Verilog Case-Statement-Based State Machines

Prof. Ryan Robucci
Basic State Machines

Mealy Machine

Moore Machine
Implementation of Mealy and Moore Machines

Mealy Machine

Moore Machine

Combinatorial Logic

Current State

Next State Logic

State Reg

Output Logic

Input

Output

Next State Logic

State Reg

Output Logic

Input

Output
All State Machines can be partitioned into a combinatorial and sequential parts. We will often code these two pieces in separate code blocks, but they may also be combined. (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

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
localparam [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 CLEAR) begin
    if (CLEAR == 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 CLEAR in sensitivity list implements async. CLEAR/RESET

Mealy or Moore?

output

nextstate
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
localparam [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 CLEAR) begin
  if (CLEAR == 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
  output
  nextstate
end
always @ (posedge CLOCK or posedge CLEAR) begin
  if (CLEAR == 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
“Registered Output Moore” and “Registered Output Mealy” Machines

Registered Outputs:
• Avoid Glitches
  - critical for generating edge-sensitive control signals
  - May achieve lower-power operation by avoiding charge/discharge cycles
• Mitigate Metastability
  (invalid voltage levels or late transitions)
  - Adds clk cycle delay to outputs
  - Standardizes delay constraints for blocks
    (avoid creating long combinatorial paths through multiple blocks)
Registered Outputs (continued):
- Additionally, code can reuse previous outputs for future output and state updates, meaning the new registers are part of the state of the system.
- In code, you can update outputs as needed in a given state and retain values by default if not assigned.

![Diagram of 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”), a “Registered Output Mealy” machine is technically a Moore Machine.
always @ (posedge CLOCK or posedge CLEAR) begin
  if (CLEAR == 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
  endcase
end

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.
Coding Styles for Statemachines

- Registered output causes “cycle 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 (discuss output on prev. slide)

- Sometimes this feels like you’re 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 alongside 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:
  1) Combinatorial Next State Logic along with any combinatorial outputs
  2) Sequential State Register
  3) Sequential Registered Outputs according to the destination (next) state from the combinatorial Block.

- Good contrasting examples can be found here:
Avoids 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; /*NS is result of combinatorial logic*/
  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 combinational outputs

Next State Registered Output Logic
Once Per State Rather than once per transition
-May also include Transition-specific output rules based on both CS and NS
Example State-Machine: An Arbitrator

For simplicity, in the following code this block is omitted.
• Assume slave need a start signal for two clock cycles
• Using more states simplifies output logic

• Using Less states requires more output logic and in the code shown, effectively embeds extended state information in additional registers
module control_state_machine2(
    input clk,
    input rst,
    output reg startA, //start signal to slave A
    output reg startB, //start signal to slave B
    input busyA_n, //busy signal from slave A
    input busyB_n, //busy signal from slave B
    input ready, //ready signal to master
    input selAnB //go signal from master
    input start //go signal from master
);

reg [7:0] CS;

localparam S_init = 8'b00000000;
localparam S_start0 = 8'b00000001;
localparam S_start1 = 8'b00000010;

assign ready = ~(selAnB?busyB_n:busyA_n);

always @ (posedge clk) begin
    if (rst == 1)
        CS<=S_init;
    else
        case (CS)
            S_init: begin
                if (start == 1) begin
                    startA <= ~selAnB;
                    startB <= selAnB;
                    CS<=S_start0;
                end else begin
                    startA <= 0;
                    startB <= 0;
                    CS<=S_init;
                end
            end
            S_start0: begin
                startA <= startA;
                startB <= startB;
                CS<=S_start1;
            end
            S_start1: begin
                startA <= 0;
                startB <= 0;
                CS<=S_init;
            end
        endcase
end
endmodule

Coding update alongside transition within the case code for the current state (the state held in the state register prior to the corresponding updates in the case code).
Timing Diagram

• Place holder for in-class timing diagram.
  Key concepts:
  - Identification of implied combinatorial signals including register inputs (outputs and state (NS))
  - Arrival of input changes versus response timing
  - Coincidence of Code and Output Timing
module control_state_machine2(
    input clk,
    input rst,
    output reg startA, // start signal to slave A
    output reg startB, // start signal to slave B
    input busyA_n, // busy signal from slave A
    input busyB_n, // busy signal from slave B
    output ready, // ready signal to master
    input selAnB // go signal from master
);

reg [7:0] CS,NS;

localparam S_init = 8'b00000000;
localparam S_start0 = 8'b00000001;
localparam S_start1 = 8'b00000010;

assign ready = ~(selAnB?busyB_n:busyA_n);

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

always @ (posedge clk) begin
    if (rst == 1)
        NS = CS;
    case (CS)
        S_init: begin
            if (go) begin
                NS = S_start0;
            end
        end
        S_start0: begin
            NS = S_start1;
        end
        S_start1: begin
            NS = S_init;
        end
    endcase
end // always @ (*)

endmodule