Finite State Machines

CS141 Spring 2019

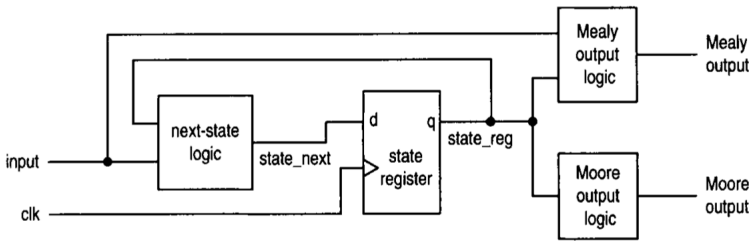
Introduction

A finite state machine is a sequential circuit that moves between a finite number of internal states. The transition to the next state depends on the current state and external input. In practice the main purpose of an FSM is to act as the controller of a digital system. As we will see later in the course, an FSM is used to control the operation of a CPU datapath, which is composed of combinational logic and registers to store intermediate computations.

The figures and examples are adapted from *FPGA Prototyping by SystemVerilog Examples* By Pong P. Chu.

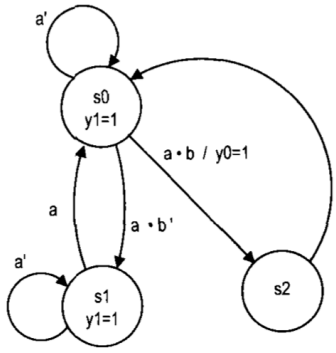
Basic circuit diagram

The general block diagram (shown below) of an FSM is the same as the block diagram we saw last time for the sequential circuit, except now we differentiate between "Mealy" and "Moore" FSMs. The FSM is a *Mealy Machine* if the output is a function of the current state and the input. The FSM is a *Moore Machine* if the output is just a function of the current state. The outputs of Moore and Mealy machines are similar and both may be used to construct FSMs.



State diagram

A state diagram is composed of *nodes* (states) and *transitional arcs* (state transitions). The Moore output values are placed inside the node since they depend only on the current state. Mealy output values are associated with the transitional arcs since they depend on current state and current input. A simple state diagram is shown below, with two inputs `a` and `b`, one Moore output signal `y1` and one Mealy output signal `y0`. The `y1` (Moore) signal is asserted when the FSM is in state 0 or 1, and the `y0` (Mealy) signal is asserted when the FSM is in state 0 and `a` and `b` are both 1.



FSM code development skeleton

The procedure for developing Verilog code for an FSM is similar to building a sequential circuit: we have a register that stores the current state and a combinational always block to determine the next state.

To facilitate symbolic state names, we use the `localparam` Verilog keyword, which defines a parameter that is local to the current module:

```
localparam [1:0] s0 = 2'b00,
                 s1 = 2'b01,
                 s2 = 2'b10;
```

During synthesis, software sometimes can recognize the FSM structure and may map these symbolic constants to different binary representations.

The complete code for an FSM follows the following structure:

```
module fsm
    (
        input wire clk, reset,
        input wire a, b,
        output wire y0, y1
    );

    // fsm state declaration
    localparam [1:0] s0 = 2'b00,
                     s1 = 2'b01,
                     s2 = 2'b10;

    // signal declaration
    reg [1:0] state_reg, state_next;

    always @(posedge clk, posedge reset)
        if (reset)
            state_reg <= s0;
        else
            state_reg <= state_next;
```

```
    // next-state logic
    always @*
        case (state_reg)
            s0:
                if (a)
                    if (b)
                        state_next = s2;
                    else
                        state_next = s1;
                else
                    state_next = s0;
            s1:
                if (a)
                    state_next = s0;
                else
                    state_next = s1;
            s2: state_next = s0;
            default: state_next = s0;
        endcase

    // Moore output logic
    assign y1 = (state_reg==s0) || (state_reg==s1);

    // Mealy output logic
    assign y0 = (state_reg==s0) & a & b;
endmodule
```

