EE178 Lecture Verilog FSM Examples

Eric Crabill

SJSU / Xilinx

Fall 2007

XILINX®

Finite State Machines

- In *Real-time Object-oriented Modeling*, Bran Selic and Garth Gullekson view a state machine as:
  - A set of input events
  - A set of output events
  - A set of states
  - A function that maps states and input to output
  - A function that maps states and inputs to states
  - A description of the initial state

Finite State Machines

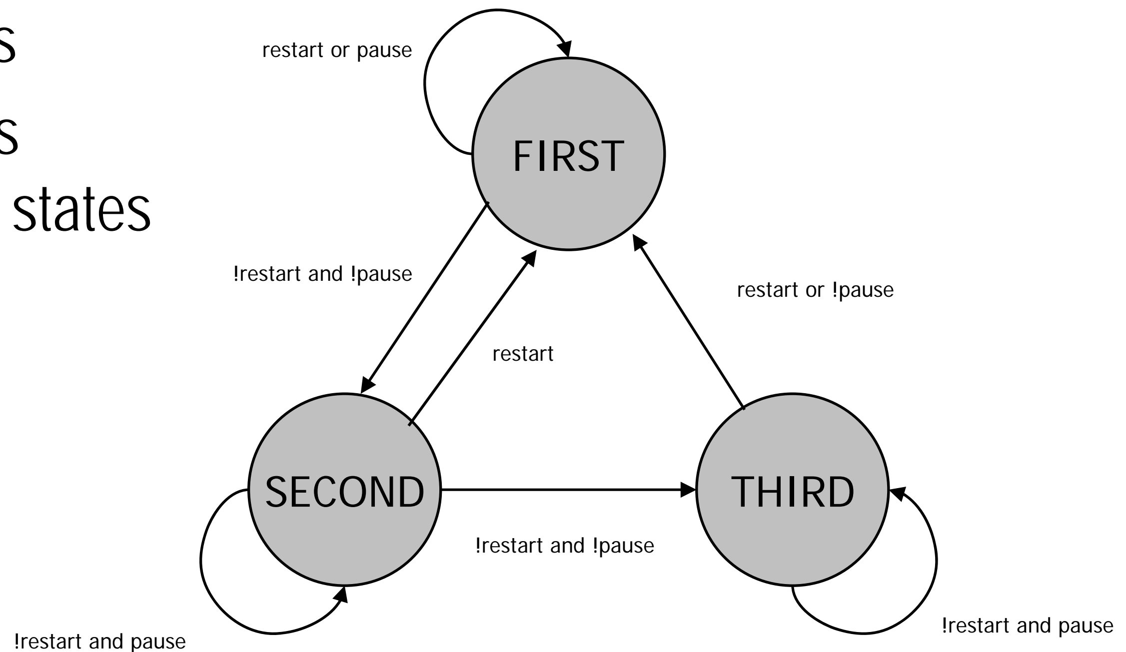
- A finite state machine is one that has a limited, or *finite*, number of states.
- The machine state is described by a collection of state variables.
- A finite state machine is an abstract concept, and may be implemented using a variety of techniques, including digital logic.

Finite State Machines

- For an edge-triggered, synchronous FSM implemented in digital logic, consider:
  - A set of input events (input signals, including clock)
  - A set of output events (output signals)
  - A set of states (state variables are flip flops)
  - A function that maps states and input to output (this is the output logic)
  - A function that maps states and inputs to states (this is the next-state logic)
  - A description of the initial state (initial flip flop value)

