FSM and Efficient Synthesizable FSM Design using Verilog

S. Rawat

Introduction

- There are many ways to code FSMs including many very poor ways to code FSMs.
- This lecture offers guidelines for doing efficient coding, simulation and synthesis of FSM designs.
- In this lecture multiple references are made to combinational always blocks and sequential always blocks.

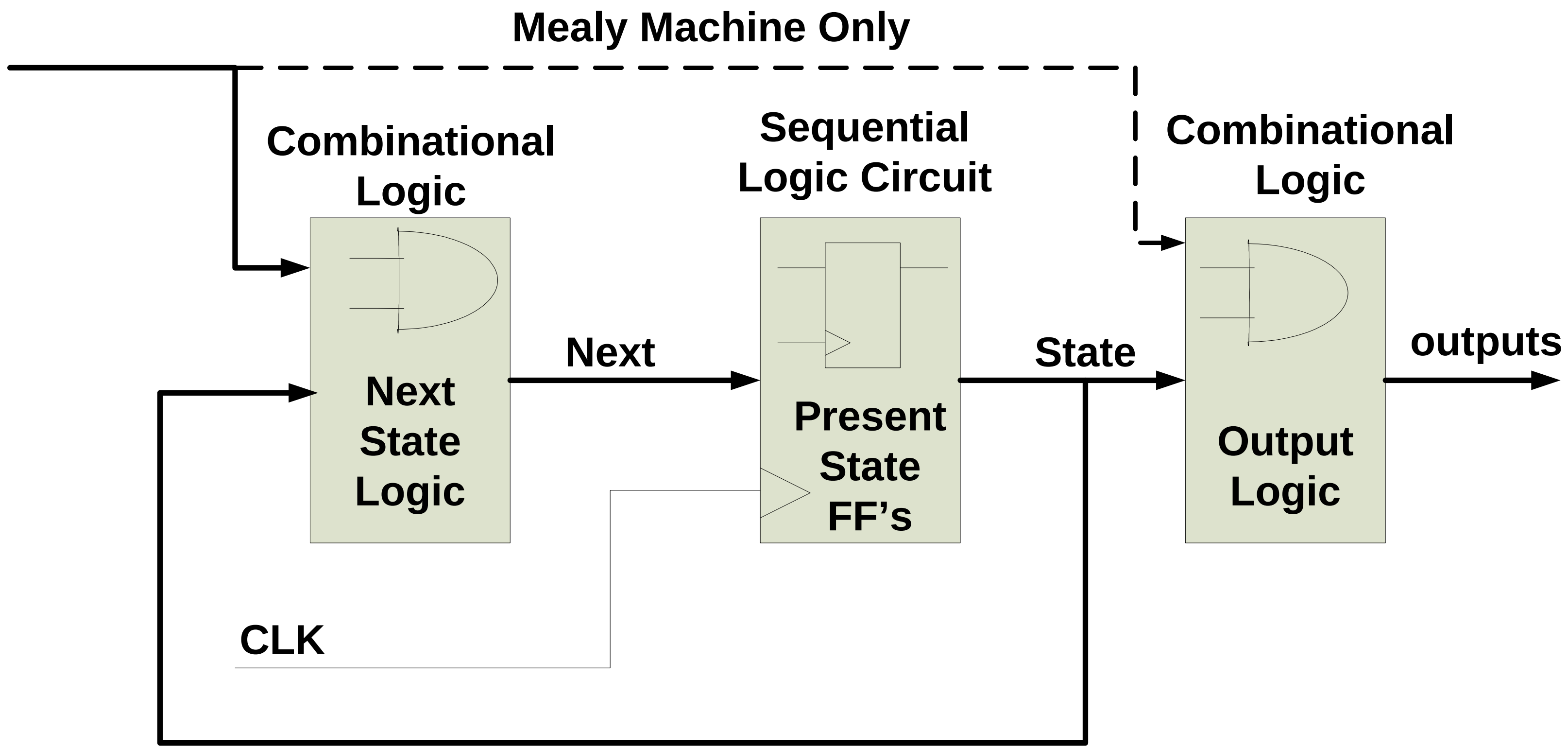
Introduction

- Combinational *always* blocks are *always* blocks that are used to code combinational logic functionality and are strictly coded using *blocking assignments*. A combinational always block has a combinational sensitivity list, a sensitivity list without *"posedge"* or *"negedge"* Verilog keywords.
- Sequential *always* blocks are *always* blocks that are used to code clocked or sequential logic and are always coded using *nonblocking* assignments. A sequential always block has an edge-based sensitivy list.