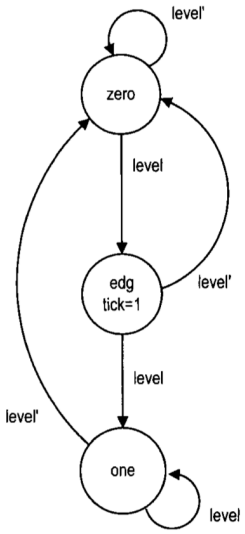
It is also possible to merge the combinational logic blocks into one as shown below, just make sure that outputs are always assigned to for every path!

```
always @*
begin
    state_next = state_reg; // default next state: the same
    y1 = 1'b0;              // default output: 0
    y0 = 1'b0;              // default output: 0
    case (state_reg)
        s0: begin
            y1 = 1'b1;
            if (a)
                if (b)
                begin
                    state_next = s2;
                    y0 = 1'b1;
                end
                else
                    state_next = s1;
        end
        s1: begin
            y1 = 1'b1;
            if (a)
                state_next = s0;
        end
        s2: state_next = s0;
```

```
        default: state_next = s0;
    endcase
end
```

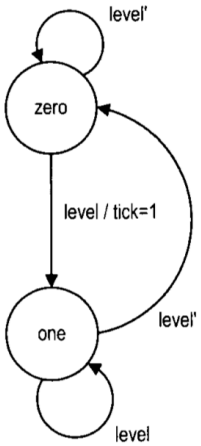
Example: Rising Edge Detector

The rising edge detector is a circuit that generates a short one-cycle pulse every time the input signal changes from a 0 to a 1. The state diagram for a Moore machine is shown below.



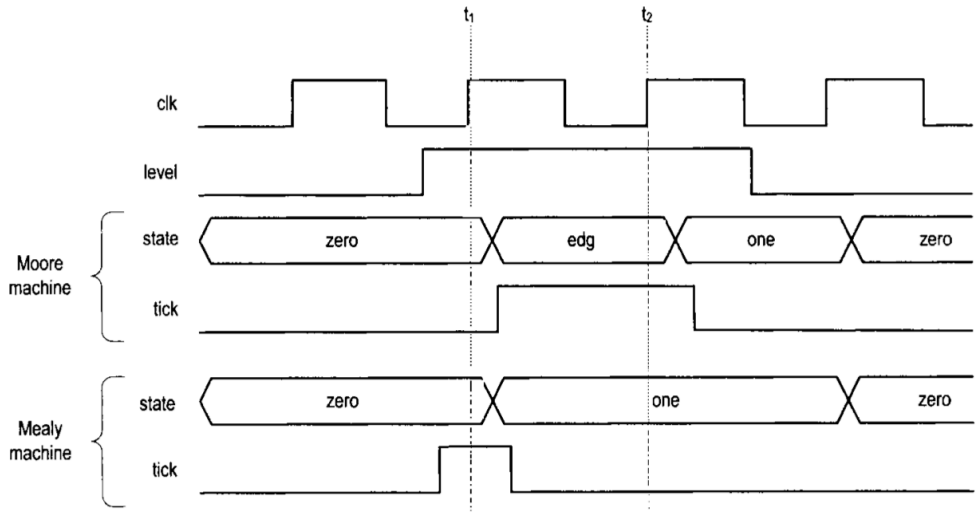
The `zero` and `one` states indicate that the input signal has been constant for a while. When the input (`level`) has a rising edge, the state will be `zero` and the input will be a `1`. The FSM will then move to the `edg` state and output `tick=1`. On the next clock edge, the FSM will move to `zero` or `one` depending on the input.

We can also implement this FSM as a Mealy machine, as shown below.



In this case, the `zero` and `one` states have similar meanings, but now on a rising edge we can assert 1 as part of the transitional arc. On the next clock the output will be deasserted.

The timing diagram of both implementations is shown below. Note that the propagation delay of gates is taken into account in this diagram.



We can implement this in Verilog using the techniques from the example FSM above.

See the "verilog examples" folder on the course website for both Moore and Mealy implementations of the rising edge detector (named `edge_detect_mealy.v` and `edge_detect_moore.v`).