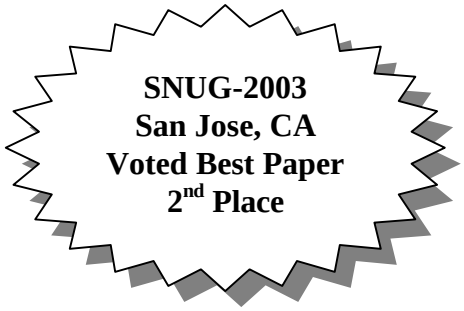
Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements



Clifford E. Cummings

Sunburst Design, Inc.

ABSTRACT

This paper details RTL coding and synthesis techniques of Finite State Machine (FSM) design using new Accellera SystemVerilog 3.0 capabilities. Efficient existing RTL coding styles are compared to new SystemVerilog 3.0 enhanced coding styles. SystemVerilog 3.0 enumerated type and implicit port connection enhancements are emphasized in this paper, along with background information detailing reasons why the SystemVerilog 3.0 enhancements were implemented as defined.

1.0 Introduction

The Accellera SystemVerilog 3.0 Standard[11], released at DAC 2002, includes many enhancements to the IEEE Verilog-2001 hardware description language[7]. A few of these enhancements were added to assist in the efficient development of Finite State Machine (FSM) designs.

The SystemVerilog enhancements were not only added to improve RTL coding capability, but also to improve simulation debug and synthesis capabilities.

Before you can code an efficient FSM design using SystemVerilog 3.0 RTL enhancements, you need to know how to code efficient Verilog-2001 FSM designs. Section 2.0 shows efficient Verilog-2001 styles for coding FSM designs and Sections 10.0 shows and details SystemVerilog enhancements for FSM design.

Section 8.0 also details Synopsys DC 2002.05 FSM Compiler enhancements and their impact on standard FSM designs.

1.1 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 modifiable 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.

1.2 Important coding style notes:

There are a few FSM coding guidelines that apply to all FSM coding styles[2]. The common guidelines are:

Guideline: Make each FSM design a separate Verilog module.

It is easier to maintain the FSM code if each FSM is a separate module, plus third-party FSM optimization tools work best on isolated and self-contained FSM designs.

Guideline: Use parameters to define state encodings instead of the Verilog **`define** macro definition construct.

If an engineer creates a large ASIC or FPGA design with multiple state machines, it is not uncommon to reuse state names such as IDLE or READ as shown in Figure 1. If **`define** macro definitions are used to code these FSM designs, The compiler will report "macro redefinition" warnings and any testbench that probes the internal FSM designs to extract state information will only have access to the last `IDLE or `READ state. Parameters are constants that are local to a module and whenever a constant is added to a design, an engineer should think "use parameters" first, and only use a global macro definition if the macro is truly needed by multiple modules or the entire design.

Guideline: When creating Verilog constants, think parameters first, then find good justification before changing to use a global **`define** macro.

Most Verilog constants should be coded using parameters0.

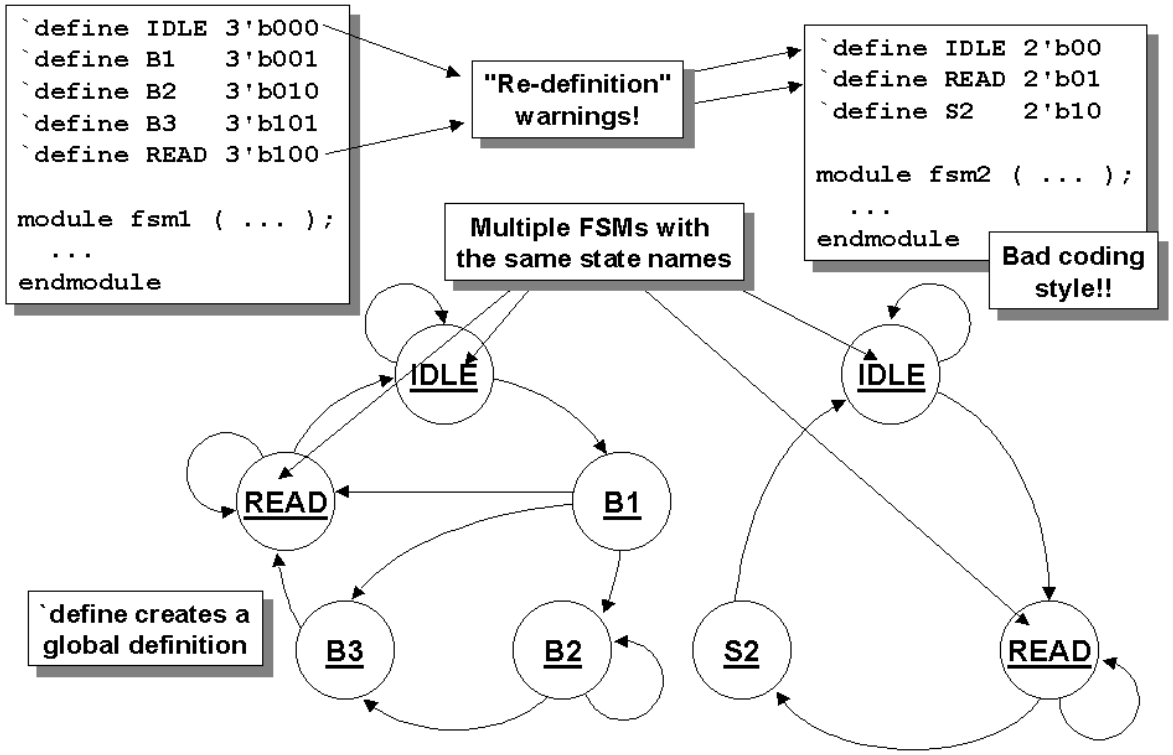


Figure 1 - FSM design using `define macro definitions - NOT Recommended

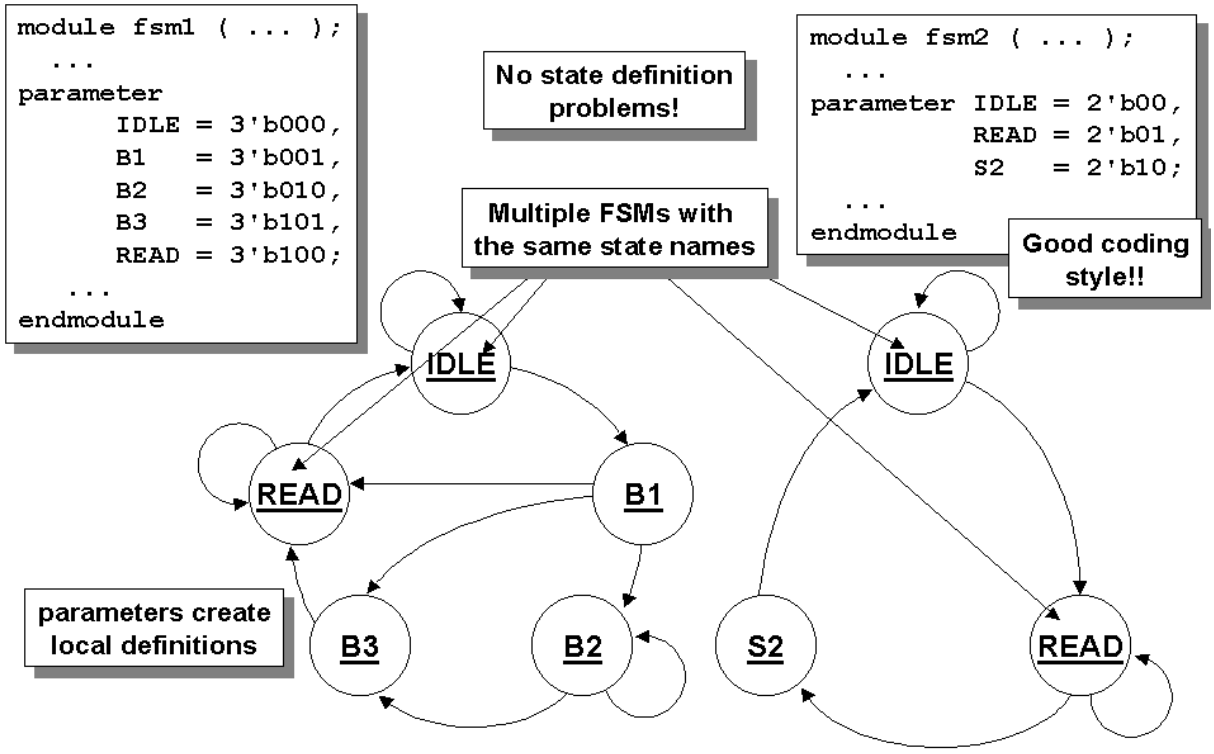


Figure 2 - FSM design using parameters - Recommended

After parameter definitions are created, the symbolic parameter names 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 at the top of the module need to be modified while the rest of the Verilog code remains unchanged.

Guideline: make **`state`** and **`next`** (next state) declarations right after the parameter assignments.

Some FSM-optimization tools require state parameter assignments to be declared before making the state and next declarations. It is a good coding style and a good habit to make these declarations right after the state name parameter declarations.

Guideline: Code all sequential always block using nonblocking assignments.

Guideline: Code all combinational always block using blocking assignments.

These two guidelines help to code a design that will not be vulnerable to Verilog simulation race conditions[4].

2.0 Review of standard Verilog FSM coding styles

There are proven coding styles to efficiently implement Verilog FSM designs. This section will show a two always block style that implements an FSM design with combinational outputs and then five more styles will be shown to implement an FSM design with registered outputs.

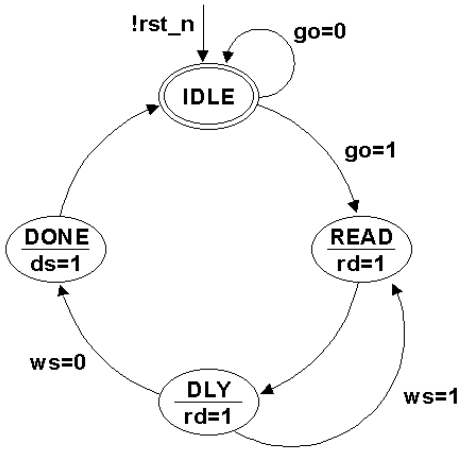


Figure 3 - fsm1 - Simple 4-state FSM design

The example used in this section to examine different FSM coding styles will be a simple 4-state **`fsm1`** design, with asynchronous low-true **`rst_n`** signal, a clock signal named **`clk`**, two inputs named **`go`** and **`ws`** (wait state) and two outputs name **`rd`** (read) and **`ds`** (done strobe). Except where noted on the diagram, the outputs **`rd`** and **`ds`** equal **`0`**.

The state diagram for the **`fsm1`** design is shown in Figure 3.

2.1 Two always block style with combinational outputs (Good Style)

The FSM block diagram for a two always block coding style is shown in Figure 4.

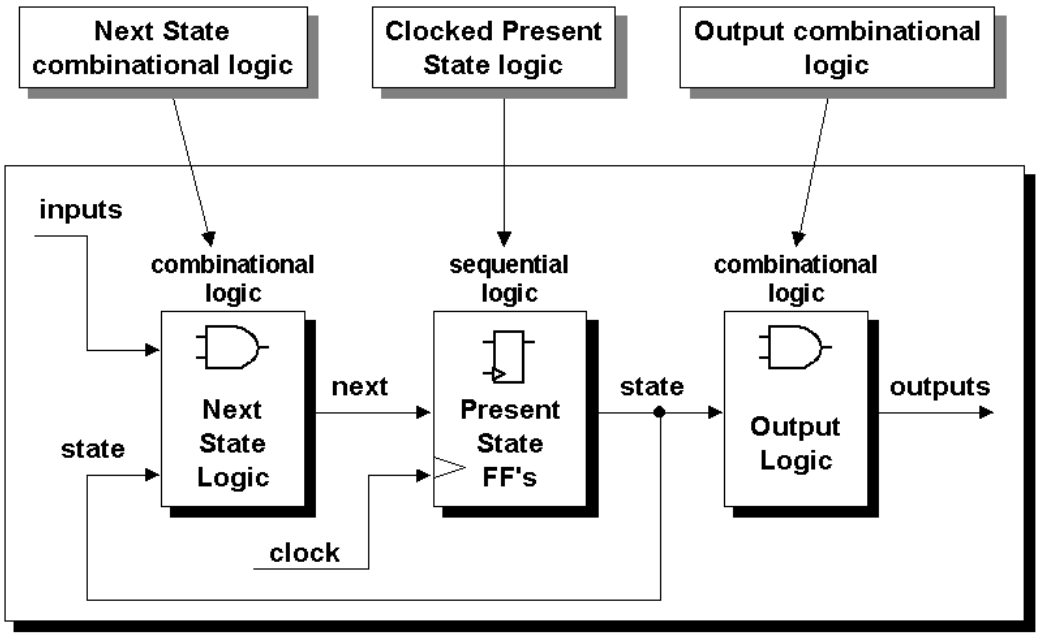


Figure 4 - Moore FSM Block Diagram - two always block coding style

The two always block coding style, as shown in Example 1, uses one sequential always block (with assignments coded using nonblocking assignments) and one combinational always block (with assignments coded using blocking assignments). The sequential always block is typically just three lines of code.

The two always block style is an efficient coding style because output assignments are only required to be listed once (at most) for each state in the **`case`** statement.

```
module fsm_cc1_2
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  parameter IDLE = 2'b00,
            READ = 2'b01,
            DLY  = 2'b11,
            DONE = 2'b10;

  reg [1:0] state, next;

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

  always @(state or go or ws) begin
    next = 'bx;
    rd   = 1'b0;
    ds   = 1'b0;
    case (state)
      IDLE : if (go)      next = READ;
             else         next = IDLE;
      READ : begin
               rd = 1'b1;
                          next = DLY;
```

```
            end
        DLY  : begin
                 rd = 1'b1;
                 if (!ws)   next = DONE;
                 else       next = READ;
               end
        DONE : begin
                 ds = 1'b1;
                            next = IDLE;
               end
      endcase
    end
  endmodule
```

Example 1 - fsm1 - two always block coding style (Recommended)

A summary of some of the important coding styles shown in the Example 1 code include:

- 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.
- The combinational always block has a default **`next`** state assignment at the top of the **`always`** block.
- Default output assignments are made prior to 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, place all of the **`next`** assignments in a single column, as opposed to placing inline **`next`** assignments that follow the contour of the RTL code.

A common trick that is used in all FSM coding styles is to make default X-assignments to the **`next`** variable at the top of the **`always`** block, just under the sensitivity list.

