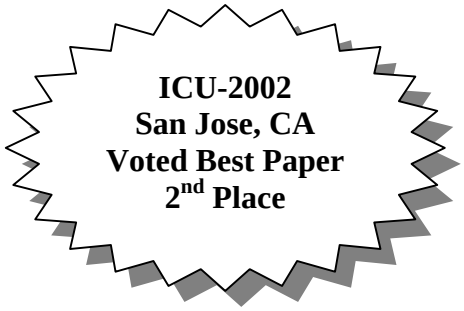
The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and BuildGates



Clifford E. Cummings

Sunburst Design, Inc.
503-641-8446
cliffc@sunburst-design.com

INTERNATIONAL CADENCE USERGROUP CONFERENCE
September 16-18, 2002
San Jose, California

Abstract

This paper details proven RTL coding styles for efficient and synthesizable Finite State Machine (FSM) design using IEEE-compliant Verilog simulators. Important techniques related to one and two always block styles to code FSMs with combinational outputs are given to show why using a two always block style is preferred. An efficient Verilog-unique onehot FSM coding style is also shown. Reasons and techniques for registering FSM outputs are also detailed. Myths surrounding erroneous state encodings, full-case and parallel-case usage are also discussed. Compliance and enhancements related to the IEEE 1364-2001 Verilog Standard, the proposed IEEE 1364.1 Verilog Synthesis Interoperability Standard and the proposed Accellera SystemVerilog Standard are also discussed.

1. Introduction

FSM is an abbreviation for *Finite State Machine*.

There are many ways to code FSMs including many very poor ways to code FSMs. This paper will examine some of the most commonly used FSM coding styles, their advantages and disadvantages, and offer guidelines for doing efficient coding, simulation and synthesis of FSM designs.

This paper will also detail Accellera SystemVerilog enhancements that will facilitate and enhance future Verilog FSM designs.

In this paper, multiple references are made to combinational always blocks and sequential always blocks. Combinational always blocks are always blocks that are used to code combinational logic functionality and are strictly coded using blocking assignments (see Cummings[4]). 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 (see Cummings[4]). A sequential always block has an edge-based sensitivy list.

2. Mealy and Moore FSMs

A common classification used to describe the type of an FSM is Mealy and Moore state machines[9][10].

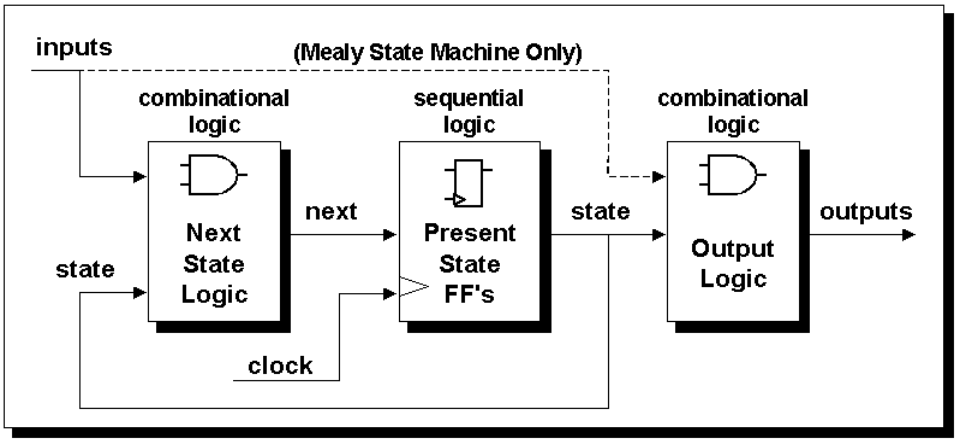


Figure 1 - Finite State Machine (FSM) block diagram

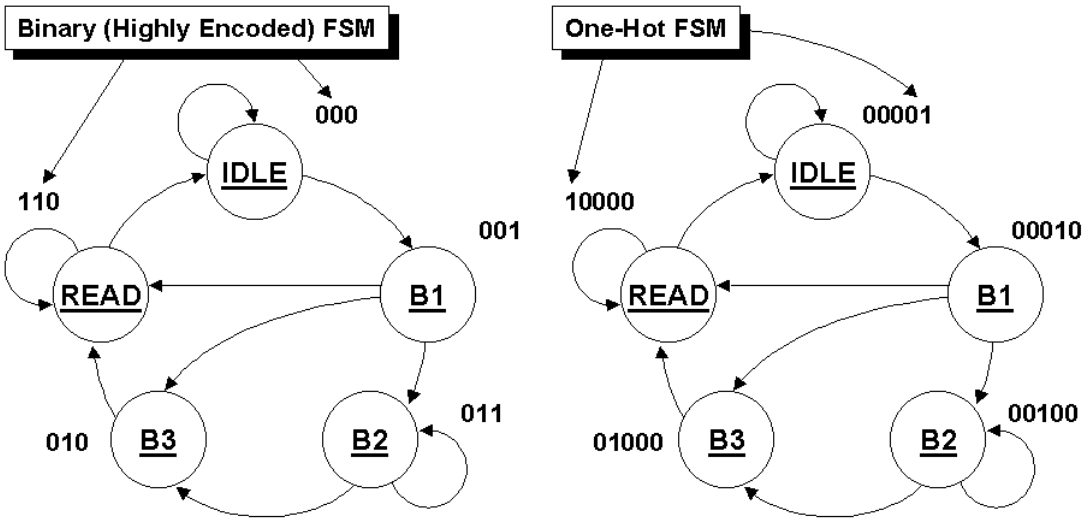
A Moore FSM is a state machine where the outputs are only a function of the present state. A Mealy FSM is a state machine where one or more of the outputs is a function of the present state and one or more of the inputs. A block diagram for Moore and Mealy FSMs is shown Figure 1.

3. Binary Encoded or Onehot Encoded?

Common classifications used to describe the state encoding of an FSM are Binary (or highly encoded) and Onehot.

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. The actual number of flip-flops required is equal to the ceiling of the log-base-2 of the number of states in the FSM.

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).



FPGA vendors frequently recommend using a onehot state encoding style because flip-flops are plentiful in an FPGA and the combinational logic required to implement a onehot FSM design is typically smaller than most binary encoding styles. Since FPGA performance is typically related to the combinational logic size of the FPGA design, onehot FSMs typically run faster than a binary encoded FSM with larger combinational logic blocks[8].

4. FSM Coding Goals

To determine what constitutes an efficient FSM coding style, we first need to identify HDL coding goals and why they are important. After the HDL coding goals have been identified, we can then quantify the capabilities of various FSM coding styles.

The author has identified the following HDL coding goals as important when doing HDL-based FSM design:

- 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.

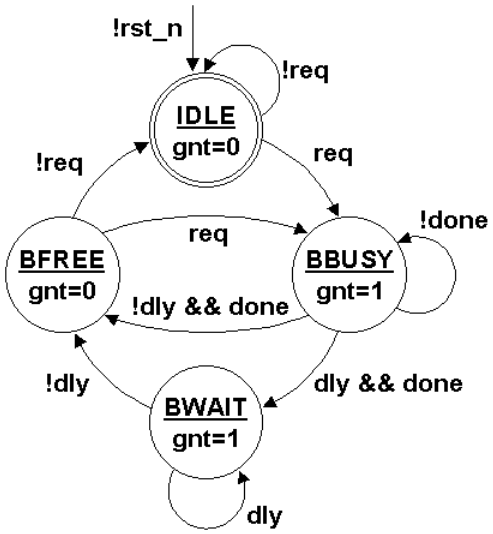
- The coding style should facilitate debugging.
- The coding style should yield efficient synthesis results.