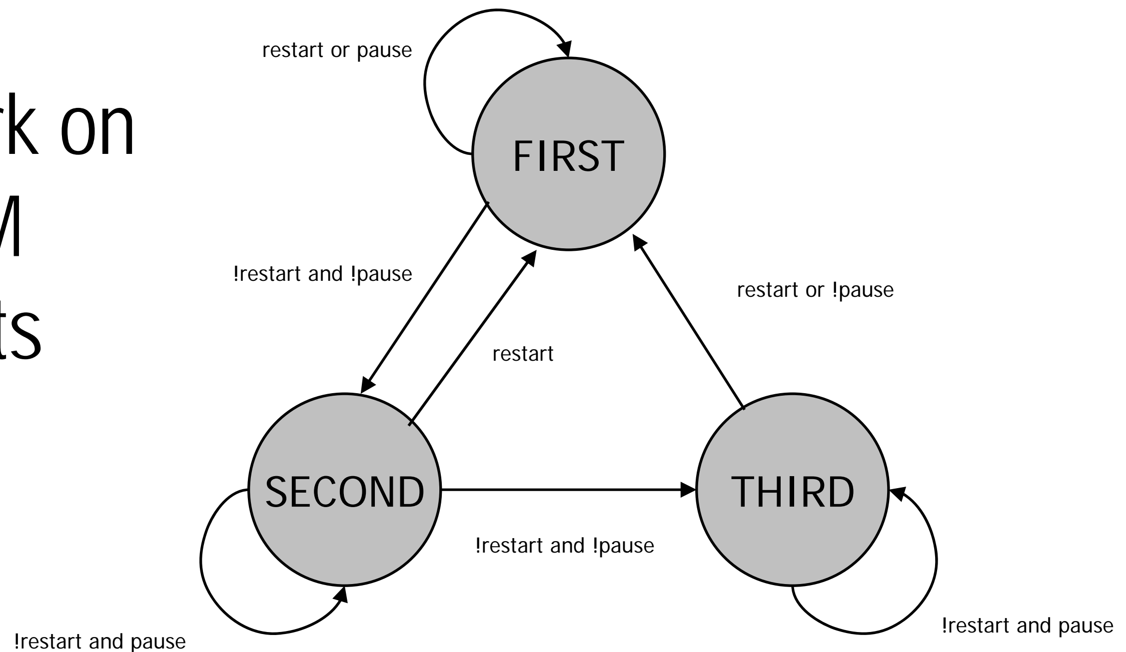
Finite State Machines

- Consider this edge-triggered, synchronous FSM to be implemented in digital logic:
  - A set of states
  - A set of input events
  - A function that maps states and inputs to states
  - A description of the initial state



Finite State Machines

- Things that are not shown (yet):
  - A set of output events
  - A function that maps states and input to output
- For now, let's work on modeling the FSM without the outputs and output logic.



Finite State Machines

- The state variables must be able to represent at least three unique states for this FSM.
  - A flip flop has two unique states.
  - N flip flops can represent up 2^N unique states.
  - How many flip flops are required for three states?
    - One flip flop is not enough.
    - Two flip flops are minimally sufficient.
    - More flip flops may be used, if desired.

Finite State Machines

- Select a state encoding method:
  - Binary
  - Gray
  - Johnson
  - One Hot
  - Custom

State	Binary	Gray	Johnson	One Hot
0	3'b000	3'b000	4'b0000	8'b00000001
1	3'b001	3'b001	4'b0001	8'b00000010
2	3'b010	3'b011	4'b0011	8'b00000100
3	3'b011	3'b010	4'b0111	8'b00001000
4	3'b100	3'b110	4'b1111	8'b00010000
5	3'b101	3'b111	4'b1110	8'b00100000
6	3'b110	3'b101	4'b1100	8'b01000000
7	3'b111	3'b100	4'b1000	8'b10000000