2.1.1 X's help to debug the design during simulation

Simulation Trick: X-assignments help highlight bugs in the simulation because if you forget to make a **`next`**-assignment somewhere in the combinational **`always`** block, the **`next`** state variable will go to all X's in the waveform display at the point where the missing assignment should have occurred.

2.1.2 X's help to optimize the design during synthesis

Synthesis Tricks: X-assignments are treated as "don't cares" by synthesis tools so the X-assignment also informs the synthesis tool that for any undefined state-encoding bit-pattern, the **`next`** state is a don't care. Some synthesis tools also benefit from adding a **`case`**-**`default`** X-assignment to the **`next`** variable and all outputs to help identify "don't cares" to the synthesis tool.

2.2 One sequential always block style with registered outputs - (Avoid this style!)

The FSM block diagram for a one always block coding style is shown in Figure 5.

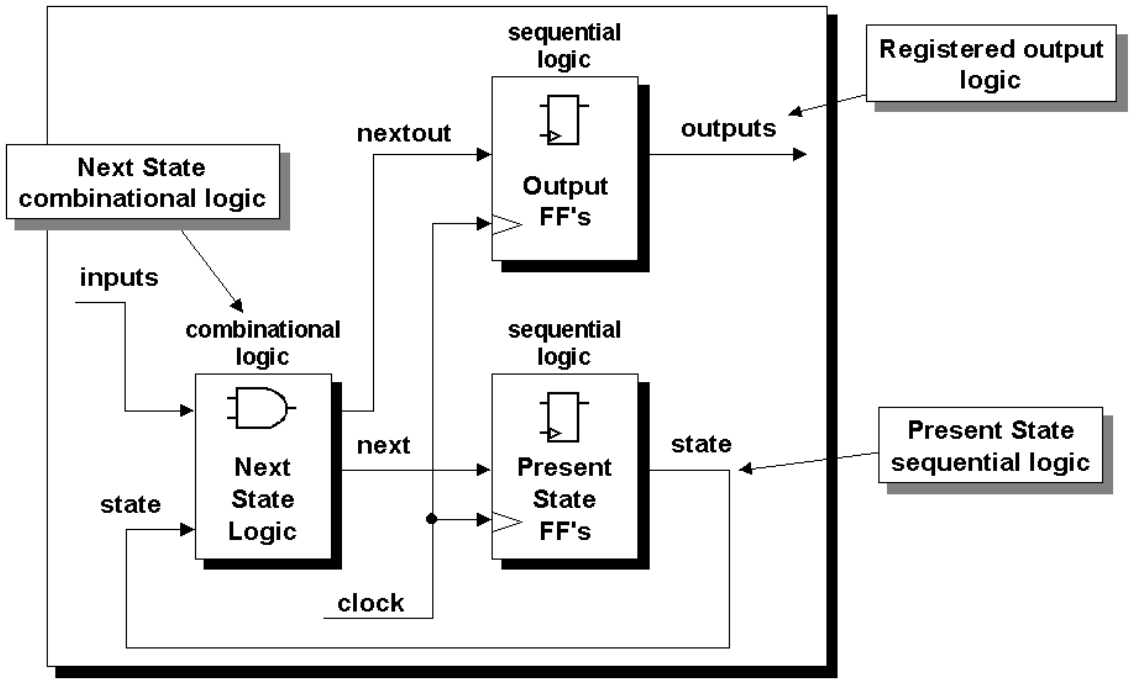


Figure 5 - Moore FSM Block Diagram - one always block coding style

The one always block coding style, as shown in Example 2, uses one large sequential always block (with assignments coded using nonblocking assignments).

The one always block style is a verbose and somewhat error prone coding style because output assignments must be made for every transition arc, as opposed to making output assignments just once for each state as was the case with the two always block coding style.

When making output assignments, you need to think "when I transition to this next state, the output values will be ..."

For small FSM designs, this coding style is not too verbose, but it will be shown that for larger FSM designs, this coding style may require 88% - 165% more code than equivalent three always block coding styles.

```
module fsm_cc1_1
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  parameter IDLE = 2'b00,
            READ = 2'b01,
            DLY  = 2'b11,
            DONE = 2'b10;

  reg [1:0] state;

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

```
        rd    <= 1'b0;
        ds    <= 1'b0;
        case (state)
          IDLE : if (go) begin
                   rd <= 1'b1;
                                 state <= READ;
                 end
                 else            state <= IDLE;
          READ : begin
                   rd <= 1'b1;
                                 state <= DLY;
                 end
          DLY  : if (!ws) begin
                   ds <= 1'b1;
                                 state <= DONE;
                 end
                 else begin
                   rd <= 1'b1;
                                 state <= READ;
                 end
          DONE :                 state <= IDLE;
        endcase
      end
endmodule
```

Example 2 - fsm1 - one always block coding style (NOT recommended!)

2.3 Three always block style with registered outputs (Good style)

The FSM block diagram for a three always block coding style is shown in Figure 6.

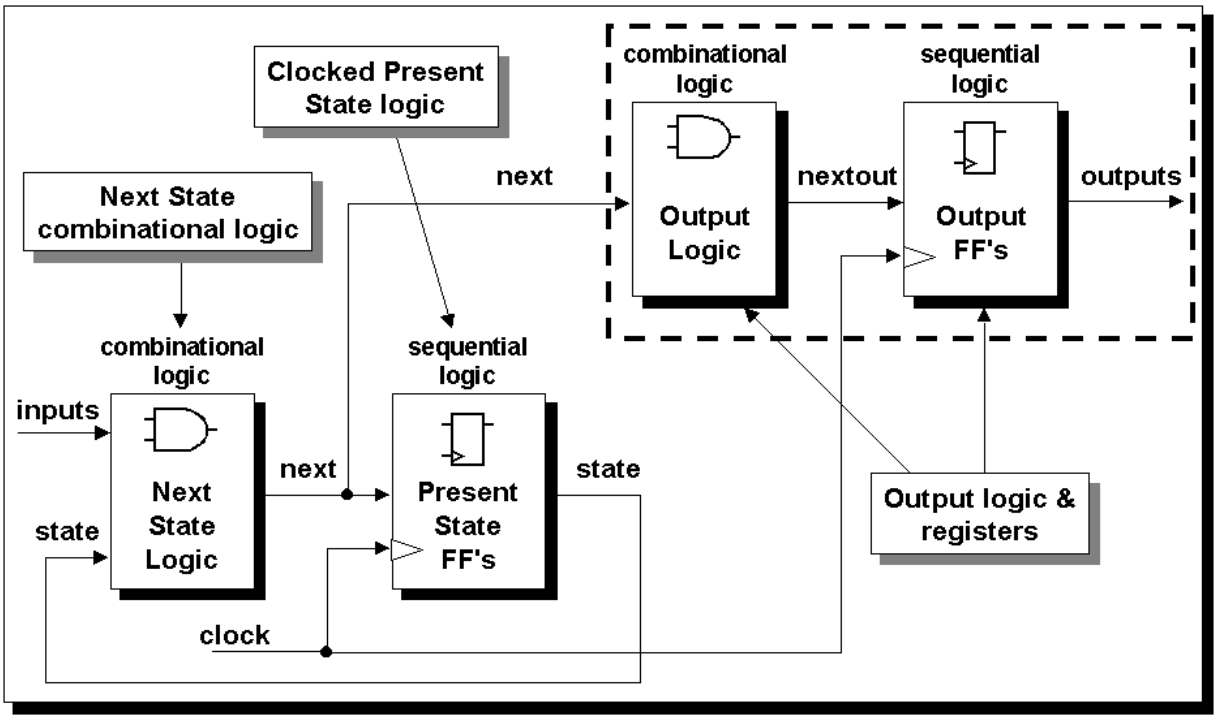


Figure 6 - Moore FSM Block Diagram - three always block coding style

The three always block coding style, as shown in Example 3, uses two sequential always blocks (with assignments coded using nonblocking assignments) and one combinational always block (with assignments coded using blocking assignments). The first sequential always block is typically just three lines of code, the same as the two always block coding style. The second always block is the combinational **`next`** state always block and is very similar to the combinational always block of the two always block coding style, except that the output assignments have been removed. The third always block tests the value of the **`next`** state assignment to determine what the next registered output assignment should be.

The three always block style is an efficient coding style because output assignments are only required once for each state and are placed into a separate sequential always block to register the outputs.

```
module fsm_cc1_3
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  parameter IDLE = 2'b00,
            READ = 2'b01,
            DLY  = 2'b11,
            DONE = 2'b10;

  reg [1:0] state, next;

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

  always @(state or go or ws) begin
    next = 'bx;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE;
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      rd <= 1'b0;
      ds <= 1'b0;
    end
    else begin
      rd <= 1'b0;
      ds <= 1'b0;
      case (next)
        READ : rd <= 1'b1;
        DLY  : rd <= 1'b1;
        DONE : ds <= 1'b1;
      endcase
    end
endmodule
```

Example 3 - fsm1 - three always block coding style (Recommended)

2.4 Onehot coding styles

The index-parameter and encoded-parameter onehot FSM coding styles shown in this paper are variations of the efficient three always block coding style shown in Section 2.3.

2.4.1 Index-parameter style with registered outputs (Good style)

This coding style is unique to Verilog. This coding style uses a reverse **`case`** statement to test to see if a case item is "true" by using a **`case`** header of the form **`case (1'b1)`**. Using this type of **`case`** statement ensures inference of efficient comparison logic that only does 1-bit comparisons against the onehot bits of the **`state`** and **`next`** vectors.

The key to understanding this coding style is to recognize that the **`parameter`** values no longer represent the state encoding of the onehot FSM, instead they represent an index into the **`state`** and **`next`** vectors. This is an index to the onehot bit that is being tested in the design.

```
module fsm_cc1_3oh
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  parameter IDLE = 0,
            READ = 1,
            DLY  = 2,
            DONE = 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 go or ws) begin
    next = 4'b0;
    case (1'b1) // synopsys parallel_case
      state[IDLE] : if (go)  next[READ] = 1'b1;
                    else     next[IDLE] = 1'b1;
      state[READ] :          next[ DLY] = 1'b1;
      state[ DLY] : if (!ws) next[DONE] = 1'b1;
                    else     next[READ] = 1'b1;
      state[DONE] :          next[IDLE] = 1'b1;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      rd <= 1'b0;
      ds <= 1'b0;
    end
    else begin
      rd <= 1'b0;
      ds <= 1'b0;
      case (1'b1) // synopsys parallel_case
        next[READ] : rd <= 1'b1;
```

```
        next[ DLY] : rd <= 1'b1;
        next[DONE] : ds <= 1'b1;
      endcase
    end
endmodule
```

Example 4 - fsm1 - indexed onehot coding style (Recommended)

2.4.2 Encoded-parameter style with registered outputs (Avoid this style!)

This coding style is identical to the three always block coding style shown in Section 2.3 except that the **parameters** now show the encoded onehot patterns.

```
module fsm_cc1_3parm_oh
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  parameter IDLE = 4'b0001,
            READ = 4'b0010,
            DLY  = 4'b0100,
            DONE = 4'b1000;

  reg [3:0] state, next;

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

  always @(state or go or ws) begin
    next = 4'bx;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE;
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      rd <= 1'b0;
      ds <= 1'b0;
    end
    else begin
      rd <= 1'b0;
      ds <= 1'b0;
      case (next)
        READ : rd <= 1'b1;
        DLY  : rd <= 1'b1;
        DONE : ds <= 1'b1;
      endcase
    end
endmodule
```

Example 5 - fsm1 - encoded onehot coding style (NOT recommended!)

Using this coding style generally infers large and slow synthesized designs, because this coding style is forcing full-vector comparisons for both the **`state`** and **`next`** vectors.

This coding style will be shown to be synthesis-inefficient and should be avoided.

2.5 Output encoded style with registered outputs (Good style)

The FSM block diagram for an output encoded coding style is shown in Figure 7.

The output encoded FSM coding style shown in this paper is a variation of the efficient three always block coding style shown in Section 2.3. The techniques used to choose the encodings for this coding style are explained in a separate paper[3].

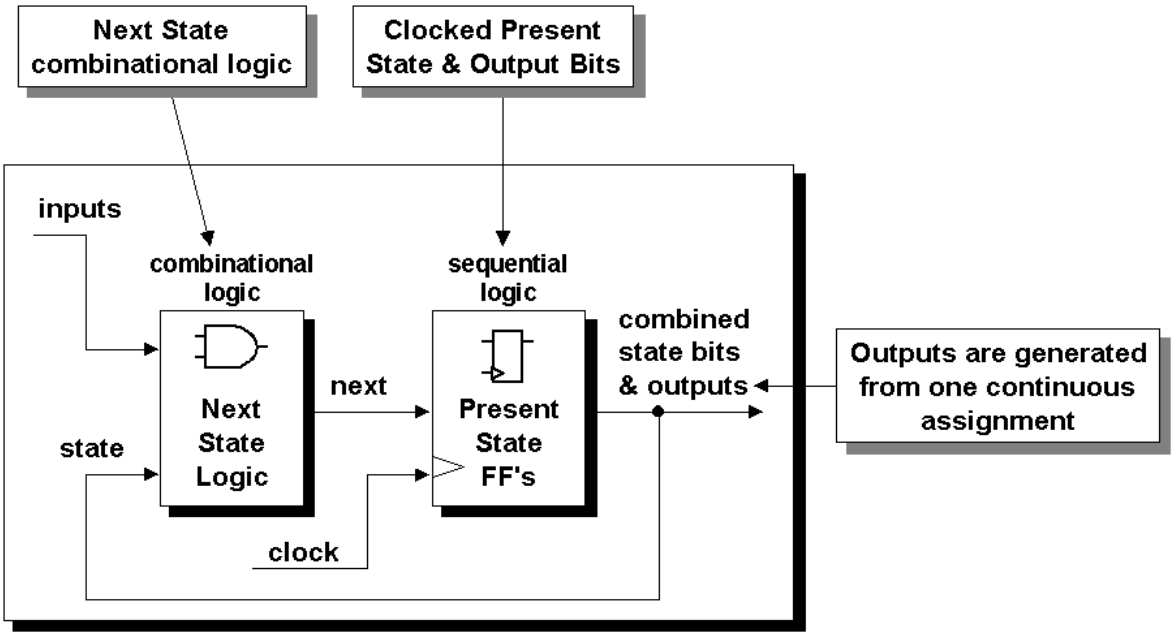


Figure 7 - Moore FSM Block Diagram - output encoded coding style

The output encoded coding style, as shown in Example 6, uses one sequential always block (with assignments coded using nonblocking assignments), one combinational always block (with assignments coded using blocking assignments) and one continuous assignment to assign the significant state bits to the appropriate output signals. The first sequential always block is typically just three lines of code, the same as the three always block coding style. Second always block is the combinational **`next`** state always block and is the same as the combinational always block of the three always block coding style. The continuous assignment assigns state bits to FSM outputs.