Three different FSM designs will be examined in this paper. The first is a simple 4-state FSM design labeled fsm_cc4 with one output. The second is a 10-state FSM design labeled fsm_cc7 with only a few transition arcs and one output. The third is another 10-state FSM design labeled fsm_cc8 with multiple transition arcs and three outputs. The coding efforts to create these three designs will prove interesting.

5. 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.

```
module fsm_cc4_2
  (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;

  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
```



Example 1 - fsm_cc4 design - two always block style - 37 lines of code

5.1 Important coding style notes:

- Parameters are used to define state encodings instead of the Verilog `define macro definition construct[3]. After parameter definitions are created, the parameters are used throughout the rest of the

design, not the state encodings. This means that if an engineer wants to experiment with different state encodings, only the parameter values need to be modified while the rest of the Verilog code remains unchanged.

- Declarations are made for **`state`** and **`next`** (next state) after the parameter assignments.
- The sequential always block is coded using nonblocking assignments.
- The combinational always block sensitivity list is sensitive to changes on the **`state`** variable and all of the inputs referenced in the combinational always block.
- Assignments within the combinational always block are made using Verilog blocking assignments.
- The combinational always block has a default **`next`** state assignment at the top of the always block (see section 5.3 for details about making default-X assignments).
- Default output assignments are made before coding the **`case`** statement (this eliminates latches and reduces the amount of code required to code the rest of the outputs in the **`case`** statement and highlights in the **`case`** statement exactly in which states the individual output(s) change).
- In the states where the output assignment is not the default value assigned at the top of the always block, the output assignment is only made once for each state.
- There is an **`if`**-statement, an **`else`**-**`if`**-statement or an **`else`** statement for each transition arc in the FSM state diagram. The number of transition arcs between states in the FSM state diagram should equal the number of **`if`**-**`else`**-type statements in the combinational always block.
- For ease of scanning and debug, all of the **`next`** assignments have been placed in a single column, as opposed to finding **`next`** assignments following the contour of the RTL code.

5.2 The unfounded fear of transitions to erroneous states

In engineering school, we were all cautioned about "what happens if you FSM gets into an erroneous state?" In general, this concern is both invalid or poorly developed.

I do not worry about most of my FSM designs going to an erroneous state any more than I worry about any other register in my design spontaneously changing value. It just does not occur!

There are exceptions, such as satellites (subject to alpha particle bombardment) or medical implants (subject to radiation and requiring extra robust design), plus other examples. In these situations, one does have to worry about FSMs going to an erroneous state, but most engineering schools fail to note that getting back to a known state is typically not good enough! Even though the FSM is now in a known state, the rest of the hardware is still expecting activity related to another state. It is possible for the design to lockup waiting for signals that will never arrive because the FSM changed states without resetting the rest of the design. At the very least, the FSM should transition to an error state that communicates to the rest of the design that resetting will occur on the next state transition, "get ready!"

5.3 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.

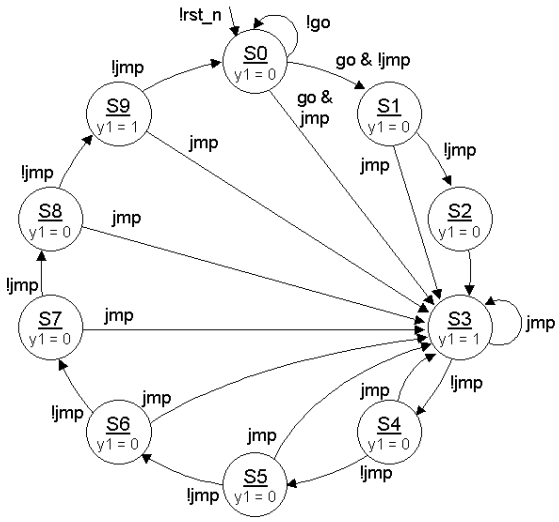
Some designs require an assignment to a known state as opposed to assigning X's. Examples include: satellite applications, medical applications, designs that use the FSM flip-flops as part of a diagnostic scan

chain and some designs that are equivalence checked with formal verification tools. Making a default next state assignment of either IDLE or all 0's typically satisfies these design requirements and making the initial default assignment might be easier than coding all of the explicit next-state transition assignments in the case statement.

5.4 10-state simple FSM design - two always blocks

Example 2 is the fsm_cc7 design implemented with two always blocks. Using two always blocks, the fsm_cc7 design requires 50 lines of code (coding requirements are compared in a later section).

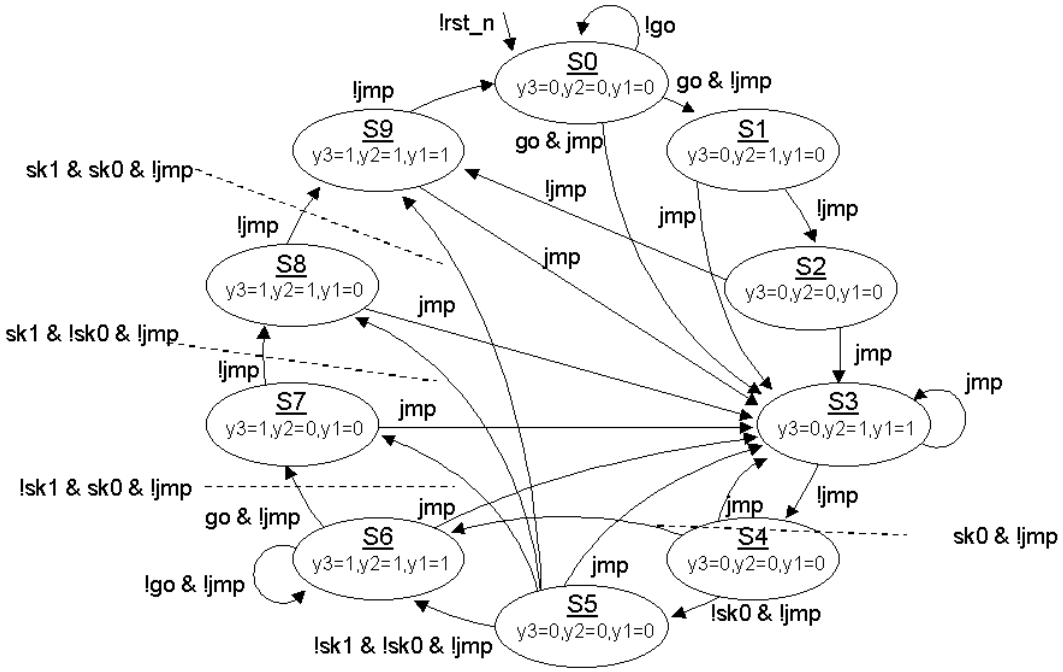
```
module fsm_cc7_2
        (output reg y1,
         input      jmp, go, clk, rst_n);

        parameter S0  = 4'b0000,
                  S1  = 4'b0001,
                  S2  = 4'b0010,
                  S3  = 4'b0011,
                  S4  = 4'b0100,
                  S5  = 4'b0101,
                  S6  = 4'b0110,
                  S7  = 4'b0111,
                  S8  = 4'b1000,
                  S9  = 4'b1001;

        reg [3:0] state, next;

        always @(posedge clk or negedge rst_n)
          if (!rst_n) state <= S0;
          else        state <= next;

        always @(state or go or jmp) begin
          next = 4'bx;
          y1 = 1'b0;
          case (state)
            S0 : if (!go)      next = S0;
                 else if (jmp) next = S3;
                 else          next = S1;
            S1 : if (jmp)      next = S3;
                 else          next = S2;
            S2 :               next = S3;
            S3 : begin y1 = 1'b1;
                   if (jmp)    next = S3;
                   else        next = S4;
                 end
            S4 : if (jmp)      next = S3;
                 else          next = S5;
            S5 : if (jmp)      next = S3;
                 else          next = S6;
            S6 : if (jmp)      next = S3;
                 else          next = S7;
            S7 : if (jmp)      next = S3;
                 else          next = S8;
            S8 : if (jmp)      next = S3;
                 else          next = S9;
            S9 : if (jmp)      next = S3;
                 else          next = S0;
          endcase
        end
      endmodule
```



