Basic State Machines

Mealy Machine

Moore Machine
Implementation of Mealy and Moore Machines

Mealy Machine

Moore Machine

Current State

Next State Logic

State Reg

Output Logic

input → Output Logic → output

input → Next State Logic → State Reg → Output Logic → output

input → Next State Logic → State Reg → Output Logic → output
Every State Machine can be segmented into a combinatorial and sequential parts. We will often code these two pieces in separate code blocks.

(Note: simple register operations such as a synchronous reset can be coded in either the comb. or the seq. block, but such simple things make sense to be coded in the sequential piece. An asynchronous reset should be part of the seq. block.)
State-Machine Implementation

Code header

```verilog
module one_hot (clk, rst, x, y);

input  clk, rst, x;
output  y;

reg [1:0] y;

// Declare the symbolic names for states using parameter
parameter [6:0] S1 = 7'b0000001, S2 = 7'b0000010, ...
    S7 = 7'b1000000;

// Declare current state and next state variables
reg [2:0] CS, NS; // reg != register
```

Consider more meaningful names, e.g. S_WAIT_FOR_READY, S_INIT
always @ (posedge CLOCK or posedge RESET) begin
    if (RESET == 1'b1) CS <= S1;
    else CS <= NS;
end

always @ (CS or x) begin
    case (CS)
        S1 : begin
            y  = 2'b00;
            if (x[2] && ~x[1] && x[0])
                NS = S2;
            else if (x[2] && x[1] && ~x[0])
                NS = S4;
            else
                NS = S1;
        end
        S2 : begin
            y  = 2'b10;
        end
    endcase
end

Note: Having RESET in sensitivity list implements Async. reset

Mealy or Moore?

output

nextstate
Code header

```VERILOG
module one_hot (clk, rst, x, y);

input  clk, rst, x;
output  y;

reg [1:0] y;

// Declare the symbolic names for states using parameter
parameter [6:0] S1 = 7'b0000001, S2 = 7'b0000010, ...  
  S7 = 7'b1000000;

// Declare current state and next state variables
reg [2:0] CS, NS; // reg!= register
```

Consider more meaningful names, e.g. S_WAIT_FOR_READY, S_INIT
always @ (posedge CLOCK or posedge RESET) begin
    if (RESET == 1'b1) CS <= S1;
    else
        CS <= NS;
end

always @ (CS or x) begin
    case (CS)
        S1 : begin
            y = {x[2] , ~x[1] && x[0]};
            if (x[2] && ~x[1] && x[0])
                NS = S2;
            else if (x[2] && x[1] && ~x[0])
                NS = S4;
            else
                NS = S1;
        end
        S2 : begin
            y = {x[1] && ~x[1] , x[0]};
    end
end
always @ (posedge CLOCK or posedge RESET) begin
    if (RESET == 1'b1) CS <= S1;
    else
        CS <= NS;
end

always @ (CS or x) begin
    case (CS)
        S1 : begin
            y = {x[2], ~x[1] && x[0]};
            if (x[2] && ~x[1] && x[0])
                NS = S2;
            else if (x[2] && x[1] && ~x[0])
                NS = S4;
            else
                NS = S1;
        end
        S2 : begin
            y = {x[1] && ~x[1] , x[0]};
             .
             .
             .
    end
end

Output logic

- Combinatorial Logic uses blocking statements
- Every output is set in every case to avoid latches

Registers use non-blocking assignments. ...will play better with other code in simulations.
Registered Outputs: 
- Avoid Glitches
- Mitigate Metastability
- Add clk cycle delay to outputs

“Registered Output Moore” and “Registered Output Mealy” Machines
If you consider the output logic register to be part of the state (serving as extended state register variable), a “Registered Output Mealy” machine is technically a Moore Machine.
“OK” Coding Mixed Style for Registered Output Logic (Mealy)...
if you are careful

always @ (posedge CLOCK or posedge RESET)
begin
  if (RESET == 1'b1) CS <= S1;
  else case (CS)
    S1 : begin
      temp = ~x[1];
      y  <= {x[2] , temp & x[0]};
      if (x[2] && temp && x[0])
        CS <= S2;
      else if (x[2] && x[1] && ~x[0])
        CS <= S4;
      else
        CS <= S1;
    end
    S2 : begin
      y  <= {x[1] && ~x[1] , x[0]};
      .
  end
end

Embedded comb. circuit signal assignments should be blocking and thus encode “immediate” effect. All consumers of blocking assignment results should be within this code block. Even within the block, no consumer may rely on a value assigned from a blocking statement in a previous trigger event.

