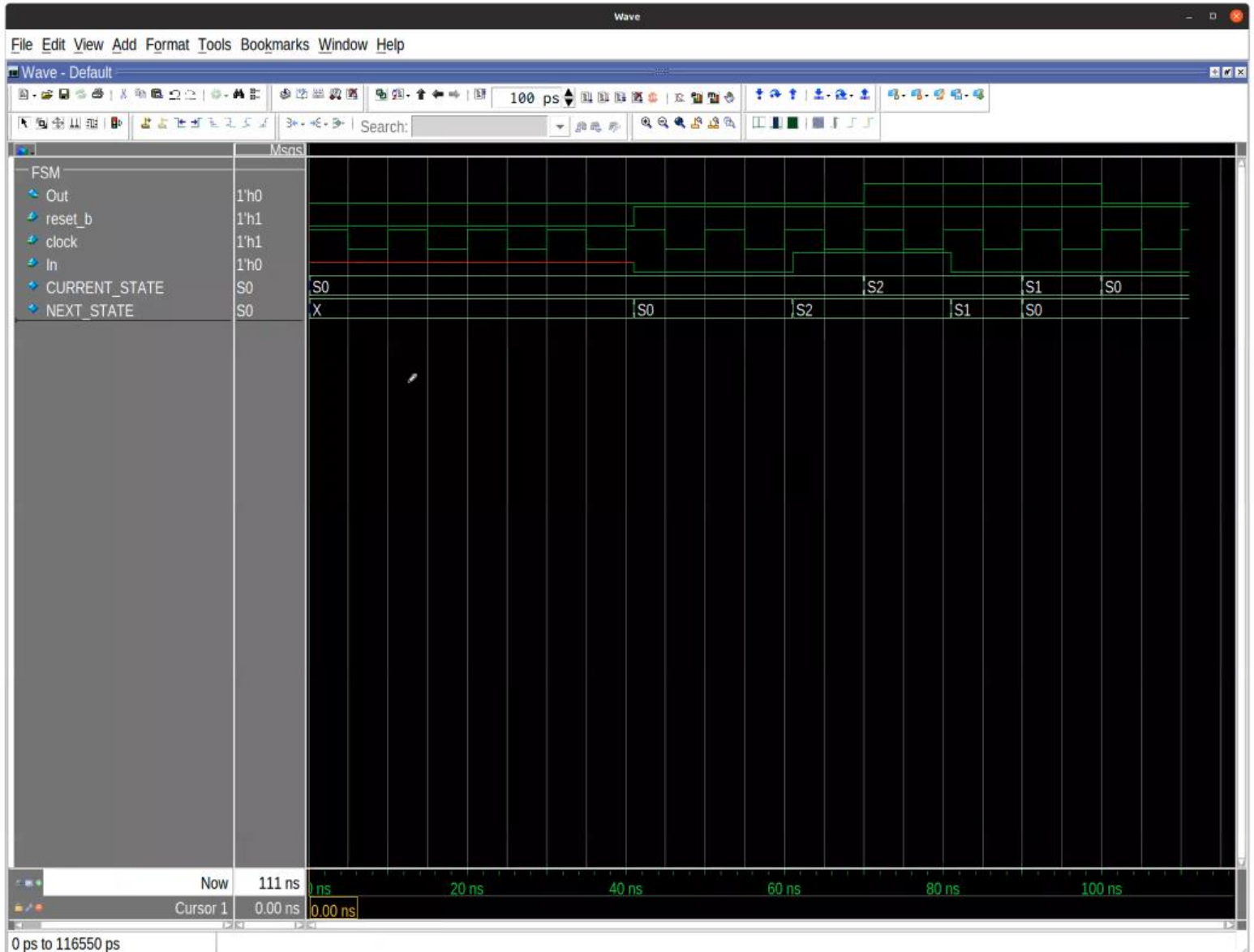
Basic Clock Divider Architecture



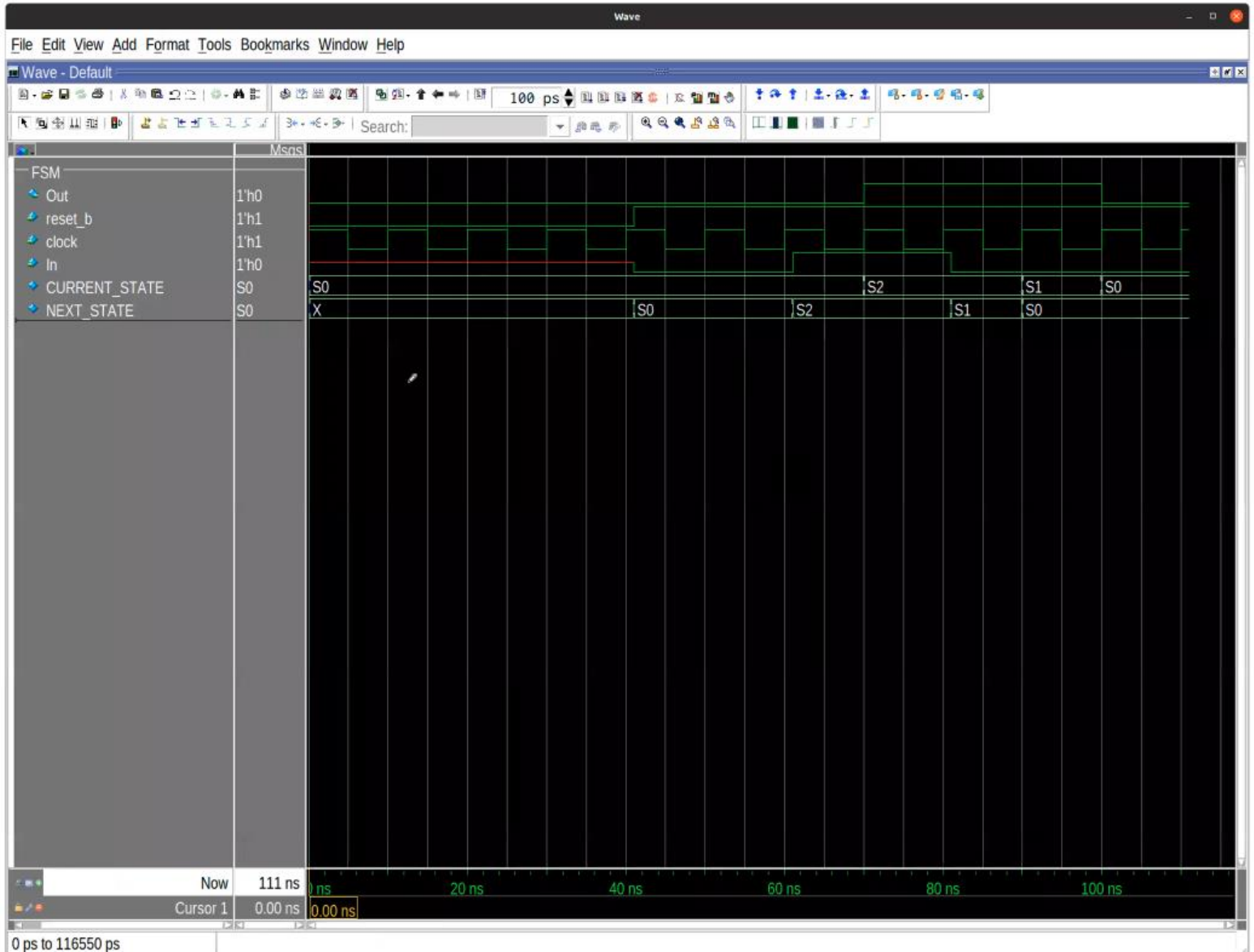
Hints

- Use the LEDs, 7-segment display, and other items on your DSDB to help you.
- Always simulate with the testbench before you begin implementing!!!
  - Do not even attempt the implementation without trying to completely verify your state machine in a testbench.
  - FSMs are notoriously more difficult than combinational logic in that the output is now comprised of inputs + current state which leaves so many more possible combinations that can lead to failure.
- Use the `clk_en` output from the clock divider to clock your FSM.
  - Play around with the counter to change the time the lights blink.
- Use your GUI to help you debug things
  - Always output CURRENT_STATE and NEXT_STATE to your GUI to see what state you are in to see what is happening.

MGC ModelSim Wave Window



MGC ModelSim Wave Window



DOT files

- Drawing your FSM might be difficult.
- I have provided a DOT file in the repository for graphviz as its easy to use.
- Search “graphviz online”
  - Let’s try…

Conclusion

- FSMs are fun but good methodologies for implementing and debugging HDL are essential to making these designs work.
- Almost every digital design uses a FSM to control its behavior
  - It is important you understand how to design FSMs and implement them.
  - Good combinational logic understanding is essential to making FSMs work properly.
  - Design of FSMs on paper differs from what you implement in a HDL but everything always starts with the state-transition table (do not forget to make one for this laboratory).