Example 2 - fsm_cc7 design - two always block style - 50 lines of code

5.5 10-state moderately complex FSM design - two always blocks

Example 3 is the fsm_cc8 design implemented with two always blocks. Using two always blocks, the fsm_cc8 design requires 80 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_2
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or jmp or go or sk0 or sk1) begin
    next = 4'bx;
    y1 = 1'b0;
    y2 = 1'b0;
    y3 = 1'b0;
    case (state)
      S0 : if      (!go)                 next = S0;
           else if (jmp)                 next = S3;
```

```
            else                              next = S1;
       S1 : begin
              y2 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S2;
            end
       S2 : if (jmp)                          next = S3;
            else                              next = S9;
       S3 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S4;
              end
       S4 : if      (jmp)                     next = S3;
            else if (sk0 && !jmp)             next = S6;
            else                              next = S5;
       S5 : if      (jmp)                     next = S3;
            else if (!sk1 && !sk0 && !jmp)    next = S6;
            else if (!sk1 &&  sk0 && !jmp)    next = S7;
            else if ( sk1 && !sk0 && !jmp)    next = S8;
            else                              next = S9;
       S6 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              y3 = 1'b1;
              if      (jmp)                   next = S3;
              else if (go && !jmp)            next = S7;
              else                            next = S6;
            end
       S7 : begin
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S8;
            end
       S8 : begin
              y2 = 1'b1;
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S9;
            end
       S9 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S0;
            end
    endcase
  end
endmodule
```

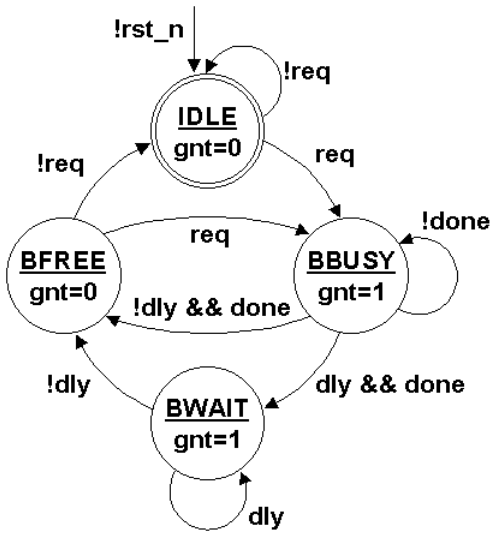
Example 3 - fsm_cc8 design - two always block style - 80 lines of code

6. 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. This coding style is very similar to coding styles that were popularized by PLD programming languages of the mid-1980s, such as ABEL. 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.

Reconsider the fsm_cc4 design shown in section 5.

```
module fsm_cc4_1
  (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;

  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              state <= IDLE;
        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
```



Example 4 - fsm_cc4 design - one always block style - 47 lines of code

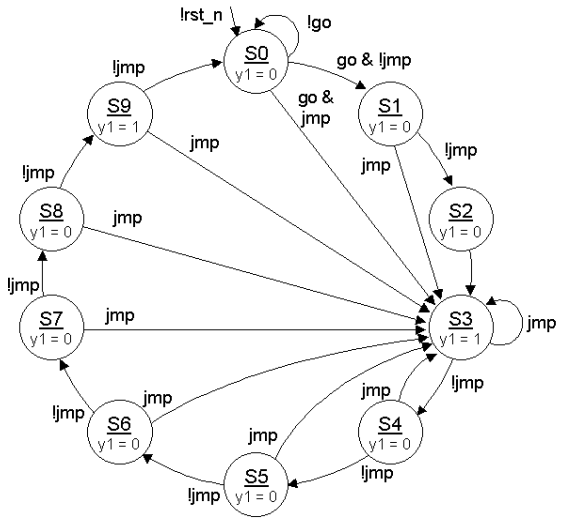
6.1 Important coding style notes:

- Parameters are used to define state encodings, the same as the two always block coding style.
- A declaration is made for **`state`**. Not for **`next`**.
- There is just one sequential always block, coded using nonblocking assignments.
- The there is still a default **`state`** assignment before the **`case`** statement, then the **`case`** statement tests the **`state`** variable. Will this be a problem? No, because the default **`state`** assignment is made with a nonblocking assignment, so the update to the **`state`** variable will happen at the end of the simulation time step.
- Default output assignments are made before coding the **`case`** statement (this reduces the amount of code required to code the rest of the outputs in the **`case`** statement).
- A **`state`** assignment must be made for each transition arc that transitions to a **`state`** where the output will be different than the default assigned value. For multiple outputs and for multiple transition arcs into a **`state`** where the outputs change, multiple **`state`** assignments will be required.
- The **`state`** assignments do not correspond to the current **`state`** of the **`case`** statement, but the **`state`** that **`case`** statement is transitioning to. *This is error prone* (but it does work if coded correctly).
- Again, for ease of scanning and debug, the all of the **`state`** assignments have been placed in a single column, as opposed to finding **`state`** assignments following the contour of the RTL code.
- All outputs will be registered (unless the outputs are placed into a separate combinational always block or assigned using continuous assignments). No asynchronous Mealy outputs can be generated from a single synchronous always block.
- Note: some misinformed engineers fear that making multiple assignments to the same variable, in the same always block, using nonblocking assignments, is undefined and can cause race conditions. This is not true. Making multiple nonblocking assignments to the same variable in the same always block is defined by the Verilog Standard. The last nonblocking assignment to the same variable wins! (See reference [5] for details).

6.2 10-state simple FSM design - one always blocks

Example 5 is the fsm_cc7 design implemented with one always blocks. Using one always blocks, the fsm_cc7 design requires 79 lines of code (coding requirements are compared in a later section).