```
module fsm_cc1_3oe
  (output rd, ds,
   input  go, ws, clk, rst_n);

  parameter IDLE = 3'b0_00,
            READ = 3'b0_01,
            DLY  = 3'b1_01,
            DONE = 3'b0_10;

  reg [2:0] state, next;

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

  always @(state or go or ws) begin
    next = 'bx;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE;
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
    endcase
  end

  assign {ds,rd} = state[1:0];
endmodule
```

Example 6 - fsm1 - output encoded coding style (Recommended)

Since the registered outputs are also shared as part of the state encoding, this coding style typically uses fewer flip-flops than the equivalent three always block coding style and is therefore usually the most area-efficient design style when synthesized.

3.0 fsm7 Example - 2 inputs, 1 output, 10 states and 20 transition arcs

The **fsm7** design is an example of a 10 state FSM design with an average number of transition arcs and one output. The block diagram for the **fsm7** design is shown in Figure 8.

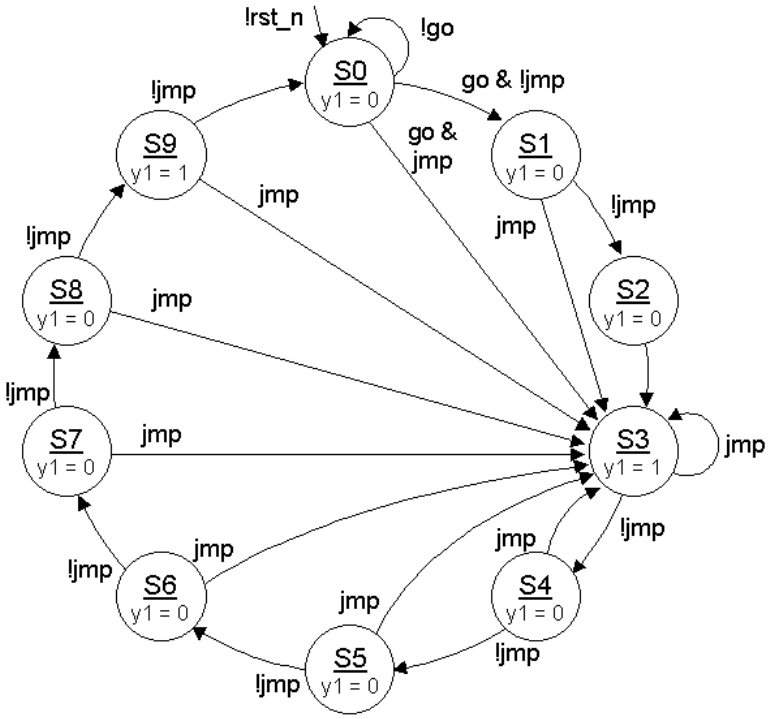


Figure 8 - fsm7 - 10-state FSM design, some transition arcs, 1 output

3.1 fsm7 - one always blocks style (Avoid this style!)

The **fsm_cc7_1** design is the **fsm7** design implemented with one always block. The actual one always block code for the **fsm7** design is listed in Section 15.1.

The one always block version of the **fsm7** design requires 79 lines of code (coding requirements are compared in 6.0).

3.2 fsm7 - three always blocks style (Good style)

The **fsm_cc7_3** design is the **fsm7** design implemented with three always blocks. The actual three always block code for the **fsm7** design is listed in Section 15.2.

The three always block version of the **fsm7** design requires 56 lines of code (coding requirements are compared in 6.0).

4.0 fsm8 Example - 4 inputs, 3 outputs, 10 states and 26 transition arcs

The **fsm8** design is an example of a more complex 10 state FSM design with more transition arcs and three outputs. The block diagram for the **fsm8** design is shown in Figure 9.

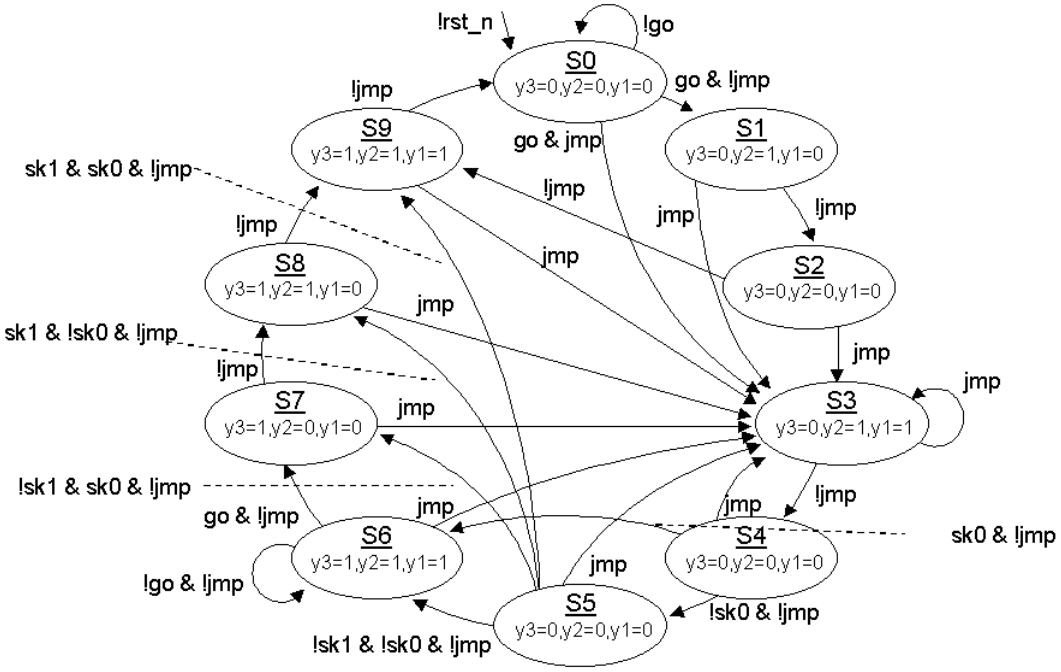


Figure 9 - fsm8 - 10-state FSM design, multiple transition arcs, 3 outputs

4.1 fsm8 - one always blocks style (Avoid this style!)

The **fsm_cc8_1** design is the **fsm8** design implemented with one always block. The actual one always block code for the **fsm8** design is listed in Section 16.1.

The one always block version of the **fsm8** design requires 146 lines of code (coding requirements are compared in 6.0).

4.2 fsm8 - three always blocks style (Good style)

The **fsm_cc8_3** design is the **fsm8** design implemented with three always blocks. The actual three always block code for the **fsm8** design is listed in Section 16.2.

The three always block version of the **fsm8** design requires 82 lines of code (coding requirements are compared in 6.0).

5.0 prep4 Example - 8-bit input, 8-bit output, 16 states and 40 transition arcs

The **prep4** design is an example of a complex 16 state FSM design with multiple transition arcs and an 8-bit output[9]. The block diagram for the **prep4** design is shown in Figure 10.

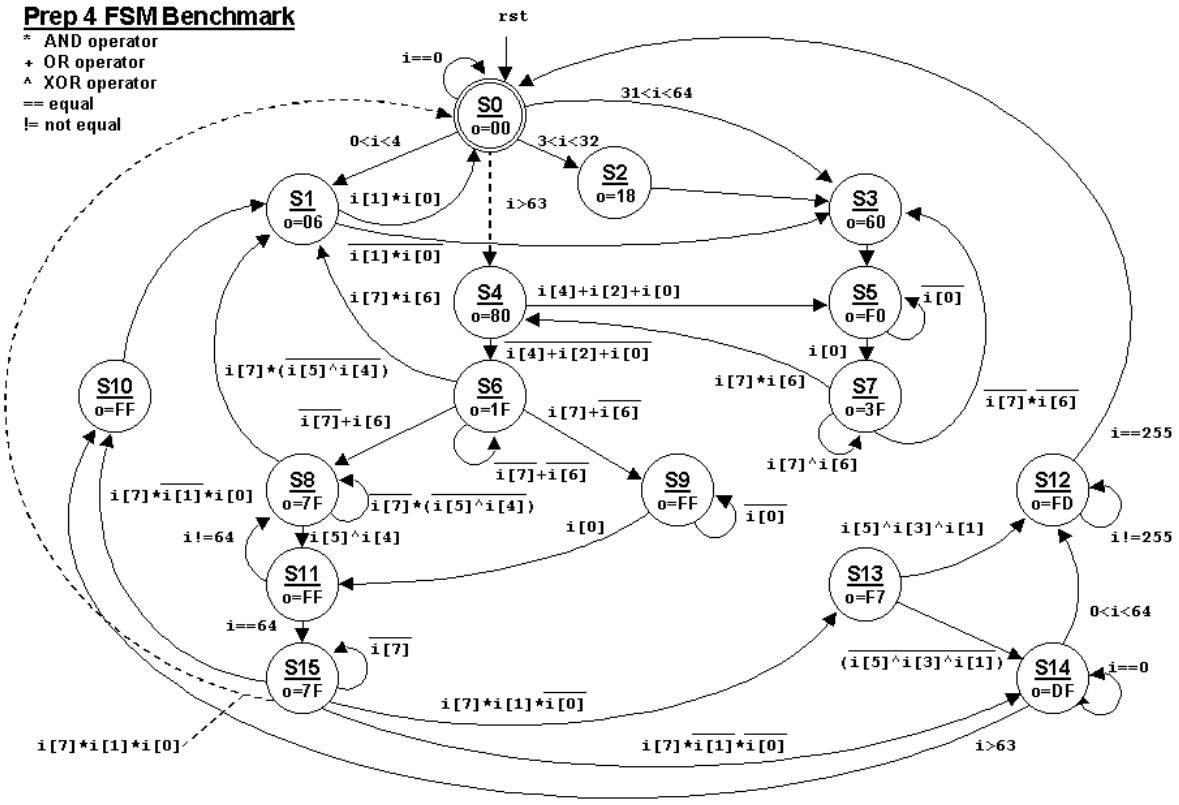


Figure 10 - prep4 - 16-state FSM design, multiple transition arcs, 8-bit output

5.1 prep4 - one always blocks style (Avoid this style!)

The **prep4_1** design is the **prep4** design implemented with one always block. The actual one always block code for the **prep4** design is listed in Section 17.1.

The one always block version of the **prep4** design requires 197 lines of code (coding requirements are compared in 6.0).

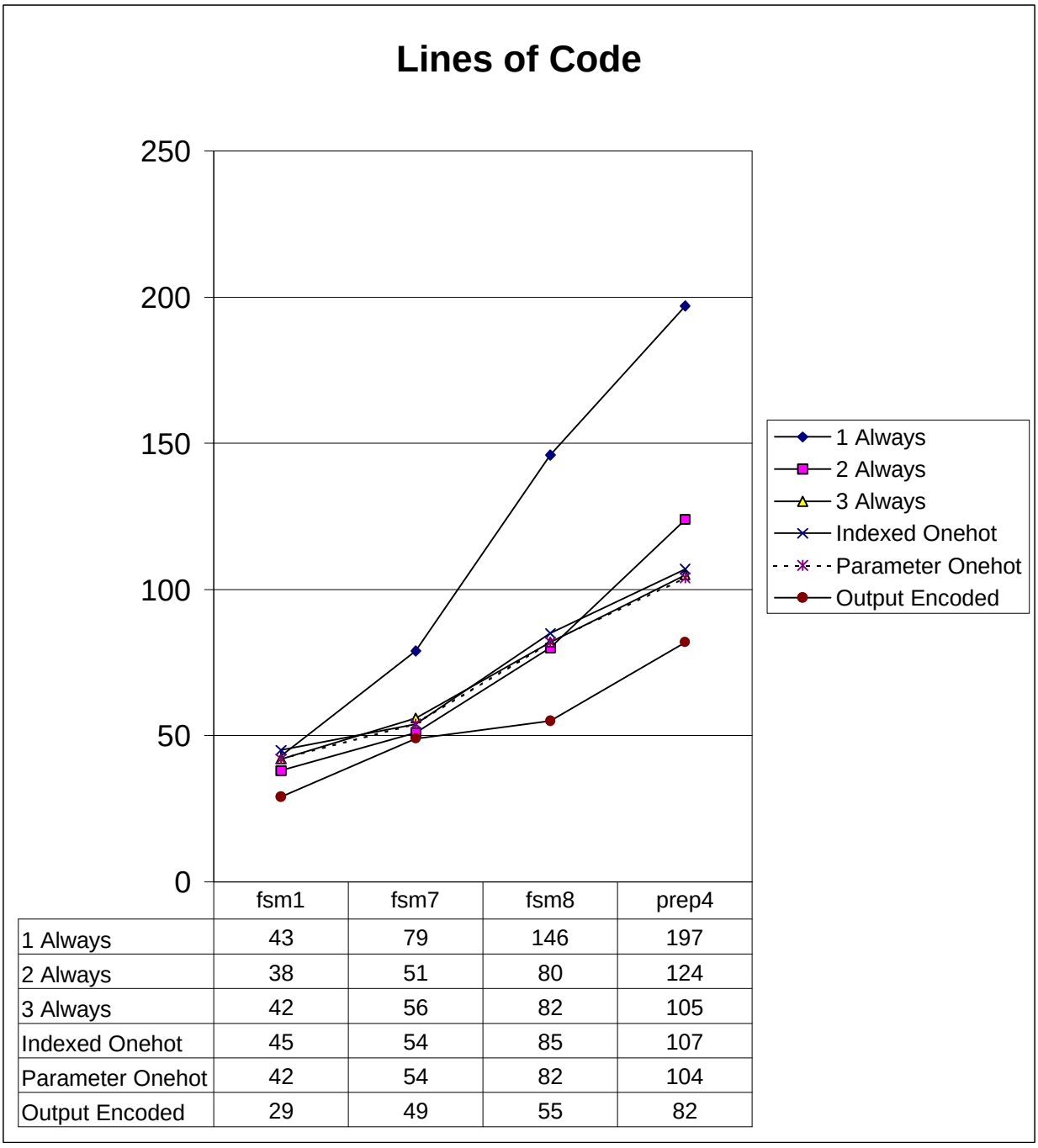
5.2 prep4 - three always blocks style (Good style)

The **prep4_3** design is the **prep4** design implemented with three always blocks. The actual three always block code for the p**r**ep4 design is listed in Section 17.2.

The three always block version of the **prep4** design requires 105 lines of code (coding requirements are compared in 6.0).

6.0 Coding benchmarks for standard FSM coding styles

To evaluate the coding effort required to code the **`fsm1`**, **`fsm7`**, **`fsm8`** and **`prep4`** FSM designs using one always block, two always blocks, three always blocks, three always blocks with indexed onehot style, three always blocks with state-encoded onehot state, and output encoded (two always blocks an one continuous assignment), the number of lines of code were measured and put into graph form as shown in Table 1.



	fsm1	fsm7	fsm8	prep4
1 Always	43	79	146	197
2 Always	38	51	80	124
3 Always	42	56	82	105
Indexed Onehot	45	54	85	107
Parameter Onehot	42	54	82	104
Output Encoded	29	49	55	82

Table 1 - RTL Coding Effort

Note that for small FSM designs, the number of lines of code for all coding styles is almost equal, except for the very efficient output encoded style, but as FSM designs increase in size and complexity, we see from the examples

chosen (**fsm7**, **fsm8** and **prep4**) that the additional effort required to code a one always block FSM design increases by 41%-165% over comparable three always block and output encoded FSM styles.

What does not show up in the graph is that the one always block FSM style is more error prone since output assignments must be made for every transition arc instead of every state, and the output assignments always correspond to output assignments for the next state that we are about to transition to, not the state that we are currently in.

7.0 Synthesis benchmarks for standard FSM coding styles