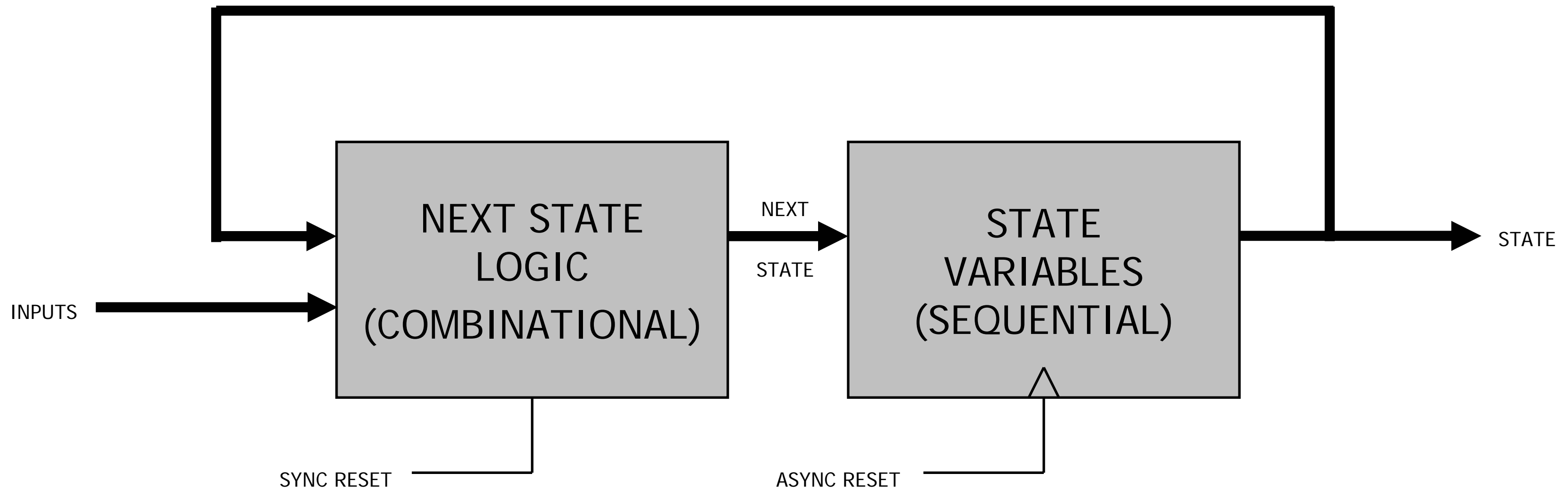
- Your encoding selection may require more than the minimally sufficient number of flip flops.

Finite State Machines

- Describe the state variables in Verilog.
- Provide a mechanism to force an initial state.
- Describe a function that maps inputs and current state to a new, or next state.
  - Literal transcription of excitation equations
  - Behavioral description using case, if-else, etc…
- Some additional things to consider:
  - Resets, synchronous or asynchronous?
  - Unused states (error, or no resets) and recovery

Finite State Machines

- Describe it in Verilog just like the block diagram!
- I have selected a custom state encoding.



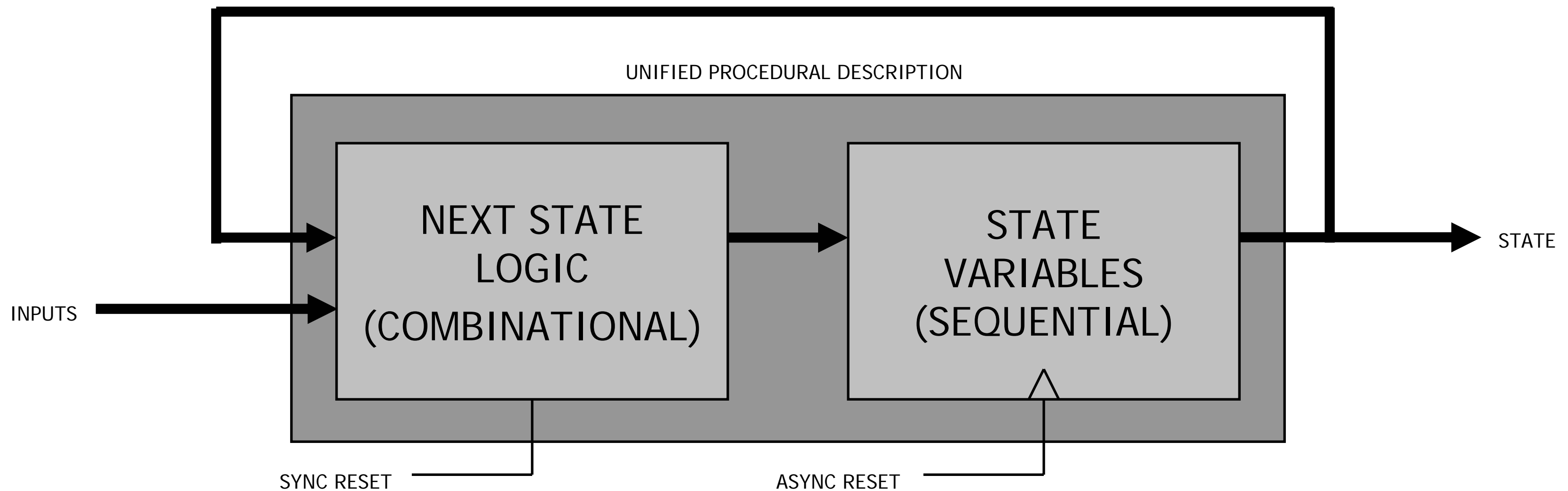
Finite State Machines

```
module fsm (
  input wire pause,
  input wire restart,
  input wire clk,
  input wire rst,
  output reg [1:0] state
  );

  reg [1:0] next_state;

  parameter [1:0] FIRST  = 2'b11;
  parameter [1:0] SECOND = 2'b01;
  parameter [1:0] THIRD  = 2'b10;

  always @(posedge clk or posedge rst) // sequential
  begin
    if (rst) state <= FIRST;
    else state <= next_state;
  end

  always @* // combinational
  begin
    case(state)
      FIRST:   if (restart | pause) next_state = FIRST;
               else next_state = SECOND;
      SECOND:  if (restart) next_state = FIRST;
               else if (pause) next_state = SECOND;
               else next_state = THIRD;
      THIRD:   if (!restart & pause) next_state = THIRD;
               else next_state = FIRST;
      default: next_state = FIRST;
    endcase
  end

endmodule
```