```
module fsm_cc7_1
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state <= S0;
      y1 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
```

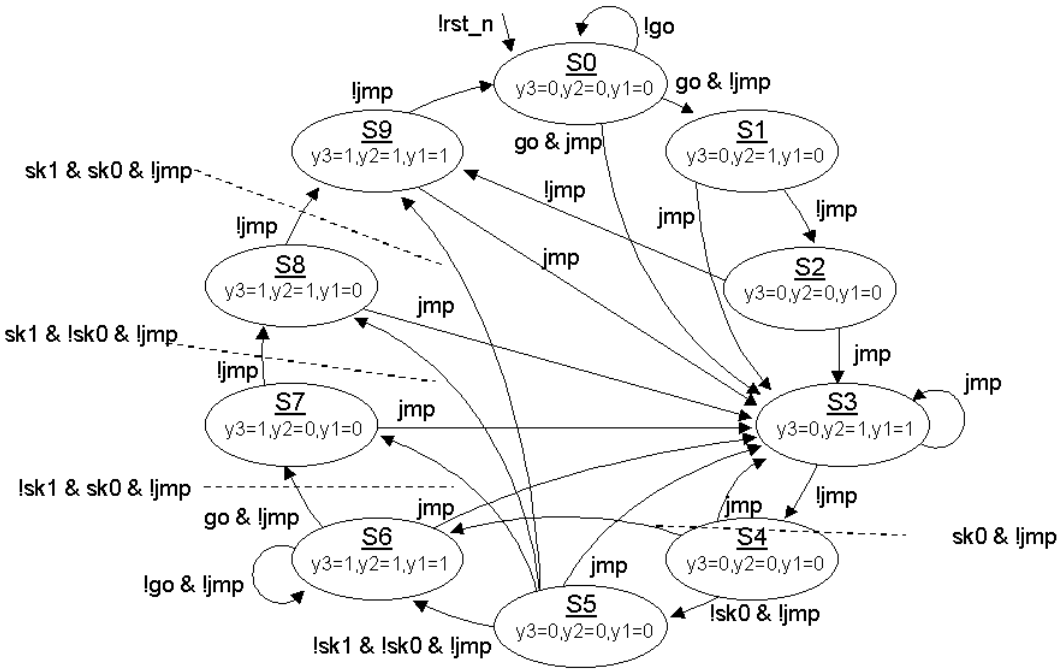


```
        state <= 4'bx;
        case (state)
          S0 : if (!go)      state <= S0;
               else if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S1;
          S1 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S2;
          S2 : begin
                 y1 <= 1'b1;
                             state <= S3;
               end
          S3 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S4;
          S4 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S5;
          S5 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S6;
          S6 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S7;
          S7 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S8;
          S8 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S9;
          S9 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S0;
        endcase
      end
    endmodule
```

Example 5 - fsm_cc7 design - one always block style - 79 lines of code

6.3 10-state moderately complex FSM design - one always blocks

Example 6 is the fsm_cc8 design implemented with one always blocks. Using one always blocks, the fsm_cc8 design requires 146 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_1
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                            state <= S0;
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
                            state <= 4'bx;
      y1 <= 1'b0;
      y2 <= 1'b0;
```

```
y3 <= 1'b0;
case (state)
  S0 : if (!go)      state <= S0;
       else if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else begin
                     state <= S1;
         y2 <= 1'b1;
       end
  S1 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else          state <= S2;
  S2 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else begin
                     state <= S9;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
  S3 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else          state <= S4;
  S4 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else if (sk0 && !jmp) begin
                     state <= S6;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
       else          state <= S5;
  S5 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else if (!sk1 && !sk0 && !jmp) begin
                     state <= S6;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
       else if (!sk1 && sk0 && !jmp) begin
                     state <= S7;
         y3 <= 1'b1;
       end
       else if (sk1 && !sk0 && !jmp) begin
```

```
                        state <= S8;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
                else begin
                        state <= S9;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S6 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else if (go && !jmp) begin
                        state <= S7;
                  y3 <= 1'b1;
                end
                else begin
                        state <= S6;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S7 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else begin
                        state <= S8;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S8 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else begin
                        state <= S9;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S9 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else          state <= S0;
        endcase
      end
endmodule
```

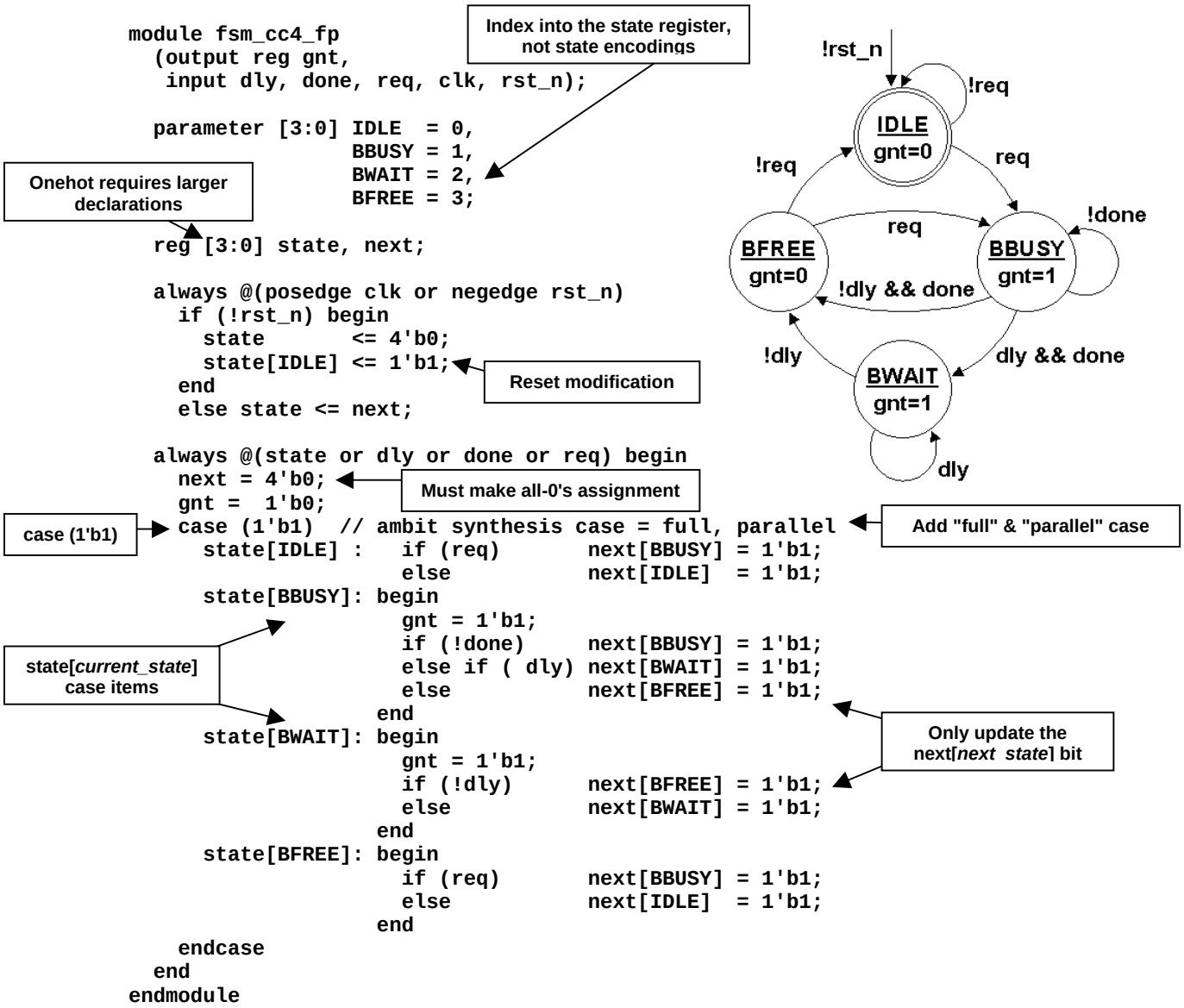
Example 6 - fsm_cc8 design - one always block style - 146 lines of code

7. Onehot FSM Coding Style (Good Style)

Efficient (small and fast) onehot state machines can be coded using an inverse case statement; a case statement where each case item is an expression that evaluates to true or false.