To evaluate the synthesis efficiency of the **fsm1**, **fsm7**, **fsm8** and **prep4** FSM designs using one always block, two always blocks, three always blocks, three always blocks with indexed onehot style, three always blocks with state-encoded onehot state, and output encoded coding styles, each design was compiled with at least two different timing goals and the resultant timing and area for each compiled design was reported and put into graph form.

The timing goals were arrived at experimentally and chosen as follows:

- **fsm1** design: create_clock clk with periods of 3.0ns and 4.0ns.
- **fsm7** design: create_clock clk with periods of 3.0ns and 4.5ns.
- **fsm8** design: create_clock clk with periods of 3.0ns, 5.5ns and 6.0ns.
- **prep4** design: create_clock clk with periods of 6.0ns and 7.5ns.

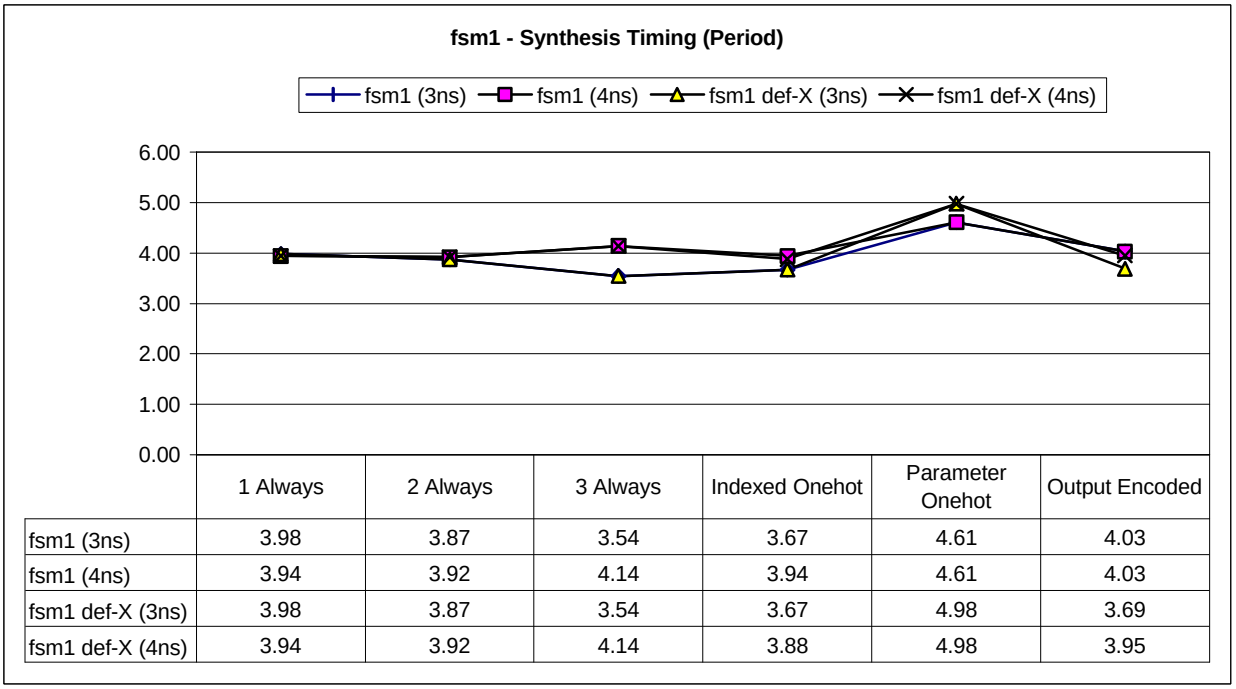
To quickly measure the worst case clock period for each design, a script was setup that would set the desired clock period, compile the design, then set the clock period to 0 and report timing. Setting the clock period to 0 after compiling the design meant that every design would report VIOLATION warnings (very easy to search for this keyword in report files) and the worst case period would be reported as a negative number. All I had to do was search for VIOLATION and change the value from negative to positive to extract the timing values I was interested in.

No effort was made to set input and output timing constraints. All designs were compiled after issuing a single create_clock command. This obviously is not good enough for real designs but it gave a good idea of the synthesizable qualities of the various FSM coding styles chosen for this paper.

It should also be noted that the two always block coding style always showed some of the best results, but those numbers should be viewed with some skepticism. Remember that the combinational outputs for those designs will consume a significant portion of the next clock period, leaving less time for the input-logic that is driven by the FSM outputs. This is one reason we generally try to register outputs from modules when doing RTL coding, to ensure that the down-stream design still has most of the clock period to synthesize efficient input hardware.

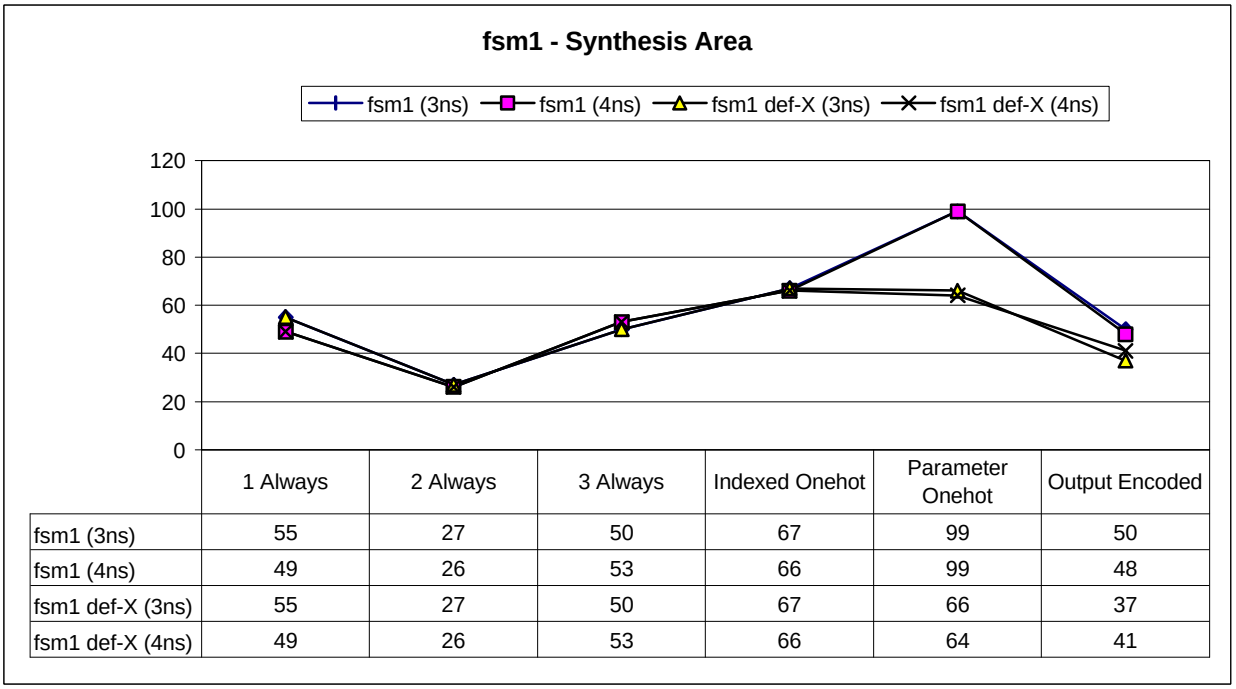
7.1 fsm1 synthesis results

The **fsm1** design was compiled using clock periods of 3ns and 4ns. **case**-**default**-X assignments were also added to the design to see if any timing and area improvements could be achieved.



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
fsm1 (3ns)	3.98	3.87	3.54	3.67	4.61	4.03
fsm1 (4ns)	3.94	3.92	4.14	3.94	4.61	4.03
fsm1 def-X (3ns)	3.98	3.87	3.54	3.67	4.98	3.69
fsm1 def-X (4ns)	3.94	3.92	4.14	3.88	4.98	3.95

Table 2 - fsm1 - Synthesis Timing (Period)



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
fsm1 (3ns)	55	27	50	67	99	50
fsm1 (4ns)	49	26	53	66	99	48
fsm1 def-X (3ns)	55	27	50	67	66	37
fsm1 def-X (4ns)	49	26	53	66	64	41

Table 3 - fsm1 - Synthesis Area

From Table 2 and Table 3 for the **fsm1** design, we can see that the encoded parameter onehot FSM style was inefficient in both timing and area. The superior area efficiencies of the two always block and output encoded styles is not too surprising because the two always block style does not register the outputs and the output encoded style shares output registers with state bits.

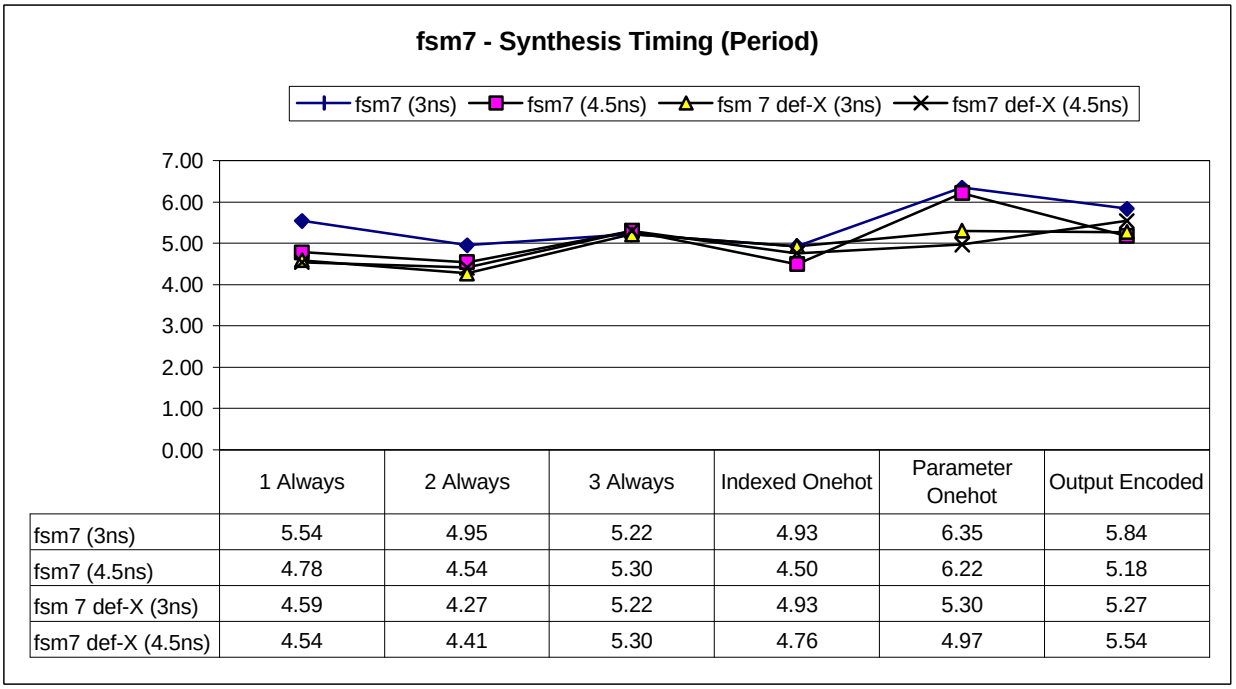
Aside from the parameter encoded onehot style, the timing values are comparable using any of the coding styles, with small improvements noted using a clock period goal of 3ns and **case-default**-X coding added to the three always block, indexed onehot and output encoded styles.

7.2 fsm7 synthesis results

The **fsm7** design was compiled using clock periods of 3ns and 4.5ns. **case-default**-X assignments were also added to the design to see if any timing and area improvements could be achieved.

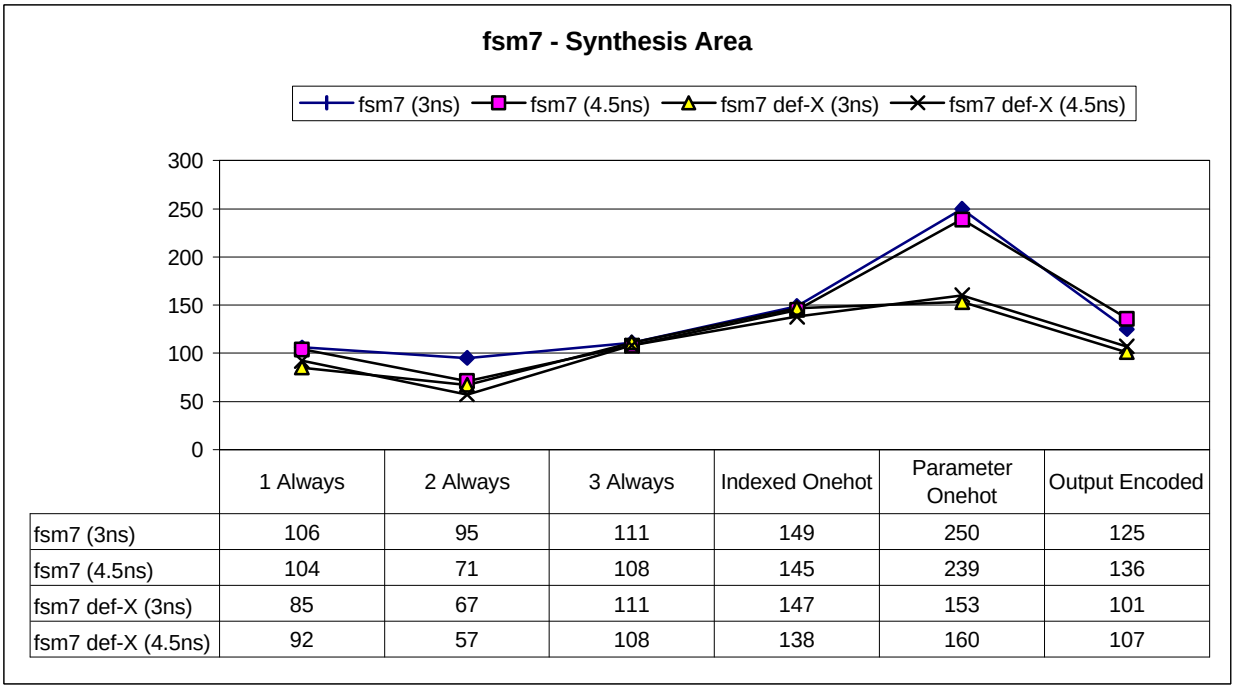
From Table 4 and Table 5 for the **fsm7** design, we can see that the encoded parameter onehot FSM style was inefficient in both timing and area; however, both area and timing did improve when **case-default**-X assignments were added to the RTL code. The two always block style again has slightly better area efficiency but for this design the output encoded style did not demonstrate any area advantage over most coding styles.

For the **fsm7** design, timing was slightly better using the one always block, two always block and indexed onehot coding styles; however, neither changes to the clock period goal or **case-default**-X addition or omission could consistently account for better results.



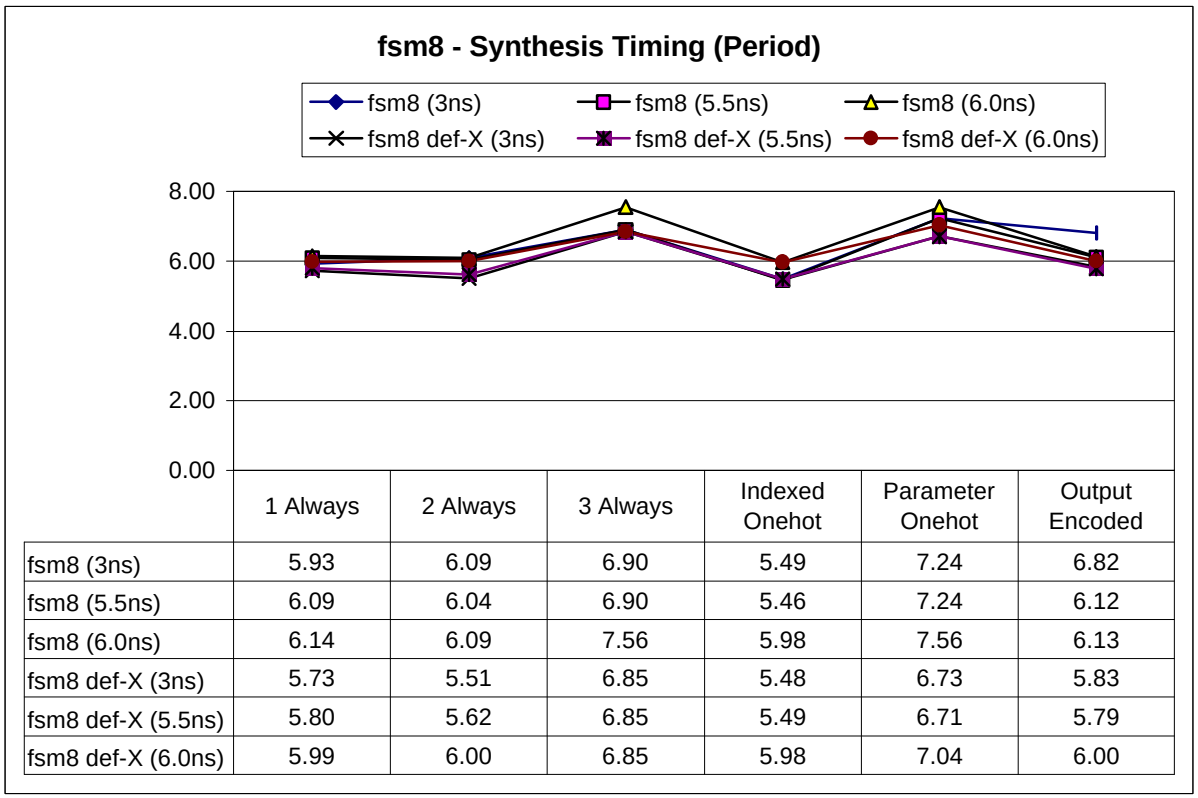
	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
fsm7 (3ns)	5.54	4.95	5.22	4.93	6.35	5.84
fsm7 (4.5ns)	4.78	4.54	5.30	4.50	6.22	5.18
fsm 7 def-X (3ns)	4.59	4.27	5.22	4.93	5.30	5.27
fsm7 def-X (4.5ns)	4.54	4.41	5.30	4.76	4.97	5.54

Table 4 - fsm7 - Synthesis Timing (Period)



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
fsm7 (3ns)	106	95	111	149	250	125
fsm7 (4.5ns)	104	71	108	145	239	136
fsm7 def-X (3ns)	85	67	111	147	153	101
fsm7 def-X (4.5ns)	92	57	108	138	160	107

Table 5 - fsm7 - Synthesis Area



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
fsm8 (3ns)	5.93	6.09	6.90	5.49	7.24	6.82
fsm8 (5.5ns)	6.09	6.04	6.90	5.46	7.24	6.12
fsm8 (6.0ns)	6.14	6.09	7.56	5.98	7.56	6.13
fsm8 def-X (3ns)	5.73	5.51	6.85	5.48	6.73	5.83
fsm8 def-X (5.5ns)	5.80	5.62	6.85	5.49	6.71	5.79
fsm8 def-X (6.0ns)	5.99	6.00	6.85	5.98	7.04	6.00

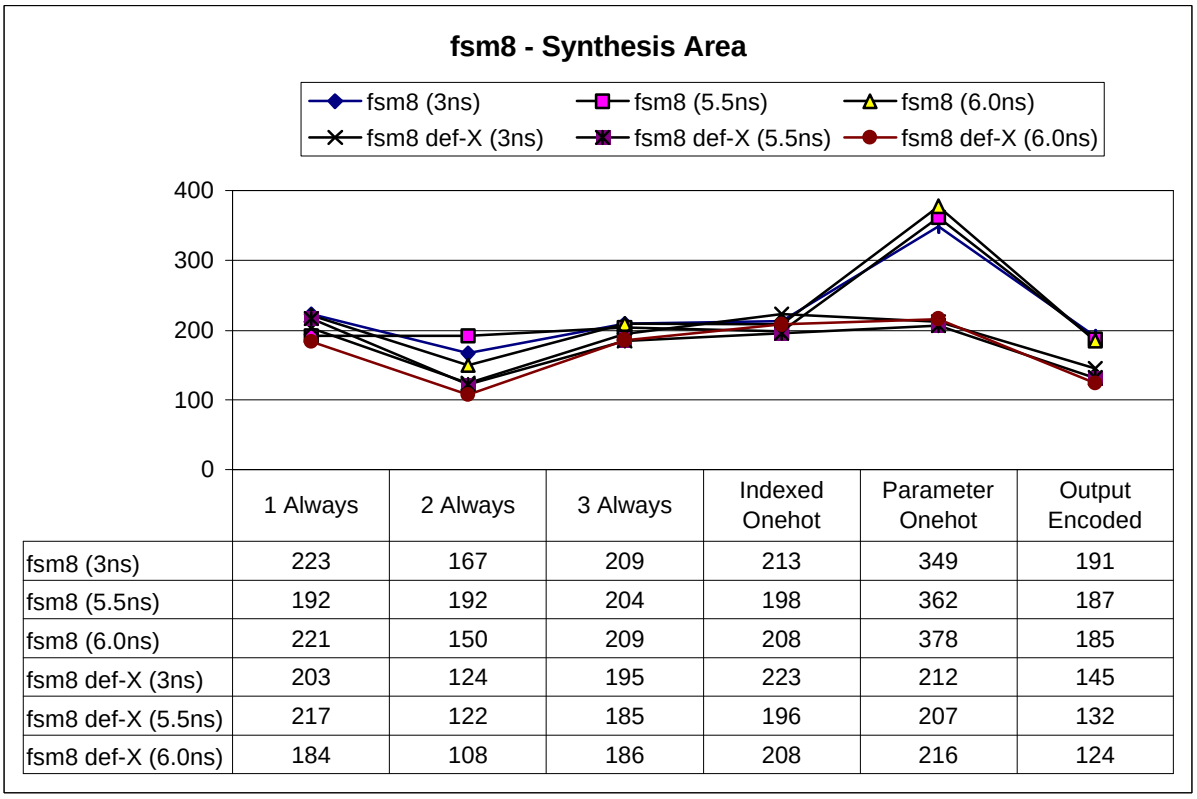
Table 6 - fsm8 - Synthesis Timing (Period)

7.3 fsm8 synthesis results

The **fsm8** design was compiled using clock periods of 3ns, 5.5ns and 6ns. **case-default**-X assignments were also added to the design to see if any timing and area improvements could be achieved.