- Note use of parameters; easy to change encoding
- Asynchronous reset is implemented with state
- Synchronous reset is implemented with logic
- Default clause covers the one unused state
- Explicit next state signal

Finite State Machines

- You can also describe it in one procedural block.
  - No access to "next state" signal (important?)
  - More compact…



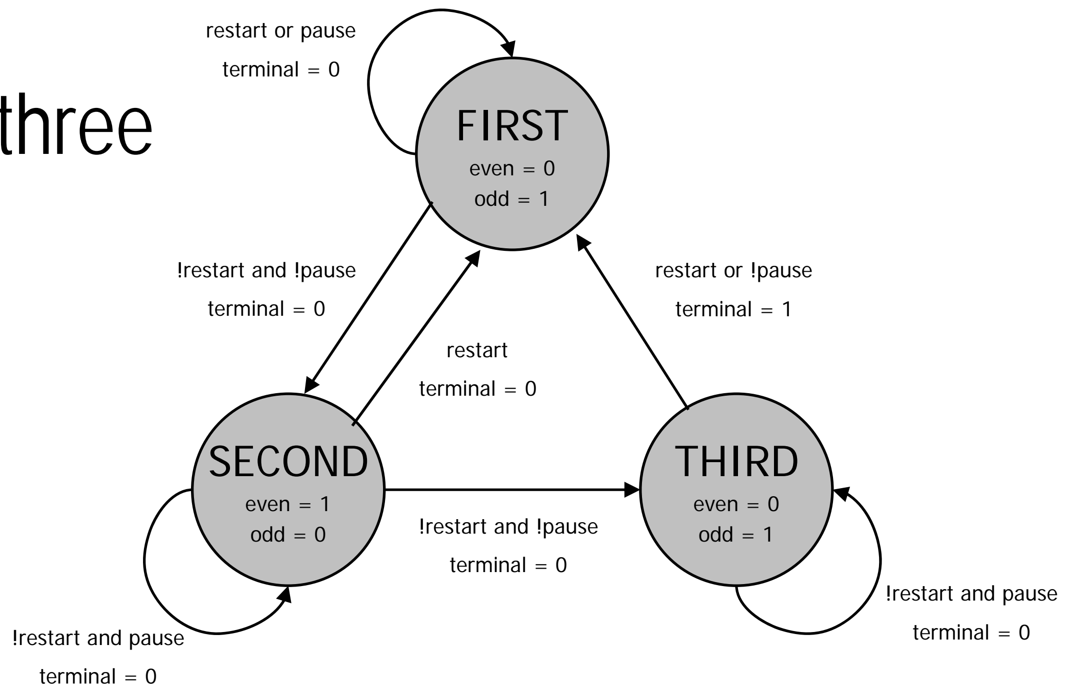
Finite State Machines

```
module fsm (
  input wire pause,
  input wire restart,
  input wire clk,
  input wire rst,
  output reg [1:0] state
  );

  parameter [1:0] FIRST  = 2'b11;
  parameter [1:0] SECOND = 2'b01;
  parameter [1:0] THIRD  = 2'b10;

  always @(posedge clk or posedge rst) // sequential
  begin
    if (rst) state <= FIRST;
    else
    begin
      case(state)
        FIRST:   if (restart | pause) state <= FIRST;
                 else state <= SECOND;
        SECOND:  if (restart) state <= FIRST;
                 else if (pause) state <= SECOND;
                 else state <= THIRD;
        THIRD:   if (!restart & pause) state <= THIRD;
                 else state <= FIRST;
        default: state <= FIRST;
      endcase
    end
  end

endmodule
```