Reconsider the fsm_cc4 design shown in section 5. Eight coding modifications must be made to the two always block coding style of section 5 to implement the efficient onehot FSM coding style.

The key to understanding the changes is to realize that the parameters no longer represent **`state`** encodings, they now represent an *index* into the **`state`** vector, and comparisons and assignments are now being made to single bits in either the **`state`** or **`next`**-state vectors. Notice how the case statement is now doing a 1-bit comparison against the onehot state bit.

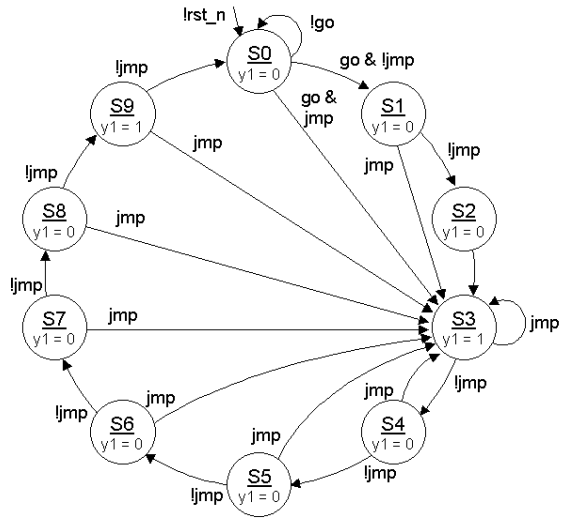



```
module fsm_cc4_fp
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [3:0] IDLE  = 0,
                  BBUSY = 1,
                  BWAIT = 2,
                  BFREE = 3;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state       <= 4'b0;
      state[IDLE] <= 1'b1;
    end
    else state <= next;

  always @(state or dly or done or req) begin
    next = 4'b0;
    gnt =  1'b0;
    case (1'b1)  // ambit synthesis case = full, parallel
      state[IDLE] :   if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
      state[BBUSY]: begin
                      gnt = 1'b1;
                      if (!done)     next[BBUSY] = 1'b1;
                      else if ( dly) next[BWAIT] = 1'b1;
                      else           next[BFREE] = 1'b1;
                    end
      state[BWAIT]: begin
                      gnt = 1'b1;
                      if (!dly)      next[BFREE] = 1'b1;
                      else           next[BWAIT] = 1'b1;
                    end
      state[BFREE]: begin
                      if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
                    end
    endcase
  end
endmodule
```

Example 7 - fsm_cc4 design - case (1'b1) onehot style - 42 lines of code

7.1 10-state simple FSM design - case (1'b1) onehot coding style

Example 8 is the fsm_cc7 design implemented with the case (1'b1) onehot coding style. Using this style, the fsm_cc7 design requires 53 lines of code (coding requirements are compared in a later section).

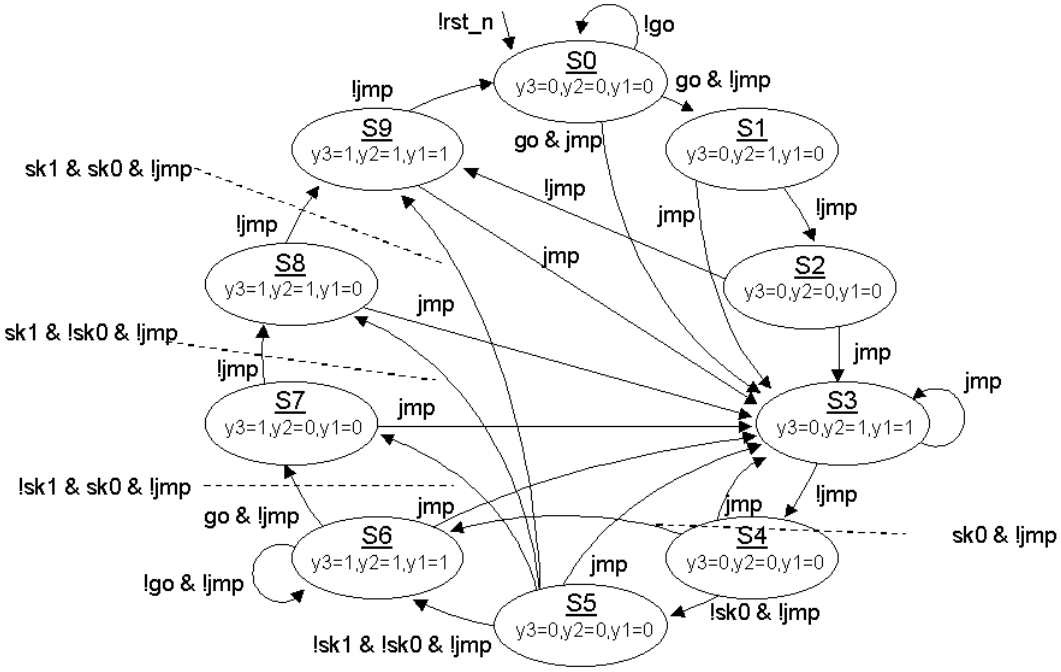



```
module fsm_cc7_onehot_fp
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 0,
            S1  = 1,
            S2  = 2,
            S3  = 3,
            S4  = 4,
            S5  = 5,
            S6  = 6,
            S7  = 7,
            S8  = 8,
            S9  = 9;

  reg [9:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                  state     <= 0;
                  state[S0] <= 1'b1;
                end
    else          state     <= next;

  always @(state or go or jmp) begin
    next = 10'b0;
    y1 = 1'b0;
    case (1'b1) // ambit synthesis case = full, parallel
      state[S0] : if (!go)      next[S0]=1'b1;
                  else if (jmp) next[S3]=1'b1;
                  else          next[S1]=1'b1;
      state[S1] : if (jmp)      next[S3]=1'b1;
                  else          next[S2]=1'b1;
      state[S2] :               next[S3]=1'b1;
      state[S3] : begin y1 = 1'b1;
                    if (jmp)    next[S3]=1'b1;
                    else        next[S4]=1'b1;
                  end
      state[S4] : if (jmp)      next[S3]=1'b1;
                  else          next[S5]=1'b1;
      state[S5] : if (jmp)      next[S3]=1'b1;
                  else          next[S6]=1'b1;
      state[S6] : if (jmp)      next[S3]=1'b1;
                  else          next[S7]=1'b1;
      state[S7] : if (jmp)      next[S3]=1'b1;
                  else          next[S8]=1'b1;
      state[S8] : if (jmp)      next[S3]=1'b1;
                  else          next[S9]=1'b1;
      state[S9] : if (jmp)      next[S3]=1'b1;
                  else          next[S0]=1'b1;
    endcase
  end
endmodule
```