From Table 6 and Table 7 for the **fsm8** design, we can see that the encoded parameter onehot FSM style was inefficient in both timing and area. When **case-default**-X assignments were added to the design, the timing did not improve but the area did improve. For this design, the three always block style had timing results that were somewhat poor. The two always block and output encoded styles had better area efficiency.

For the **fsm8** design, timing was slightly better using the two always block and indexed onehot coding styles, while adding **case-default**-X assignments helped improve the area efficiency of the encoded onehot style.



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
fsm8 (3ns)	223	167	209	213	349	191
fsm8 (5.5ns)	192	192	204	198	362	187
fsm8 (6.0ns)	221	150	209	208	378	185
fsm8 def-X (3ns)	203	124	195	223	212	145
fsm8 def-X (5.5ns)	217	122	185	196	207	132
fsm8 def-X (6.0ns)	184	108	186	208	216	124

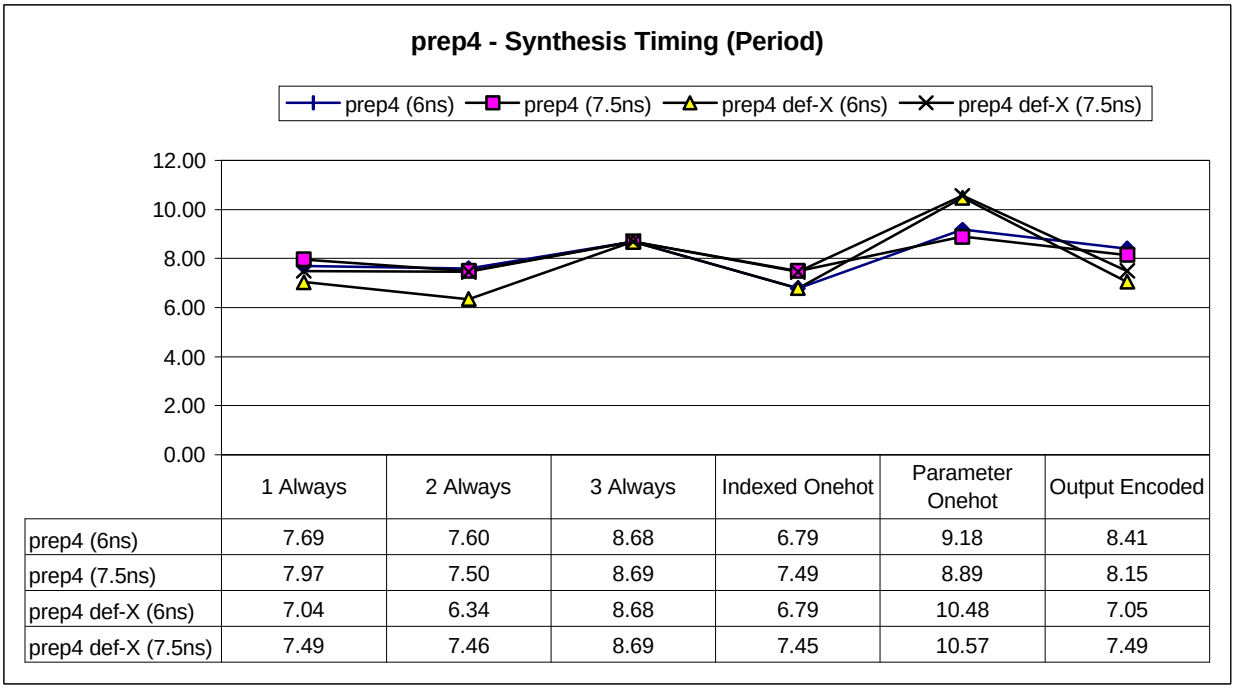
Table 7 - fsm8 - Synthesis Area

7.4 prep4 synthesis results

The **prep4** design was compiled using clock periods of 6ns and 7.5ns. **case-default**-X assignments were also added to the design to see if any timing and area improvements could be achieved.

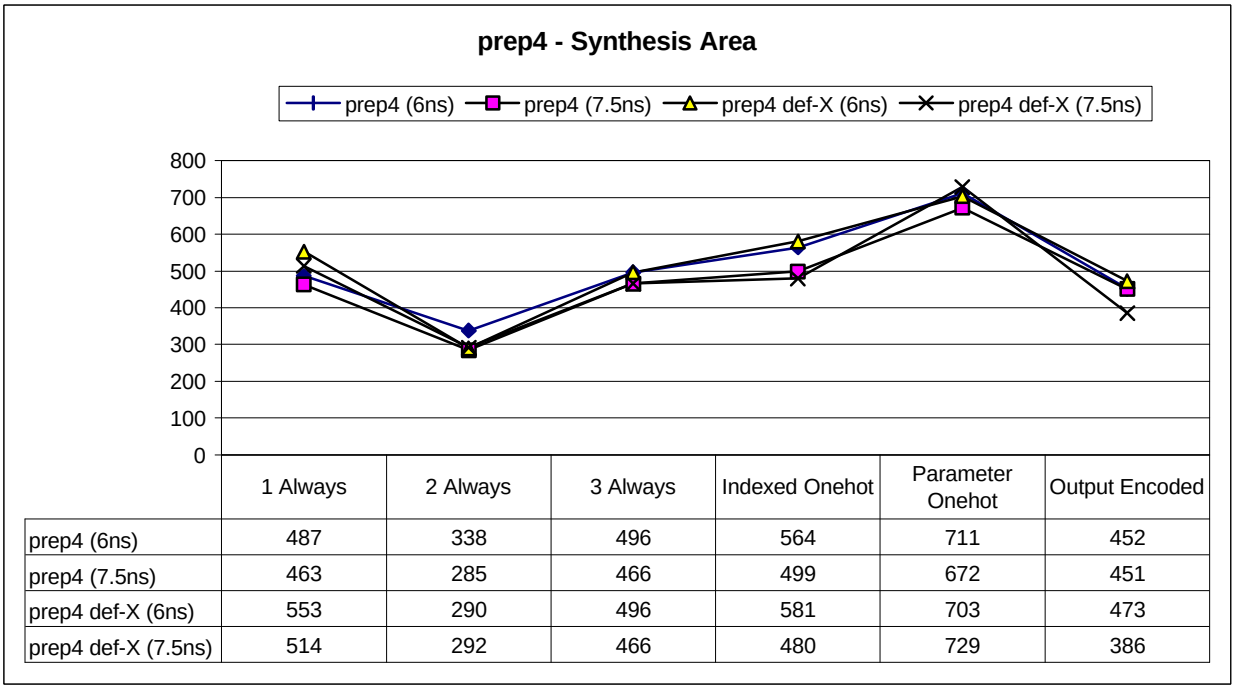
From Table 8 and Table 9 for the **prep4** design, we can see that the encoded parameter onehot FSM style was inefficient in both timing and area and neither area nor timing improved when **case-default**-X assignments were added to the RTL code. The two always block and output encoded styles again had better area efficiency.

For the **prep4** design, timing was slightly better using the two always block and indexed onehot coding styles; however, neither changes to the clock period goal or **case-default**-X addition or omission could consistently account for better results.



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
prep4 (6ns)	7.69	7.60	8.68	6.79	9.18	8.41
prep4 (7.5ns)	7.97	7.50	8.69	7.49	8.89	8.15
prep4 def-X (6ns)	7.04	6.34	8.68	6.79	10.48	7.05
prep4 def-X (7.5ns)	7.49	7.46	8.69	7.45	10.57	7.49

Table 8 - prep4 - Synthesis Timing (Period)



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
prep4 (6ns)	487	338	496	564	711	452
prep4 (7.5ns)	463	285	466	499	672	451
prep4 def-X (6ns)	553	290	496	581	703	473
prep4 def-X (7.5ns)	514	292	466	480	729	386

Table 9 - prep4 - Synthesis Area

8.0 DC-Ultra 2002.05 FSM Tool

Starting with DC-Ultra 2002.05, an enhanced FSM tool capability was added to the synthesis tools[8]. The DC-Ultra version of the FSM tool replaced the old FSM tool that had additional RTL coding requirements and more steps to achieve FSM optimizations.

The new switches added to DC-Ultra are:

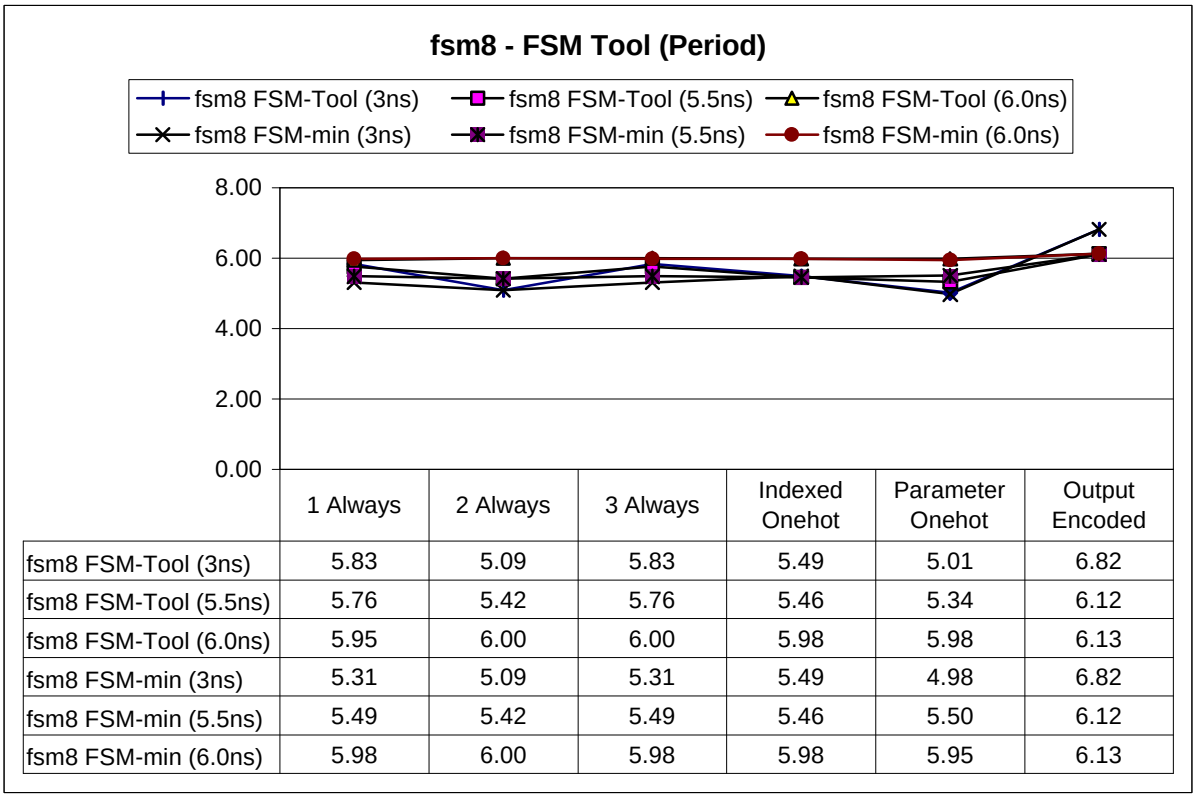
```
fsm_auto_inferring = true
set_ultra_optimization
```

These commands basically tell DC to automatically recognize and optimize FSM designs from the Verilog RTL source code.

There is one more FSM optimization switch that directs the DC FSM tools to look for reductions in the number of states required by the FSM design. This switch is:

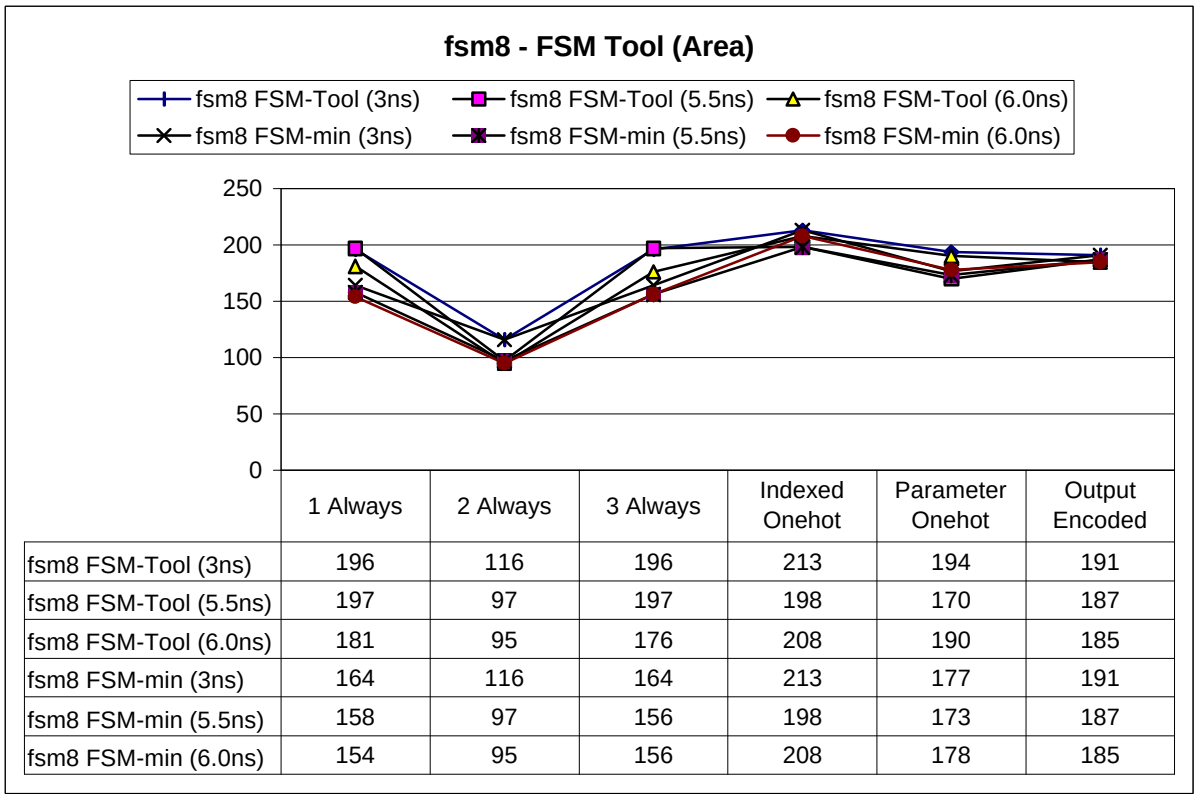
```
fsm_enable_state_minimization = true
```

These commands were used on the **`fsm8`** and **`prep4`** designs with the results shown in Table 10, Table 11, Table 12 and Table 13.



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
fsm8 FSM-Tool (3ns)	5.83	5.09	5.83	5.49	5.01	6.82
fsm8 FSM-Tool (5.5ns)	5.76	5.42	5.76	5.46	5.34	6.12
fsm8 FSM-Tool (6.0ns)	5.95	6.00	6.00	5.98	5.98	6.13
fsm8 FSM-min (3ns)	5.31	5.09	5.31	5.49	4.98	6.82
fsm8 FSM-min (5.5ns)	5.49	5.42	5.49	5.46	5.50	6.12
fsm8 FSM-min (6.0ns)	5.98	6.00	5.98	5.98	5.95	6.13

Table 10 - fsm8 - FSM Tool (Timing)



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
fsm8 FSM-Tool (3ns)	196	116	196	213	194	191
fsm8 FSM-Tool (5.5ns)	197	97	197	198	170	187
fsm8 FSM-Tool (6.0ns)	181	95	176	208	190	185
fsm8 FSM-min (3ns)	164	116	164	213	177	191
fsm8 FSM-min (5.5ns)	158	97	156	198	173	187
fsm8 FSM-min (6.0ns)	154	95	156	208	178	185

Table 11 - fsm8 - FSM Tool (Area)

Comparing the results of the **`fsm8`** design with and without the DC-Ultra FSM tool shows that the FSM tool made small improvements in timing and area for the one always, two always and three always block designs. The indexed onehot and output encoded designs experienced no improvement, probably because they were not recognized as standard FSM coding styles. The parameter onehot design made significant improvements in both timing and area.

It is also interesting that using the DC-Ultra FSM switches appears to have created equivalent designs for the one always and three always block coding styles. This appears to indicate that the FSM tool will build the same FSM designs from a standard FSM coding style whether it is the much more verbose one always block style or the much more concise three always block coding style.

Enabling the **`fsm_enable_state_minimization`** switch on the **`fsm8`** design reported messages that one of the states had been deleted. I ran the gate-level simulation to ensure that the design still functioned the same (and it did) but I have not taken the time to analyze exactly how DC-ultra modified the original design and why it still works. This may be a future research topic worthy of exploration. It should be noted that the Synopsys equivalence checking tool, Formality, does not support state minimization checking[10]. It is not likely that any tool would easily support equivalence checking of a modified and state-reduced FSM design.

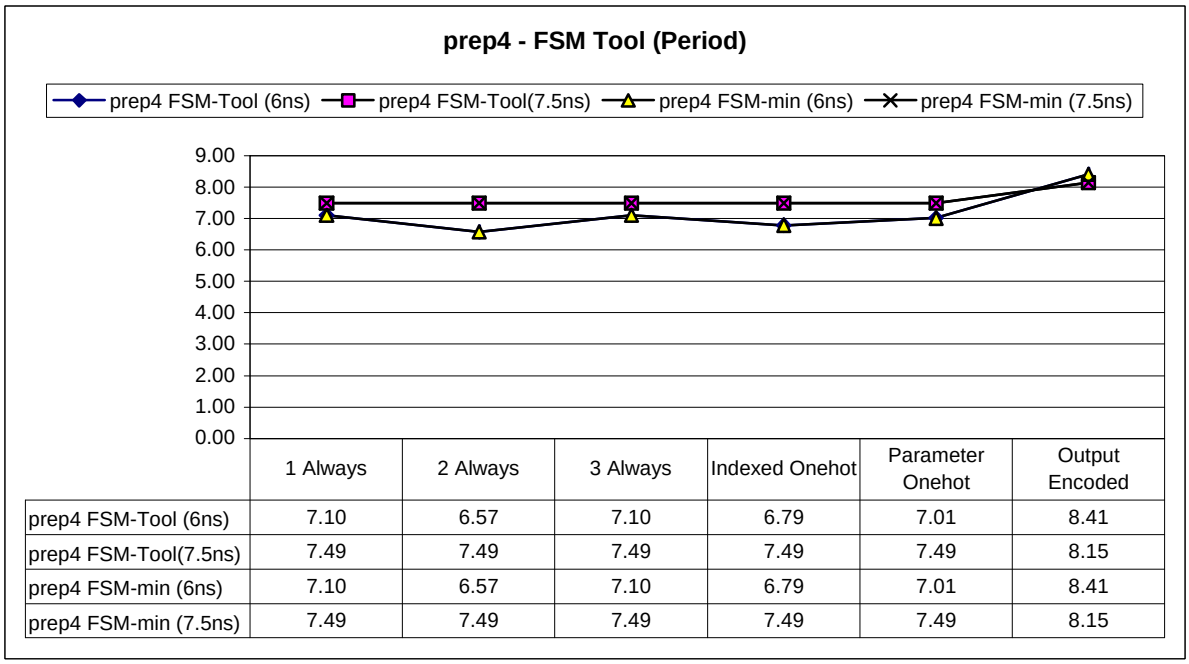
Before writing out the compiled gate-level design, it is a good idea to set the following switch:

```
verilogout_show_unconnected_pins = true
```

This switch writes forces DC to write out the gate level netlist so that all of the unused pins, such as the inverted QN outputs from flip-flops as shown below, are listed in the netlist, which eliminates unnecessary warnings about "unconnected pins" or "different number of ports."

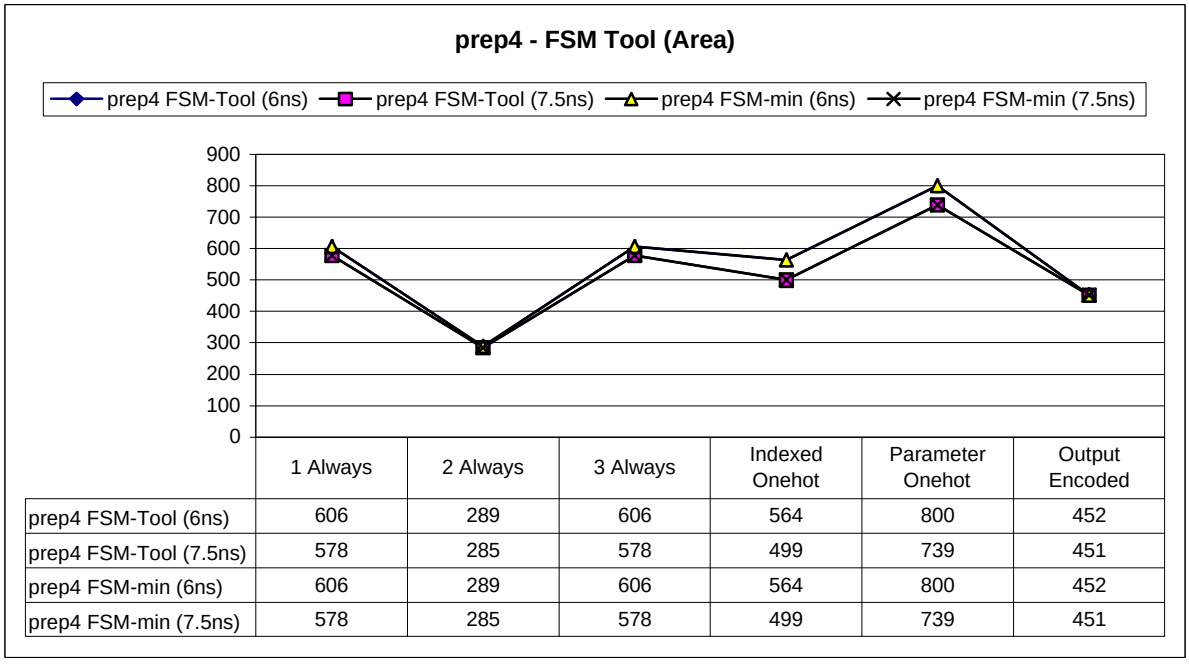
```
FD2 y1_reg ( .D(N153), .CP(clk), .CD(rst_n), .Q(y1), .QN() );
```

The same optimizations were run on the **`prep4`** design, with similar improvements. The only difference was that the **`fsm_enable_state_minimization`** switch did not cause any notable difference. The **`fsm_enable_state_minimization`** switch did not issue any messages about states being deleted, so I assume no such optimization was possible and therefore the optimization with and without this switch being set were the same.



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
prep4 FSM-Tool (6ns)	7.10	6.57	7.10	6.79	7.01	8.41
prep4 FSM-Tool(7.5ns)	7.49	7.49	7.49	7.49	7.49	8.15
prep4 FSM-min (6ns)	7.10	6.57	7.10	6.79	7.01	8.41
prep4 FSM-min (7.5ns)	7.49	7.49	7.49	7.49	7.49	8.15

Table 12 - prep4 - FSM Tool (Timing)



	1 Always	2 Always	3 Always	Indexed Onehot	Parameter Onehot	Output Encoded
prep4 FSM-Tool (6ns)	606	289	606	564	800	452
prep4 FSM-Tool (7.5ns)	578	285	578	499	739	451
prep4 FSM-min (6ns)	606	289	606	564	800	452
prep4 FSM-min (7.5ns)	578	285	578	499	739	451

Table 13 - prep4 - FSM Tool (Area)

9.0 Verliog-2001 Enhanced Coding Styles

Verilog-2001 is an IEEE Standard that includes multiple enhancements to the IEEE Verilog-1995 Standard, to increase RTL and verification capabilities. Many of the enhancements have been implemented by vendors.

9.1 Verilog-2001 combined declarations

Verilog-2001 enhances port declarations to permit combined port direction and data type in the same declaration. This enhancement reduces some of the redundancy associated with module port declarations. Example 7 shows an 8-bit register model where the **q**-port declaration combines both the **output** direction and **reg** data type into the same declaration.