- Note use of parameters; easy to change encoding
- Asynchronous reset and synchronous reset both implemented; distinction is made by sensitivity list
- Default clause covers the one unused state
- Implicit next state signal

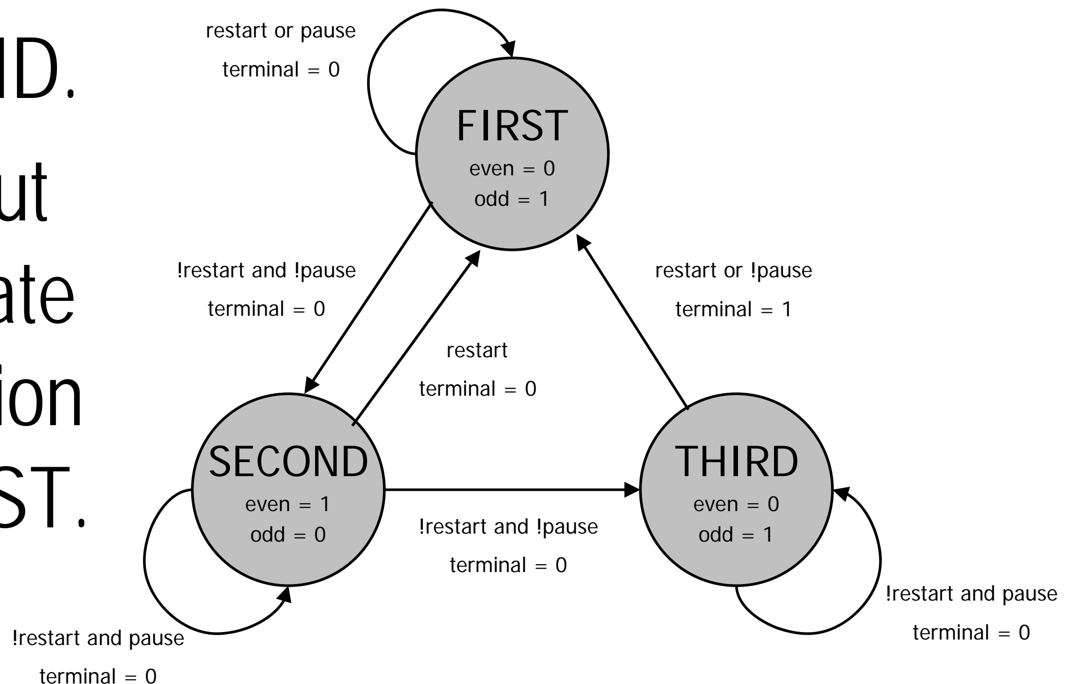
Finite State Machines

- Now, let's consider the following:
  - A set of output events
  - A function that maps states and input to output
- Suppose there are three desired outputs:
  - odd
  - even
  - terminal



Finite State Machines

- The “odd” output is asserted in FIRST and THIRD.
- The “even” output is asserted in SECOND.
- The “terminal” output is asserted to indicate the FSM will transition from THIRD to FIRST.