Example 8 - fsm_cc7 design - case (1'b1) onehot style - 53 lines of code

7.2 10-state moderately complex FSM design - case (1'b1) onehot coding style

Example 9 is the fsm_cc8 design implemented with the case (1'b1) onehot coding style. Using this style, the fsm_cc8 design requires 86 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_onehot_fp
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0 = 0,
            S1 = 1,
            S2 = 2,
            S3 = 3,
            S4 = 4,
            S5 = 5,
            S6 = 6,
            S7 = 7,
            S8 = 8,
            S9 = 9;

  reg [9:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                state     <= 0;
                state[S0] <= 1'b1;
              end
    else          state     <= next;

  always @(state or jmp or go or sk0 or sk1) begin
    next = 0;
    case (1'b1) // ambit synthesis case = full, parallel
      state[S0] : if      (!go)                 next[S0]=1'b1;
                  else if (jmp)                 next[S3]=1'b1;
                  else                          next[S1]=1'b1;
```

```
      state[S1] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S2]=1'b1;
      state[S2] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S9]=1'b1;
      state[S3] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S4]=1'b1;
      state[S4] : if      (jmp)                  next[S3]=1'b1;
                  else if (sk0 && !jmp)          next[S6]=1'b1;
                  else                           next[S5]=1'b1;
      state[S5] : if      (jmp)                  next[S3]=1'b1;
                  else if (!sk1 && !sk0 && !jmp) next[S6]=1'b1;
                  else if (!sk1 &&  sk0 && !jmp) next[S7]=1'b1;
                  else if ( sk1 && !sk0 && !jmp) next[S8]=1'b1;
                  else                           next[S9]=1'b1;
      state[S6] : if      (jmp)                  next[S3]=1'b1;
                  else if (go && !jmp)           next[S7]=1'b1;
                  else                           next[S6]=1'b1;
      state[S7] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S8]=1'b1;
      state[S8] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S9]=1'b1;
      state[S9] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S0]=1'b1;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
      case (1'b1)
        next[S0], next[S2], next[S4], next[S5] : ; // default outputs
        next[S7] :   y3 <= 1'b1;
        next[S1] :   y2 <= 1'b1;
        next[S3] : begin
                     y1 <= 1'b1;
                     y2 <= 1'b1;
                   end
        next[S8] : begin
                     y2 <= 1'b1;
                     y3 <= 1'b1;
                   end
        next[S6], next[S9] : begin
                     y1 <= 1'b1;
                     y2 <= 1'b1;
                     y3 <= 1'b1;
                   end
    endcase
  end
endmodule
```

Example 9 - fsm_cc8 design - case (1'b1) onehot style - 86 lines of code

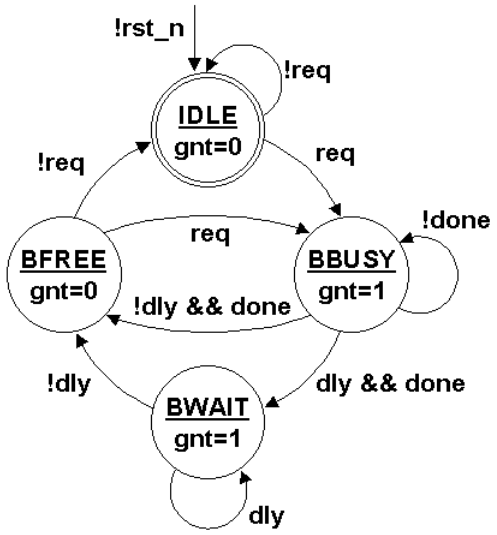
This is the only coding style where I recommend using full_case and parallel_case statements. The parallel case statement tells the synthesis tool to not build a priority encoder even though in theory, more than one of the state bits could be set (as engineers, we know that this is a onehot FSM and that only one bit can be set so no priority encoder is required). The value of the full_case statement is still in question.

8. Registered FSM Outputs (Good Style)

Registering the outputs of an FSM design insures that the outputs are glitch-free and frequently improves synthesis results by standardizing the output and input delay constraints of synthesized modules (see reference [1] for more information).

FSM outputs are easily registered by adding a third always sequential block to an FSM module where output assignments are generated in a case statement with case items corresponding to the next state that will be active when the output is clocked.

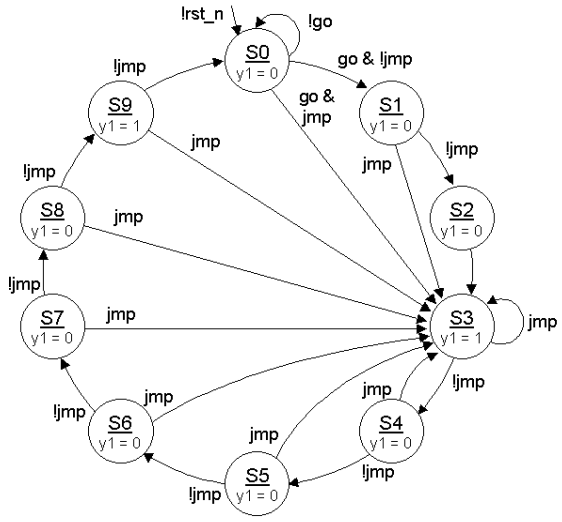
```
module fsm_cc4_2r
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  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;

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

  always @(posedge clk or negedge rst_n)
    if (!rst_n) gnt <= 1'b0;
    else begin
      gnt <= 1'b0;
      case (next)
        IDLE,  BFREE: ; // default outputs
        BBUSY, BWAIT: gnt <= 1'b1;
      endcase
    end
endmodule
```



Example 10 - fsm_cc4 design - three always blocks w/registered outputs - 40 lines of code

8.1 10-state simple FSM design - three always blocks - registered outputs

Example 11 is the fsm_cc7 design with registered outputs implemented with three always blocks. Using three always blocks, the fsm_cc7 design requires 60 lines of code (coding requirements are compared in a later section).



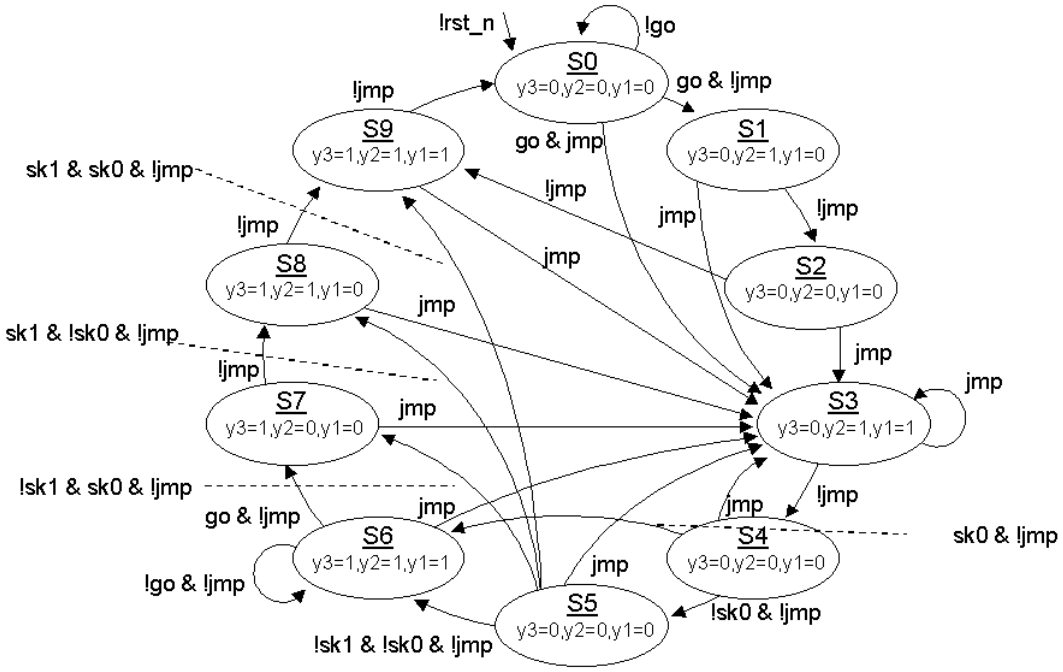