```
module regset1_n (q, d, clk, set_n);
  output reg [7:0] q;
  input      [7:0] d;
  input            clk, set_n);

  always @(posedge clk or negedge set_n)
    if (!set_n) q <= {8{1'b1}};
    else        q <= d;
endmodule
```

Example 7 - Verilog-2001 combined declarations

VCS, SystemSim and Design Compiler all support this enhancement.

9.2 Verilog-2001 ANSI style module ports

Verilog-2001 also enhances port declarations to permit combined port list, port direction and data type in the same declaration. This is often referred to as ANSI-C style module port declarations and reduces some of the redundancy associated with module port declarations. Example 8 shows an 8-bit register model where the **q**-port declaration combines the module port, the **output** direction and the **reg** data type into the same declaration. This example also combines the module port and the **input** direction declarations for all of the module inputs.

This Verilog-2001 enhancement significantly reduces the verbosity and redundancy of module port lists.

```
module regset2_n
  (output reg [7:0] q,
   input      [7:0] d,
   input            clk, set_n);

  always @(posedge clk or negedge set_n)
    if (!set_n) q <= {8{1'b1}};
    else        q <= d;
endmodule
```

Example 8 - Verilog-2001 ANSI style module ports

VCS and SystemSim support this enhancement but Design Compiler does not support this enhancement yet.

In order to test Verilog-2001 enhancements with all of the Synopsys tools, conditionally compiled module headers (as shown in Example 9 for the **fsm1** design) were added to each of the sample FSM designs, where the default was to use non-ANSI-C style ports when the files were read by Design Compiler, but to use ANSI-C style ports whenever the simulation switch **+define+RTL** was added to the command line.

```
`ifdef RTL
module fsm_cc1_1 (
  (output reg rd, ds,
   input      go, ws, clk, rst_n);
`else
```

```
module fsm_cc1_1 (rd, ds, go, ws, clk, rst_n);
  output reg rd, ds;
  input      go, ws, clk, rst_n;
`endif
...
```

Example 9 - Conditionally compiled module headers to accommodate Synopsys DC ANSI port limitations

9.3 Verilog-2001 ANSI style parameters

To create a parameterized model using Verilog-2001 ANSI-C style ports, **parameter** values are often required before the port declaration is made. Verilog-2001 added enhanced ANSI style declarations to allow parameters to be declared and then used in the ANSI-C style port declaration. Example 10 shows an 8-bit register model where a **SIZE** parameter is declared and used in the **q** and **d**-port declarations.

```
module regset3_n #(parameter SIZE=8)
  (output reg [SIZE-1:0] q,
   input      [SIZE-1:0] d,
   input                 clk, set_n);

  always @(posedge clk or negedge set_n)
    if (!set_n) q <= {SIZE{1'b1}};
    else        q <= d;
endmodule
```

Example 10 - Verilog-2001 ANSI style parameters

This Verilog-2001 enhancement supports the capability to code parameterized models with ANSI-C style ports.

9.4 Verilog-2001 implicit internal 1-bit nets

Prior to Verilog-2001, Verilog-1995[6] required that all internal 1-bit nets, driven by a continuous assignment, be explicitly declared. This requirement was inconsistent with the fact that Verilog allowed any other identifier to default to be a 1-bit **wire** if not declared. Verilog-2001 eliminated this annoying inconsistency in the Verilog language. Example 11 shows a model where the low-true **set_n** module **input** signal is inverted to drive an undeclared, high-true **preset** control signal to the 8-bit register.

```
module regset4_n (q, d, clk, set_n);
  parameter SIZE=8;
  output reg [SIZE-1:0] q;
  input      [SIZE-1:0] d;
  input                 clk, set_n);

  assign preset = ~set_n;

  always @(posedge clk or posedge preset)
    if (!preset) q <= {SIZE{1'b1}};
    else         q <= d;
endmodule
```

Example 11 - Verilog-2001 implicit internal 1-bit nets

9.5 Verilog-2001 @* combinational sensitivity list

Verilog-2001 added the much acclaimed **@*** combinational sensitivity list. The primary intent of this enhancement was to create concise, error-free combinational always blocks. The **@*** basically means, "if Synopsys DC wants the combinational signal in the sensitivity list, so do we!"

Example 12 and Example 13 show the Verilog-1995 and Verilog-2001 versions respectively of combinational sensitivity lists for the combinational always block of any of the three always block **fsm1** coding styles.

```
always @(state or go or ws)
begin
  ...
end
```

Example 12 - Verilog-1995 - combinational sensitivity list

```
always @*
begin
  ...
end
```

Example 13 - Verilog-2001 - @* combinational sensitivity list

The **@*** combinational sensitivity list as defined in the IEEE Verilog-2001 Standard can be written with or without parentheses and with or without spaces as shown in Example 14. Unfortunately **(*** is the token that is used to open a Verilog-2001 attribute, so there is some debate about removing support for all but the **always @*** form of this combinational sensitivity list. In-house tools would probably also be easier to write if the in-house tools did not have to parse anything but the most concise **@*** form. For these reasons, I recommend that users restrict their usage of the combinational sensitivity list to the **@*** form.

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

Example 14 - Legal Verilog-2001 @* combinational sensitivity lists

10.0 SystemVerilog enhancements

SystemVerilog 3.0 is an Accellera Standard conceived to make architectural, abstraction, RTL and verification enhancements to IEEE Verilog-2001.

10.1 Enumerated types

Enumerated types allow a designer to implement an FSM design without thinking about the actual state encodings that will be used in the design. SystemVerilog enumerated types then permit useful assignment of values to the enumerated names to apply designer requirements to the FSM design.

10.1.1 Abstract enumerated names

Enumerated types can be declared without assigned values. This is useful early in a design project. Note that the **enum** declaration in Example 15 defines the state names, **IDLE**, **READ**, **DLY** and **DONE**, but does not assign state encodings to these state names. At this point in the design, the encoded state values are left unassigned or abstract.

```
module fsm_sv1a_3
  (output reg rd, ds,
   input      go, ws, clk, rst_n);

  enum {IDLE,
        READ,
        DLY,
        DONE,
        XX  } state, next;

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

  always @* begin
    next = XX;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE;
      READ :          next = DLY;
```

```
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
    endcase
  end

  always @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      rd <= 1'b0;
      ds <= 1'b0;
    end
    else begin
      rd <= 1'b0;
      ds <= 1'b0;
      case (next)
        READ : rd <= 1'b1;
        DLY  : rd <= 1'b1;
        DONE : ds <= 1'b1;
      endcase
    end
endmodule
```

Example 15 - fsm1 example coded with SystemVerilog abstract enumerated types

10.1.2 Enumerated names with assigned state encodings

SystemVerilog enumerated type declarations can be updated to include user-defined values. This helps when trying to define specific state encodings for an FSM design. Using a default **enum** declaration, the user-defined values must be integer (non-4-state) values.

```
module fsm_sv1b_3
  ...
  enum {IDLE = 3'b00,
        READ = 2'b01,
        DLY  = 2'b10,
        DONE = 2'b11,
        XX   = 3'b111} state, next;
  ...
endmodule
```

Example 16 - fsm1 example modifications to use SystemVerilog assigned integer enumerated values

10.1.3 Enumerated names that permit X-assignments

As noted in Section 2.1.1 and Section 2.1.2, X-assignments help both simulation and synthesis. SystemVerilog enumerated types permit non-integer values to be assigned, such as all X's to assist in design. To use non-integer values, the **enum** type must be set to something like the **reg** type with a size, as shown in Example 17.

```
module fsm_sv1b_3
  ...
  enum reg [1:0] {IDLE = 2'b00,
                  READ = 2'b01,
                  DLY  = 2'b10,
                  DONE = 2'b11,
                  XX   = 'x   } state, next;
  ...
endmodule
```

Example 17 - fsm1 example modifications to use SystemVerilog assigned 4-state enumerated values

10.1.4 Waveform display of enumerated names

A good reason to include enumerated types in a design is to permit useful display of the enumerated names and/or values in a waveform display during design debug.

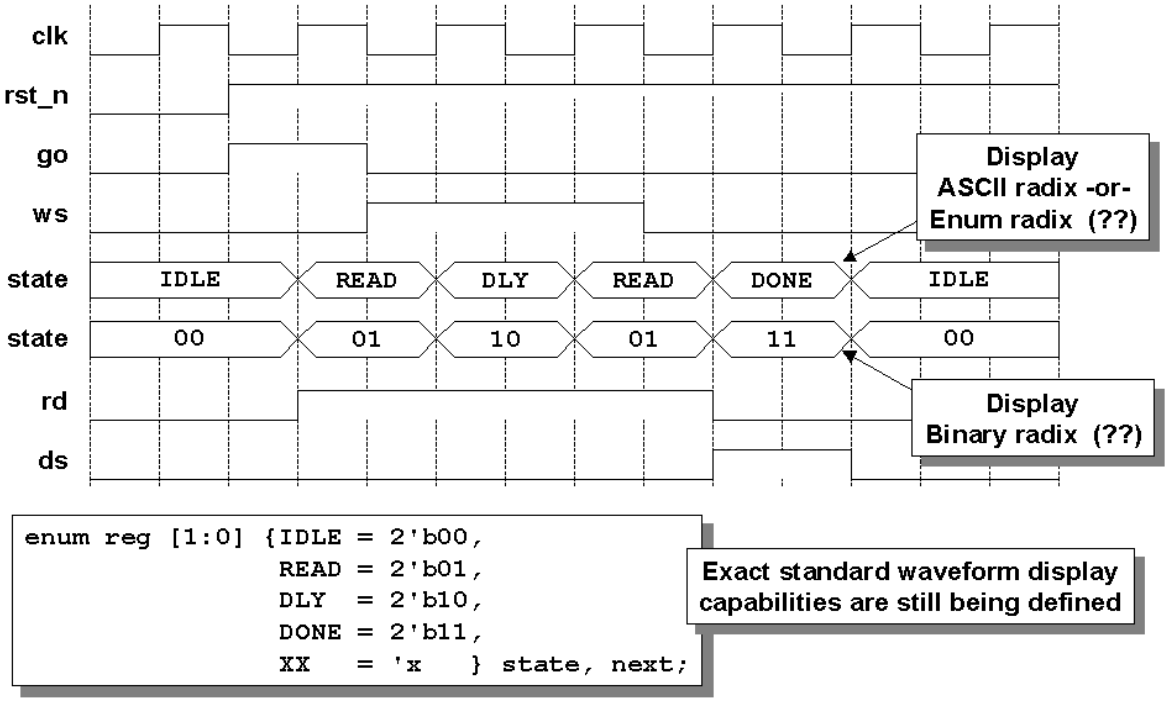


Figure 11 - Waveform display of enumerated types

At the time this paper was written, the exact mechanism for showing enumerated names and values in a waveform display had not yet been fully defined in the SystemVerilog 3.0 or SystemVerilog 3.1 standards. The representation shown in Figure 11 is one potential implementation.

10.2 @* and always_comb

Users can miss warnings from synthesis tools about missing combinational sensitivity list entries. **@*** is both a nice shorthand for combinational sensitivity lists and it will help prevent bugs due to missing sensitivity list entries.

Verilog-2001 added the much anticipated **@*** combinational sensitivity list as described in Section 9.5. SystemVerilog 3.0 added the **always_comb** procedural block which does almost the same thing as **@*** but also descends functions and tasks to extract sensitivity list entries. Example 18 shows partial code from a Verilog-2001 version of the **fsm1** design using the **@*** combinational sensitivity list while Example 19 shows the same partial code from a SystemVerilog version of the **fsm1** design using the **always_comb** combinational sensitivity list.