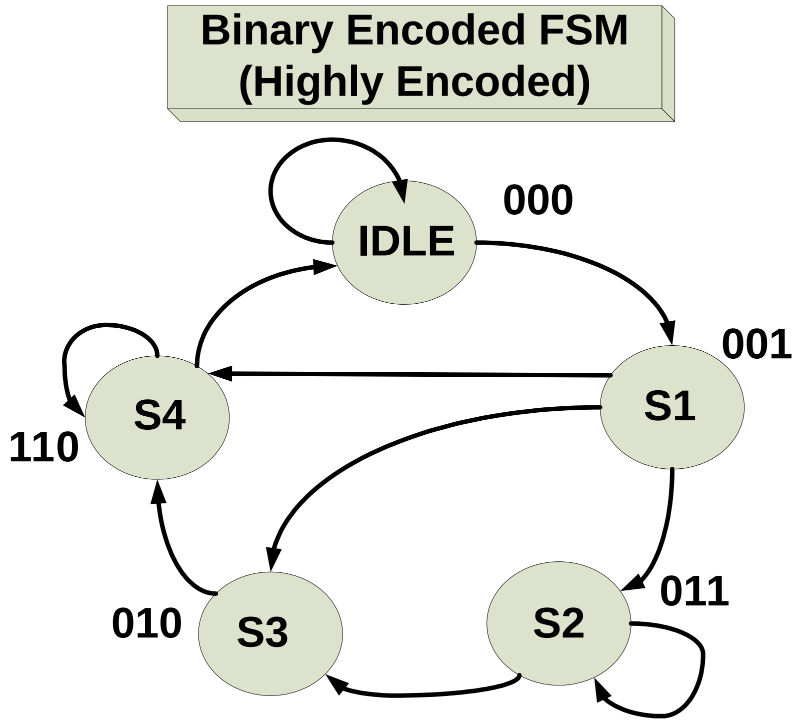
Mealy & Moore FSMs

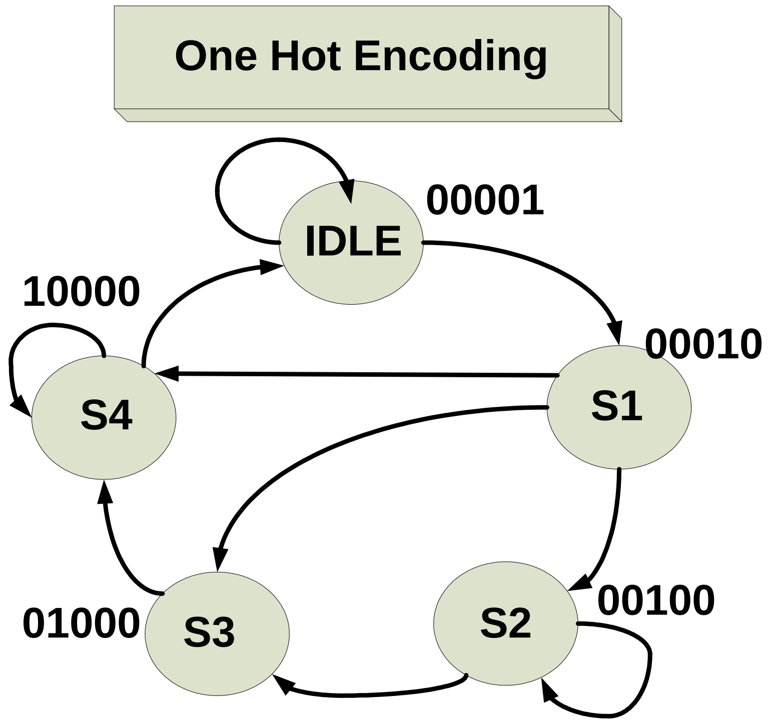
- A common classification used to describe the type of an FSM is Mealy and Moore state machines.
- A Moore FSM is a state machine where the outputs are only a function of the present state.
- A Mealy FSM is a state machine where one or more of the outputs is a function of the present state and one or more of the inputs.

Mealy & Moore FSMs (contd.)