```
module fsm_cc7_3r
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or go or jmp) begin
    next = 4'bx;
    y1 = 1'b0;
    case (state)
      S0 : if (!go)      next = S0;
           else if (jmp) next = S3;
           else          next = S1;
      S1 : if (jmp)      next = S3;
           else          next = S2;
      S2 :               next = S3;
      S3 : begin y1 = 1'b1;
             if (jmp)    next = S3;
             else        next = S4;
           end
      S4 : if (jmp)      next = S3;
           else          next = S5;
      S5 : if (jmp)      next = S3;
           else          next = S6;
      S6 : if (jmp)      next = S3;
           else          next = S7;
      S7 : if (jmp)      next = S3;
           else          next = S8;
      S8 : if (jmp)      next = S3;
           else          next = S9;
      S9 : if (jmp)      next = S3;
           else          next = S0;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) y1 <= 1'b0;
    else begin
      y1 <= 1'b0;
      case (state)
        S0, S1, S2, S4, S5, S6, S7, S8, S9:; // default
```

```
        S3 : y1 <= 1'b1;
      endcase
    end
endmodule
```

Example 11 - fsm_cc7 design - three always blocks w/registered outputs - 60 lines of code

8.2 10-state moderately complex FSM design - three always blocks - registered outputs

Example 12 is the fsm_cc8 design with registered outputs implemented with three always blocks. Using three always blocks, the fsm_cc8 design requires 83 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_3r
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or jmp or go or sk0 or sk1) begin
```

```
    next = 4'bx;
    case (state)
      S0 : if      (!go)                 next = S0;
           else if (jmp)                 next = S3;
           else                          next = S1;
      S1 : if (jmp)                      next = S3;
           else                          next = S2;
      S2 : if (jmp)                      next = S3;
           else                          next = S9;
      S3 : if (jmp)                      next = S3;
           else                          next = S4;
      S4 : if      (jmp)                 next = S3;
           else if (sk0 && !jmp)         next = S6;
           else                          next = S5;
      S5 : if      (jmp)                 next = S3;
           else if (!sk1 && !sk0 && !jmp) next = S6;
           else if (!sk1 &&  sk0 && !jmp) next = S7;
           else if ( sk1 && !sk0 && !jmp) next = S8;
           else                          next = S9;
      S6 : if      (jmp)                 next = S3;
           else if (go && !jmp)          next = S7;
           else                          next = S6;
      S7 : if (jmp)                      next = S3;
           else                          next = S8;
      S8 : if (jmp)                      next = S3;
           else                          next = S9;
      S9 : if (jmp)                      next = S3;
           else                          next = S0;
    endcase
  end

always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
      case (next)
        S0, S2, S4, S5 : ; // default outputs
        S7             :   y3 <= 1'b1;
        S1             :   y2 <= 1'b1;
        S3             : begin
                           y1 <= 1'b1;
                           y2 <= 1'b1;
                         end
        S8             : begin
                           y2 <= 1'b1;
                           y3 <= 1'b1;
                         end
        S6, S9         : begin
                           y1 <= 1'b1;
                           y2 <= 1'b1;
                           y3 <= 1'b1;
                         end
    endcase
  end
endmodule
```

Example 12 - fsm_cc8 design - three always blocks w/registered outputs - 83 lines of code

9. Comparing RTL Coding Efforts

In the preceding sections, three different FSM designs were coded four different ways: (1) two always block coding style, (2) one always block coding style, (3) onehot, two always block coding style, and (4) three always block coding style with registered outputs.

	Two always block coding style	One always block coding style (12%-83% larger)	Onehot, two always block coding style	Three always block coding style w/ registered outputs
fsm_cc4 (4 states, simple)	37 lines of code	47 lines of code (12%-27% larger)	42 lines of code	40 lines of code
fsm_cc7 (10 states, simple)	50 lines of code	79 lines of code (32%-58% larger)	53 lines of code	60 lines of code
fsm_cc8 (10 states, moderate complexity)	80 lines of code	146 lines of code (70%-83% larger)	86 lines of code	83 lines of code

Table 1 - Lines of RTL code required for different FSM coding styles

From Table 1, we see that the one always block FSM coding style is the least efficient coding style with respect to the amount of RTL code required to render an equivalent design. In fact, the more outputs that an FSM design has and the more transition arcs in the FSM state diagram, and thus the faster the one always block coding style increases in size over comparable FSM coding styles.

If you are a contractor or are paid by the line-of-code, clearly, the one always block FSM coding style should be your preferred style. If you are trying to complete a project on time and code the design in a concise manner, the one always block coding style should be avoided.

10. Synthesis Results

Synthesis results were not complete by the time the paper was submitted for publication.

11. Running Cadence BuildGates

```
ac_shell          (for command-line mode)
ac_shell -gui &   (for GUI mode with process running  in background)
```

12. Verilog-2001 Enhancements

As of this writing, the Cadence Verilog simulators do not support many (if any) of the new Verilog-2001 enhancements. All of the preceding examples were coded with Verilog-2001 enhanced and concise ANSI-style module headers. In reality, to make the designs work with the Cadence Verilog simulators, I had to also code Verilog-1995 style module headers and select the appropriate header using macro definitions. To ease the task, I have created two aliases for 1995-style Verilog simulations.

```
alias ncverilog95 "ncverilog +define+V95"
alias verilog95 "verilog +define+V95"
```

12.1 ANSI-Style port declarations

ANSI-style port declarations are a nice enhancement to Verilog-2001 but they are not yet supported by version 3.4 of NC-Verilog or Verilog-XL, but they are reported to work with BuildGates. This enhancement permits module headers to be declared in a much more concise manner over traditional Verilog-1995 coding requirements.

Verilog-1995 required each module port be declared two or three times. Verilog-1995 required that (1) the module ports be listed in the module header, (2) the module port directions be declared, and (3) for reg-variable output ports, the port data type was also required.

Verilog-2001 combined all of this information into single module port declarations, significantly reducing the verbosity and redundancy of Verilog module headers. Of the major Verilog vendors, only the Cadence Verilog simulators do not support this Verilog-2001 feature. This means that users who want to take advantage of this feature and who use simulators from multiple vendors, including Cadence, must code both styles of module headers using `ifdef statements to select the appropriate module header style.

