SV Notes #6: FSM Optimization

New Keywords and Operators

1. **`enum`**

   Specifies an **enumerated** data type that consists of a list of values. We use **`enum`** to declare variables with constant values, representing states with symbolic names:

   ```
   enum logic [1:0] {RED = 2'b00, YEL = 2'b01, GRN = 2'b11} state, nextState;
   ```

   This line defines the 2-bit signals `state` and `nextState`, which will have one of three names `RED`, `YEL`, or `GRN`. This definition assigns the binary constants **`2'b00`**, **`2'b01`**, and **`2'b11`** to these names.

   It's possible to specify an alternate **state assignment**, like the following one-hot encoding:

   ```
   enum logic [2:0] {RED = 3'b001, YEL = 3'b010, GRN = 3'b100} state, nextState;
   ```

2. **`.name`** - Example: `dut.state.name`

   When appended after an enumerated signal, returns a string presenting the the 'name' of the signal. For this example, assume there is a finite state machine called `dut` that has been instantiated. Inside the module `dut` is a signal called `state` of an enumerated type (like the examples shown for **`enum`** above). We could view the present state with the following statement in a testbench:

   ```
   $display("State: %s", dut.state.name);
   // console might display 'State: RED', 'State: YEL', or 'State: GRN'
   ```

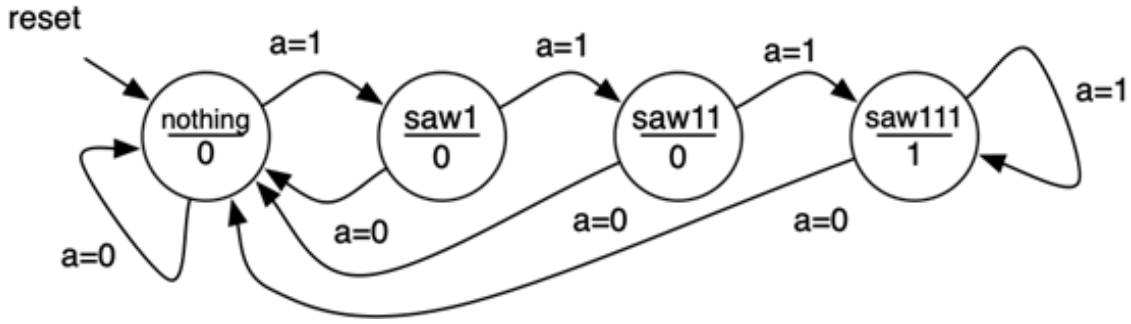
Code Listings

Finite State Machine

This listing defines an FSM that detects overlapping sequences of three ones at the input *a*, like the example discussed in Lecture 8. The code uses **`enum`** for the state assignment. This makes the specification of the finite state machine easier to read and understand. While this allows us to use descriptive names to refer to states, there is still a unique binary encoding mapped to each state.

This listing makes use of **`unique case`**`(state)` which asserts that one and only one `state` will be selected. If *two* of our states are valid at the same time, we would like the synthesis tool to throw an error! Any input combinations of `state` that do not match will be treated as don't cares by the synthesis tool.

Here is the state transition diagram for reference:




```
module sequenceDetector
  (input  logic a, clock, reset_n,
   output logic f);

  // specify the state assignment with an enum statement
  // 18240 style is for all enumerated labels are written in uppercase text
  enum logic [1:0] {NOTHING = 2'b00, SAW1 = 2'b01,
                    SAW11 = 2'b10, SAW111 = 2'b11} state, nextState;

  // state register updates on positive clock edges
  always_ff @(posedge clock, negedge reset_n)
    if (~reset_n)
      state <= NOTHING;   // reset state is NOTHING
    else
      state <= nextState;

  always_comb begin      // next state generator
    unique case (state)
      NOTHING: nextState = (a) ? SAW1   : NOTHING;
         SAW1: nextState = (a) ? SAW11  : NOTHING;
        SAW11: nextState = (a) ? SAW111 : NOTHING;
```

```
      SAW111: nextState = (a) ? SAW111 : NOTHING;
    endcase
  end

  always_comb begin      // output logic:
    unique case (state)  //      f = 1 only when FSM is in state 'SAW111'
      NOTHING: f = 0;    // can you think of an alternate way to specify this?
         SAW1: f = 0;
        SAW11: f = 0;
       SAW111: f = 1;
    endcase
  end
endmodule : sequenceDetector
```

FSM testbench

This is a testbench that generates a clock signal and tests the `sequenceDetector` from the previous listing. It's important to realize that the clock is generated in a different **`initial`** block than the **`posedge`**

Observe how **`@ (posedge clock);`** is used as an event in the initial block, a bit like an explicit delay (**`#5;`** for example). In this case, the initial block will wait for a positive clock edge and then move on to the next instruction.

The **concurrent assignment operator `<=`** is used to *schedule* signal transitions to occur simultaneously at the end of the present time step.

```
module fsm_testbench;
  logic clock, reset_n, a, f;                     // define signals

  sequenceDetector dut(.clock, .reset_n, .a, .f); // instantiate FSM as dut

  // clock/reset block
  initial begin
    clock  = 0;
    reset_n = 0;
    reset_n <= 1;        // schedules de-assertion of reset at end of startup
    forever #10 clock = ~clock;  // clock inverts every 10 time units
    // this will run forever unless a $finish is in another initial block
  end

  // monitor
  initial begin
    $monitor($time,,"State=%s, a=%b, f=%b", dut.state.name, a, f);
  end

  // generate transitions for testing
  initial begin
    a <= 0;             // schedules startup value of input
    @ (posedge clock);  // time = 10, first posedge since clock period is 20
    a <= 1;             // schedules a <= 1 at current step (time = 10)
    @ (posedge clock);  // time = 30
    @ (posedge clock);  // time = 50
    @ (posedge clock);  // time = 70
    @ (posedge clock);  // time = 90
    a <= 0;             // schedules a <= 1 at current step (time = 90)
    @ (posedge clock);  // time = 110
    #5 finish;          // time = 115, simulation ends
  end
endmodule : fsm_testbench
```