Binary Encoded or One Hot Encoding





Binary Encoded or One Hot Encoding

- A binary-encoded FSM design only requires as many flip-flops as are needed to uniquely encode the number of states in the state machine.
- Number of FF

```
if(log2(#states) == integer)
        required FF = log2(#states)
else
required FF = integer(log2(#states))+1;
```

Binary Encoded or One Hot Encoding

- A onehot FSM design requires a flip-flop for each state in the design and only one flip-flop (the flip-flop representing the current or "hot" state) is set at a time in a onehot FSM design.
- For a state machine with 9-16 states, a binary FSM only requires 4 flip-flops while a onehot FSM requires a flip-flop for each state in the design (9-16 flip-flops).

FSM Coding Goals

- What constitutes an efficient FSM coding style?
- Identify HDL coding goals and why they are important.
- Quantify the capabilities of various FSM coding styles.

FSM Coding Goals

- The FSM coding style should be easily modified to change state encodings and FSM styles.
- The coding style should be compact.
- The coding style should be easy to code and understand.
- The coding style should facilitate debugging.
- The coding style should yield efficient synthesis results.

Two Always Block FSM Style (Good Style)

- One of the best Verilog coding styles is to code the FSM design using *two always blocks*, one for the *sequential state register* and one for the *combinational next-state* and *combinational output logic*.

Two Always Block FSM Style (Good Style)

```
module fsm_4states
            (output reg gnt,
            input dly, done, req, clk, rst_n);

parameter [1:0] IDLE = 2'b00,
BBUSY = 2'b01,
BWAIT = 2'b10,
BFREE = 2'b11;
reg [1:0] state, next;

always @(posedge clk or negedge
rst_n)
if (!rst_n) state <= IDLE;
else state <= next;
```

Two Always Block FSM Style (Good Style)

```
always @(state or dly or done or req) begin
next = 2'bx;
gnt = 1'b0;
case (state)
        IDLE : if (req) next = BBUSY;
        else next = IDLE;
        BBUSY: begin
        gnt = 1'b1;
        if (!done) next = BBUSY;
        else if ( dly) next = BWAIT;
        else next = BFREE;
        end
        BWAIT: begin
        gnt = 1'b1;
        if (!dly) next = BFREE;
        else next = BWAIT;
        end
        BFREE: if (req) next = BBUSY;
        else next = IDLE;
endcase end endmodule
```

Making default *next* equal all X's assignment

- Placing a default next state assignment on the line immediately following the always block sensitivity list is a very efficient coding style. This default assignment is updated by next-state assignments inside the case statement.
- There are three types of default next-state assignments that are commonly used: (1) next is set to all X's, (2) next is set to a predetermined recovery state such as IDLE, or (3) next is just set to the value of the state register.
- By making a default next state assignment of X's, pre-synthesis simulation models will cause the state machine outputs to go unknown if not all state transitions have been explicitly assigned in the case statement.
- This is a useful technique to debug state machine designs, plus the X's will be treated as "don't cares" by the synthesis tool.
- Some designs require an assignment to a known state as opposed to assigning X's. Examples include: satellite applications, medical applications, designs that use the FSM flip-flops as part of a diagnostic scan

One Always Block FSM Style (Avoid This Style!)

- One of the most common FSM coding styles in use today is the one sequential always block FSM coding style.
- For most FSM designs, the one always block FSM coding style is more verbose, more confusing and more error prone than a comparable two always block coding style.

One Always Block FSM Style:

```
module fsm_4states
            (output reg gnt,
            input dly, done, req, clk, rst_n);

parameter [1:0] IDLE = 2'd0,
BBUSY = 2'd1,
BWAIT = 2'd2,
BFREE = 2'd3;

reg [1:0] state;
```

One Always Block FSM Style:

```
always @(posedge clk or negedge rst_n)
        if (!rst_n) begin
        state <= IDLE;
        gnt <= 1'b0;
        end
        else begin
        state <= 2'bx;
        gnt <= 1'b0;

case (state)
        IDLE : if (req) begin
        state <= BBUSY;
        gnt <= 1'b1;
        end
        else
```

One Always Block FSM Style

```
BBUSY: if (!done) begin
           state <= BBUSY;
           gnt <= 1'b1;
           end
       else if ( dly) begin
           state <= BWAIT;
           gnt <= 1'b1;
           end
       else state <= BFREE;
BWAIT: if ( dly) begin
           state <= BWAIT;
           gnt <= 1'b1;
           end
       else state <= BFREE;
BFREE: if (req) begin
           state <= BBUSY;
           gnt <= 1'b1;
           end
           else state <= IDLE;
endcase
end
endmodule
```