Remember, any variable written to inside an edge-triggered block can become a register regardless of the use of blocking or non-blocking...consider every output variable, comb. and seq., in every case and branch of decision tree and make sure assignments are always made to avoid latches

Intended register outputs should use non-blocking
• Registered output cause “delays” so some output transitions need to be coded along with the state transition to a state rather than with the state they are supposed to coincide with.
• Sometimes this feels like your coding outputs in the “previous state” or coding output ahead of time to account for register delay. I refer to it as coding the output along with the state transition. This leads to additional lines of code as you need to code each output logic possibly for every transition to a state rather than once per state.
• To avoid this “code bloat”, yet another approach is to code the registered outputs in a separate block. This leads to three blocks:
  • Combinatorial Next State Logic along with any combinatorial outputs
  • Sequential State Register
  • Sequential Registered Outputs according to the destination (next) state from the combinatorial Block.
• Good contrasting examples can be found here:
“Registered Output Moore” and “Registered Output Mealy” Machines

Avoid Delays while allowing registered outputs to be coded with the state they are intended to coincide with, by coding Next State and Output for Next state together.
registered outputs using three always blocks

always (posedge clk) begin
    if reset ... 
    else CS<=NS;
end

always(*) begin
    NS=CS;
    case(CS)
    S_init: begin
        if (go==1 && selAB==0) NS=S_startA;
        ...
    end;
    S_startA: begin
        NS=S_init;
    end;
    ...
end

Next State Logic and Any combinatorial outputs

Next State Registered Output Logic
Once Per State Rather than once per transition
-May also include Transition dependent rules based on Current State
Example State-Machine: An Arbitrator

Master

uP

selMD

start

do

di

busy

startA

busA

doA

diA

busyB

doB

diB

Multiplies

Divider

Slaves

startM

busM

doM

diM

startD

busD

doD

diD

Multiplier

Divider
Example State-Machine: An Arbitrator

Master

uP

selMD

start

do

di

busy

start

di

busy

Multiplier

Slaves

startM

busyM

doM

diM

startA

busyA

doA

diA

selAB
module my_state_machine2(
  input clk,
  input rst,
  output reg startA, // start signal to slave A
  input busyA_n, // busy signal from slave A
  output ready, // ready signal to master
  input start // go signal from master
);

reg [7:0] CS;

parameter S_init = 8'b00000000;
parameter S_startA0 = 8'b00000001;
parameter S_startA1 = 8'b00000010;

assign ready = ~busyA_n;

always @ (posedge clk) begin
  if (rst == 1)
    CS<=S_init;
  else
    case (CS)
      S_init: begin
        if (start == 1) begin
          startA <= 1;
          CS<=S_startA0;
        end else begin
          startA <= 0;
          CS<=S_init;
        end
      end
      S_startA0: begin
        startA <= 1; //***
        CS<=S_startA1;
      end
      S_startA1: begin
        startA <= 0;
        CS<=S_init;
      end
    endcase
end
endmodule
module my_state_machine2(
    input clk,
    input rst,
    output reg startA, // start signal to slave A
    input busyA_n, // busy signal from slave A
    output reg ready, // ready signal to master
    input start // go signal from master
); 

reg [7:0] CS;
reg [7:0] NS;

parameter S_init = 8'b00000000;
parameter S_startA0 = 8'b00000001;
parameter S_startA1 = 8'b00000010;

assign ready = ~busyA_n;

reg goA;
reg goA__;

always @ (posedge clk) begin
    if (rst == 1)
        CS <= S_init;
    else
        CS <= NS;
end

always @ (posedge clk) begin
    if (go) begin
        NS = S_startA0;
    end
    S_startA0: begin
        NS = S_startA1;
    end
    S_startA1: begin
        NS = S_init;
    endcase // case (CS)
end // always @ (*)

//RESET? DEFAULT?
always @ (posedge clk) begin
    case (NS)
        S_init: begin
            startA <= 0;
        end
        S_startA0: begin
            NS = S_startA1;
            startA <= 0;
        end
        S_startA1: begin
            NS = S_init;
        endcase // case (CS)
    end // always @ (*)
end // RESET? DEFAULT?

//RESET? DEFAULT?
always @ (posedge clk) begin
    case (CS)
        S_init: begin
            if (go) begin
                NS = S_startA0;
            end
        end
        S_startA0: begin
            NS = S_startA1;
        end
        S_startA1: begin
            NS = S_init;
        endcase // case (CS)
    end // always @ (*)
end // RESET? DEFAULT?