I prefer the following coding style to support retro-style Verilog simulators:

```
`ifdef V95
  // Verilog-1995 old-style, verbose module headers
`else
  // Verilog-2001 new-style, efficient module headers
`endif
```

The following example is from the actual fsm_cc4_1.v file used to test one always block FSM coding styles in this paper.

```
`ifdef V95
module fsm_cc4_1 (gnt, dly, done, req, clk, rst_n);
  output gnt;
  input  dly, done, req;
  input  clk, rst_n;
  reg    gnt;
`else
module fsm_cc4_1
  (output reg gnt,
   input dly, done, req, clk, rst_n);
`endif
```

It should be noted that this is an easy enhancement to implement, significantly improves the coding efficiency of module headers and that some major Verilog vendors have supported this enhanced coding style for more than a year at the time this paper was written. The author strongly encourages Cadence simulator developers to quickly adopt this Verilog-2001 enhancement to ease the Verilog coding burden for Cadence tool users.

12.2 @* Combinational sensitivity list

Verilog-2001 added the much-heralded @* combinational sensitivity list token. Although the combinational sensitivy list could be written using any of the following styles:

```
always @*
always @(*)
always @( * )
always @ ( * )
```

or any other combination of the characters @ (*) with or without white space, the author prefers the first and most abbreviated style. To the author, "always @*" clearly denotes that a combinational block of logic follows.

The Verilog-2001 "always @*" coding style has a number of important advantages over the more cumbersome Verilog-1995 combinational sensitivity list coding style:

- Reduces coding errors - the code informs the simulator that the intended implementation is combinational logic, so the simulator will automatically add and remove signals from the sensitivity list as RTL code is added or deleted from the combinational always block. The RTL coder is no longer burdened with manually insuring that all of the necessary signals are present in the sensitivity list. This will reduce coding errors that do not show up until a synthesis tool or linting tool reports errors in the sensitivity list. The basic intent of this enhancement is to inform the simulator, "if the synthesis tool wants the signals, so do we!"

- Abbreviated syntax - large combinational blocks often meant multiple lines of redundant signal naming in a sensitivity list. The redundancy served no appreciable purpose and users will gladly adopt the more concise and abbreviated @* syntax.

- Clear intent - an always @* procedural block informs the code-reviewer that this block is intended to behave like, and synthesize to, combinational logic.

13. SystemVerilog Enhancements

In June of 2002, Accellera released the SystemVerilog 3.0 language specification, a superset of Verilog-2001 with many nice enhancements for modeling, synthesis and verification. The basis for the SystemVerilog language comes from a donation by CoDesign Automation of significant portions of their Superlog language.

Key functionality that has been added to the Accellera SystemVerilog 3.0 Specification to support FSM design includes:

Enumerated types - Why do engineers want to use enumerated types? (1) Enumerated types permit abstract state declaration without defining the state encodings, and (2) enumerated types can typically be easily displayed in a waveform viewer permitting faster design debug. Enumerated types allow abstract state definitions without required state encoding assignments. Users also wanted the ability to assign state encodings to control implementation details such as output encoded FSM designs with simple registered outputs.

One short coming of traditional enumerated types was the inability to make X-state assignments. As discussed earlier in this paper, X-state assignments are important to simulation debug and synthesis optimization. SystemVerilog enumerated types will permit data type declaration, making it possible to declare enumerated types with an all-X's definitions.

Other SystemVerilog proposals under consideration for FSM enhancement include:

Different enumerated styles - the ability to declare different enumerated styles, such as enum_onehot, to make experimentation with different encoding styles easier to do. Currently, when changing from a binary encoding to an efficient onehot encoding style, 8 different code changes must be made in the FSM module. Wouldn't it be nice if the syntax permitted easier handling of FSM styles without manual intervention.

Transition statement and ->> next state transition operator -

These enhancements were removed from the SystemVerilog 3.0 Standard only because their definition was not fully elaborated and understood. Some people like the idea of a next-state transition operator that closely corresponds to the transition arcs that are shown on an FSM state diagram.

The infinitely abusable "goto" statement - Concern about a "goto" statement that could "cause spaghetti-code" could be avoided by limiting a goto-transition to a label within the same procedural block. Implicit FSM coding styles are much cleaner with a goto statement. A goto statement combined with a carefully crafted disable statement makes reset handling easier to do. A goto statement alleviates the problem of multiple transition arcs within a traditional implicit FSM design. Goto is just a proposal and may not pass.

14. Conclusions

There are many ways to code FSM designs. There are many inefficient ways to code FSM designs!

Use parameters to define state encodings. Parameters are constants that are local to a module. After defining the state encodings at the top of the FSM module, never use the state encodings again in the RTL code. This makes it possible to easily change the state encodings in just one place, the parameter definitions, without having to touch the rest of the FSM RTL code. This makes state-encoding-experimentation easy to do.

Use a two always block coding style to code FSM designs with combinational outputs. This style is efficient and easy to code and can also easily handle Mealy FSM designs.

Use a three always block coding style to code FSM designs with registered outputs. This style is efficient and easy to code. Note, another recommended coding style for FSM designs with registered outputs is the "output encoded" FSM coding style (see reference [1] for more information on this coding style).

Avoid the one always block FSM coding style. It is generally more verbose than an equivalent two always block coding style, output assignments are more error prone to coding mistakes and one cannot code asynchronous Mealy outputs without making the output assignments with separate continuous assign statements.

15. Acknowledgements

I would like to especially thank both Rich Owen and Nasir Junejo of Cadence for their assistance and tips enabling the use of the BuildGates synthesis tool. Their input helped me to achieve very favorable results in a short period of time.

16. References

[1] Clifford E. Cummings, "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs," *SNUG'2000 Boston (Synopsys Users Group Boston, MA, 2000) Proceedings*, September 2000. (Also available online at www.sunburst-design.com/papers)

[2] Clifford E. Cummings, "'full_case parallel_case", the Evil Twins of Verilog Synthesis,' *SNUG'99 Boston (Synopsys Users Group Boston, MA, 1999) Proceedings*, October 1999. (Also available online at www.sunburst-design.com/papers)

[3] Clifford E. Cummings, "New Verilog-2001 Techniques for Creating Parameterized Models (or Down With `define and Death of a defparam!)," *International HDL Conference 2002 Proceedings*, pp. 17-24, March 2002. (Also available online at www.sunburst-design.com/papers)

[4] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," *SNUG'2000 Boston (Synopsys Users Group San Jose, CA, 2000) Proceedings*, March 2000. (Also available online at www.sunburst-design.com/papers)

[5] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE Std 1364-1995, pg. 47, section 5.4.1 - Determinism.

[6] Nasir Junejo, personal communication

[7] Rich Owen, personal communication

[8] The Programmable Logic Data Book, Xilinx, 1994, pg. 8-171

[9] William I. Fletcher, An Engineering Approach To Digital Design, New Jersey, Prentice-Hall, 1980

[10] Zvi Kohavi, Switching And Finite Automata Theory, Second Edition, New York, McGraw-Hill Book Company, 1978

Revision 1.2 (July 2004) - What Changed?

Version 1.1 of the paper had misspelled the name of the Data IO PLD programming language in Section 6. The corrected spelling is ABEL. My thanks to a reader who found and reported this mistake. Also, the [10] reference title was corrected.

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, chaired the VSG Behavioral Task Force, which was charged with proposing enhancements to the Verilog language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group and the Accellera SystemVerilog Standardization Group.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

E-mail Address: cliffc@sunburst-design.com
An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers (Data accurate as of July 22nd, 2002)