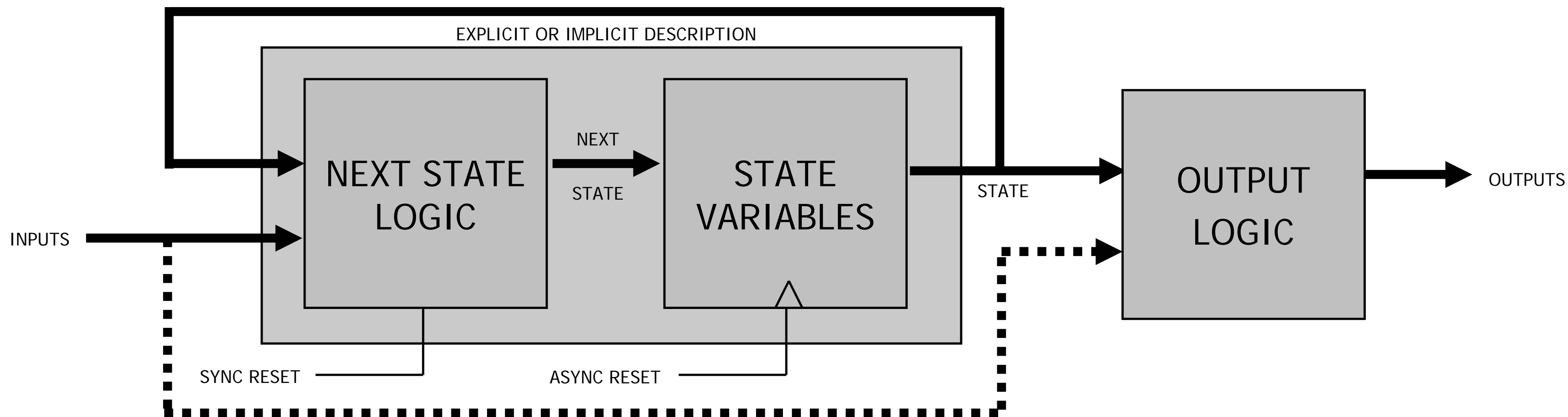


Finite State Machines

- Outputs that require functions of only the current state are Moore type outputs.
  - This includes using state bits directly.
  - Outputs “odd” and “even” are Moore outputs.
- Outputs that require functions of the current state and the inputs are Mealy type outputs.
  - Output “terminal” is a Mealy output.
- Consider the latency and cycle time tradeoffs.

Finite State Machines

- Describe the output functions in Verilog, just as shown in the block diagram...



Finite State Machines

```
module fsm (
  input wire pause,
  input wire restart,
  input wire clk,
  input wire rst,
  output reg [1:0] state,
  output wire odd,
  output wire even,
  output wire terminal
  );

  reg [1:0] next_state;

  parameter [1:0] FIRST  = 2'b11;
  parameter [1:0] SECOND = 2'b01;
  parameter [1:0] THIRD  = 2'b10;

  always @(posedge clk or posedge rst) // sequential
  begin
    if (rst) state <= FIRST;
    else state <= next_state;
  end

  always @* // combinational
  begin
    case(state)
      FIRST:   if (restart | pause) next_state = FIRST;
               else next_state = SECOND;
      SECOND:  if (restart) next_state = FIRST;
               else if (pause) next_state = SECOND;
               else next_state = THIRD;
      THIRD:   if (!restart & pause) next_state = THIRD;
               else next_state = FIRST;
      default: next_state = FIRST;
    endcase
  end

  // output logic described using continuous assignment
  assign odd = (state == FIRST) | (state == THIRD);
  assign even = (state == SECOND);
  assign terminal = (state == THIRD) & (restart | !pause);

endmodule
```

- Started with the FSM described using explicit next state logic, but could have used the other one.
- Added three assignment statements to create the output functions.

Finite State Machines

```
module fsm (
  input wire pause,
  input wire restart,
  input wire clk,
  input wire rst,
  output reg [1:0] state,
  output reg odd,
  output reg even,
  output reg terminal
  );

  parameter [1:0] FIRST  = 2'b11;
  parameter [1:0] SECOND = 2'b01;
  parameter [1:0] THIRD  = 2'b10;

  always @(posedge clk or posedge rst) // sequential
  begin
    if (rst) state <= FIRST;
    else
    begin
      case(state)
        FIRST:   if (restart | pause) state <= FIRST;
                 else state <= SECOND;
        SECOND:  if (restart) state <= FIRST;
                 else if (pause) state <= SECOND;
                 else state <= THIRD;
        THIRD:   if (!restart & pause) state <= THIRD;
                 else state <= FIRST;
        default: state <= FIRST;
      endcase
    end
  end

  // output logic described using procedural assignment
  always @* // combinational
  begin
    odd = (state == FIRST) | (state == THIRD);
    even = (state == SECOND);
    terminal = (state == THIRD) & (restart | !pause);
  end

endmodule
```

- Started with the FSM described using implicit next state logic, but could have used the other one.
- This describes the same output logic as before, but uses a procedural block to create the outputs.
  - Could have used case…
  - Could have used if-else…

XILINX®