```
module fsm_sv1b_3
  ...
  always @* begin
    next = 'x;
    case (state)
    ...
  end
  ...
endmodule
```

Example 18 - **@*** combinational sensitivity list

```
module fsm_sv1b_3
  ...
  always_comb begin
    next = 'x;
    case (state)
    ...
  end
  ...
endmodule
```

Example 19 - **always_comb** combinational sensitivity list

Differences that exist between **@*** and **always_comb** include:

- **always_comb** is sensitive to changes within the contents of a function.
- **always_comb** may allow checking for illegal latches.
- **always_comb** triggers once automatically at the end of time **0**.
- **@*** permitted within an **always** block while **always_comb** is not.

Whether or not the functionality of **@*** and **always_comb** should merge or not is still a topic of debate in both the IEEE Verilog and Accellera SystemVerilog committees.

10.3 'x, 'z, '0 and '1 assignments

SystemVerilog has a simple enhancement to make assignments of all 1's, all 0's, all X's and all Z's.

'x is a syntactically more pleasing version of the equivalent **'bx** Verilog-2001 syntax.

'z is a syntactically more pleasing version of the equivalent **'bz** Verilog-2001 syntax.

'1 assigns all 1's and replaces the common Verilog-2001 practices of either making all 1's assignments by either assigning **-1** (2's complement of **-1** is all 1's) or using the replication operator with a **SIZE parameter** to generate all 1's.

'0 is an orthogonal addition that is equivalent to assigning an unsized **0**. Whether or not engineers will ever use the new syntax for all 0's is questionable. The **'0** simply fills out the **'x** , **'z** , **'1** sequence.

In Example 20, the **next = 'bx** assignment has been replaced with **next = 'x**

```
module fsm_sv1b_3
  ...
  always @* begin
    next = 'x;
    case (state)
  ...
endmodule
```

Example 20 - SystemVerilog coding using 'x assignment

11.0 Implicit port connections

SystemVerilog adds two new concise ways to instantiate modules when port names and sizes match the signals that are attached to those ports. These enhancements are referred to as "implicit **.name** port connections" and "implicit **.*** port connections."

These enhancements will be very valuable when instantiating multiple sub-modules into a top-level module for large ASIC and FPGA designs, but they are also convenient when instantiating modules into block-level testbenches.

11.1 Implicit .name port connection capability

The testbench shown in Example 21 instantiates six different versions for the **fsm1** design using six different coding styles. The outputs are compared to ensure that the models are functionally equivalent, and all of the instantiated module inputs (**go**, **ws**, **clk** and **rst_n**) match the stimulus variables that are assigned in the testbench. Named port connections are required for the FSM outputs (because the signal name sizes do not match) but all of the inputs can use the abbreviated **.name** syntax because the signal names and sizes match the port names and sizes.

```
module tb_name1;
  wire [6:1] rd, ds;
  reg        go, ws, clk, rst_n;
  ...
```

```
  fsm_cc1_1         u1  (.rd(rd[ 1]), .ds(ds[ 1]), .go, .ws, .clk, .rst_n);
  fsm_cc1_2         u2  (.rd(rd[ 2]), .ds(ds[ 2]), .go, .ws, .clk, .rst_n);
  fsm_cc1_3         u3  (.rd(rd[ 3]), .ds(ds[ 3]), .go, .ws, .clk, .rst_n);
  fsm_cc1_3oh       u4  (.rd(rd[ 4]), .ds(ds[ 4]), .go, .ws, .clk, .rst_n);
  fsm_cc1_3parm_oh  u5  (.rd(rd[ 5]), .ds(ds[ 5]), .go, .ws, .clk, .rst_n);
  fsm_cc1_3oe       u6  (.rd(rd[ 6]), .ds(ds[ 6]), .go, .ws, .clk, .rst_n);

  initial begin // Stimulus
  ...

  initial begin // Verification
  ...
endmodule
```

Example 21 - Testbench with instantiated fsm designs using **.name** implicit ports

The **.name** implicit instantiation syntax saves a lot of redundant typing.

11.2 Implicit .* port connection capability

Very efficient way to instantiate modules in a top-level design

The testbench shown in Example 22 instantiates six different versions for the **fsm1** design using six different coding styles. The outputs are compared to ensure that the models are functionally equivalent, and all of the instantiated module inputs (**go**, **ws**, **clk** and **rst_n**) match the stimulus variables that are assigned in the testbench. Named port connections are required for the FSM outputs (because the signal name sizes do not match) but all of the inputs can be connected using the abbreviated **.*** syntax because the signal names and sizes match the port names and sizes.

```
module tb_star;
  wire [6:1] rd, ds;
  reg        go, ws, clk, rst_n;
  ...

  fsm_cc1_1         u1  (.rd(rd[ 1]), .ds(ds[ 1]), .*);
  fsm_cc1_2         u2  (.rd(rd[ 2]), .ds(ds[ 2]), .*);
  fsm_cc1_3         u3  (.rd(rd[ 3]), .ds(ds[ 3]), .*);
  fsm_cc1_3oh       u4  (.rd(rd[ 4]), .ds(ds[ 4]), .*);
  fsm_cc1_3parm_oh  u5  (.rd(rd[ 5]), .ds(ds[ 5]), .*);
  fsm_cc1_3oe       u6  (.rd(rd[ 6]), .ds(ds[ 6]), .*);

  initial begin // Stimulus
  ...

  initial begin // Verification
  ...
endmodule
```

Example 22 - Testbench with instantiated fsm designs using **.*** implicit ports

The **.*** implicit instantiation syntax is even more concise than the **.name** syntax, which makes rapid-generation of block-level testbenches easy to do.

The rules and restrictions for using **.name** and **.*** are very logical:

1. mixing **.*** and **.name** ports in the same instantiation is prohibited.
2. **.name** and **.name(signal)** connections in the same instantiation are permitted.
3. **.*** and **.name(signal)** connections in the same instantiation are permitted.
4. **.name(signal)** connections are required for size-mismatch, name-mismatch or unconnected ports.

12.0 FSM coding with SystemVerilog 3.0

It should be noted that SystemVerilog 3.0 coding enhancements do not significantly reduce FSM *coding* efforts but thy do offer very nice syntax and debugging advantages, including:

- Abstract-to-defined enumerated state names
- Enhanced debugging through enumerated-name waveform display
- **`@*`** or **`always_comb`** combinational sensitivity lists
- Nice syntax for specifying, **`'x`** = all X's, **`'z`** = all Z's and **`'1`** = all 1's
- **`.name`** and **`.*`** implicit port connections offer a very nice abbreviated and concise syntax for coding the instantiation of top-level modules in large ASIC and FPGA designs, as well as an easy way to generate a block-level testbench. The **`.name`** and **`.*`** implicit port connection enhancements *do* reduce coding efforts.

Overall, SystemVerilog enhancements will reduce coding efforts and provide powerful and improved architectural, verification and RTL coding capabilities.

13.0 Ask your vendor to support SystemVerilog, NOW!

Users unite! Vendors generally wait to hear users request features before they will implement the enhancements into their tools. If these enhancements look like something you would like to use in real designs, we (the IEEE Verilog Standards Group and the Accellera SystemVerilog Group) need you to ask *all vendors* to support these features. If you ask, they will build it!

14.0 Conclusions

Of the Verilog-2001 FSM coding styles and the FSM designs shown in this paper, the one always block coding style required up to 88% more lines of Verilog code than equivalent three always block coding styles and up to 165% more lines of code than equivalent output encoded FSM coding styles. For only the smallest FSM designs is the one always block coding style somewhat comparable in size to the equivalent three always block coding styles.

Guideline: do not use the one always block FSM coding style.

Coding a onehot FSM design using parameters with state encodings is generally very inefficient when synthesized. Adding case-default X-assignments to the **`case`** statements in the combinational and output always blocks or using the DC Ultra FSM Compiler may help improve the results, but in general follow this guideline:

Guideline: do not use state-encoded parameters to code a onehot FSM design. Use the index-encoded parameter style to implement an efficient onehot FSM design.

The Verilog-2001 **`@*`** combinational sensitivity list is a recommended enhancement to concisely indicate the combinational always block in a three always block coding style, but more importantly, it will help reduce pre-synthesis to post-synthesis simulation mismatch errors that can occur when input signals are missing from the combinational sensitivity list[5].

SystemVerilog enhancements are not yet supported by VCS and DC and cannot be used for real designs.

SystemVerilog will not appreciably reduce the amount of code required to code FSM designs, but the quality of synthesis and debugging will improve.

Continue to encourage all vendors to support enhancements that will facilitate FSM and system design using SystemVerilog.

The DC-Ultra 2002.05 FSM Compiler enhancements appear to offer significant advantages to most FSM designs in both area usage and timing improvements. Once SystemVerilog enumerated types are fully supported in both

VCS and DC, the FSM Compiler enhancements may prove even more valuable to Verilog designs with abstract enumerated FSM state definitions.

Overall, SystemVerilog adds abstraction enhancements that will facilitate design, enhance productivity and enable advanced Verification capability.

Learn what is in the SystemVerilog Standard and bug you vendors until they implement the enhancements.

Acknowledgements

My sincere thanks to Ahmad Ammar for reviewing and offering recommendations to improve the quality of this paper.

References

[1] 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*, March 2002.

[2] Clifford E. Cummings, "State Machine Coding Styles for Synthesis," *SNUG'98 (Synopsys Users Group San Jose, CA, 1998) Proceedings*, section-TB1 (3rd paper), March 1998. Also available at www.sunburst-design.com/papers

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

[4] Clifford E. Cummings, “Verilog Nonblocking Assignments with Delays, Myths & Mysteries,” *SNUG (Synopsys Users Group Boston, MA 2002) Proceedings*, September 2002. Also available at www.sunburst-design.com/papers

[5] Don Mills and Clifford E. Cummings, “RTL Coding Styles That Yield Simulation and Synthesis Mismatches,” *SNUG (Synopsys Users Group) 1999 Proceedings*, section-TA2 (2nd paper), March 1999.
Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers

[6] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-1995

[7] IEEE Standard Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001,

[8] Pete Jarvis, Anthony Redhead and Bob Wiegand, “Synthesis Highlights in DC 2002.05,” *SNUG 2002 (Synopsys Users Group Conference, Boston, MA, 2002) Tutorial*, September 2002, Section TA1, pp. 82-87.

[9] Steve Golson, "State Machine Design Techniques for Verilog and VHDL," Synopsys Journal of High-Level Design, September 1994, pp. 1-48.

[10] Synopsys SolvNet, Doc ID: 004612, “Compiling Finite State Machines in 2002.05 Tools,” Updated: 02/25/2003.

[11] *SystemVerilog 3.0 Accellera's Extensions to Verilog*, Accellera, 2002, www.ovi.org/SystemVerilog_3.0_LRM.pdf

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 21 years of ASIC, FPGA and system design experience and 11 years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, is the only Verilog and SystemVerilog trainer to co-develop and co-author the IEEE 1364-1995 Verilog Standard, IEEE 1364-2001 Verilog Standard, IEEE 1364.1-2002 Verilog RTL Synthesis Standard and the Accellera SystemVerilog 3.0 Standard.

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

Email 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 March 6th, 2003)

15.0 fsm7 Verilog & SystemVerilog code

15.1 fsm7 - one always blocks style (Avoid this style!)

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

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0011,
            S3  = 4'b0010,
            S4  = 4'b0110,
            S5  = 4'b0111,
            S6  = 4'b0101,
            S7  = 4'b0100,
            S8  = 4'b1100,
            S9  = 4'b1000;

  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 23 - fsm7 - one always block coding style (NOT recommended!)

15.2 fsm7 - three always blocks style (Good style)

```
module fsm_cc7_3
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0011,
            S3  = 4'b0010,
            S4  = 4'b0110,
            S5  = 4'b0111,
            S6  = 4'b0101,
            S7  = 4'b0100,
            S8  = 4'b1100,
            S9  = 4'b1000;

  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;
```

```
    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 : if (jmp)      next = S3;
           else          next = S4;
      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 (next)
        S3 : y1 <= 1'b1;
      endcase
    end
endmodule
```

Example 24 - fsm7 - three always block coding style (Recommended)

15.3 fsm7 - three always blocks SystemVerilog style (Good style)

```
module fsm_sv7b_3
  (output reg y1,
   input  jmp, go, clk, rst_n);

  enum reg [3:0] {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,
                  XX  = `x     } state, next;

  always @(posedge clk, negedge rst_n)
    if (!rst_n) state <= S0;
```

```
    else        state <= next;

  always @* begin
    next = XX;
    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 : if (jmp)      next = S3;
           else          next = S4;
      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, negedge rst_n)
    if (!rst_n) y1 <= 1'b0;
    else begin
      y1 <= 1'b0;
      case (next)
        S3 : y1 <= 1'b1;
      endcase
    end
endmodule
```

Example 25 - fsm7 - three always block SystemVerilog coding style (Recommended)

16.0 fsm8 Verilog & SystemVerilog code

16.1 fsm8 - one always blocks style (Avoid this style!)

```
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 <= '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 26 - fsm8 - one always block coding style (NOT recommended!)

16.2 fsm8 - three always blocks style (Good style)

```
module fsm_cc8_3
  (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 = '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)
      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 27 - fsm8 - three always block coding style (Recommended)

16.3 fsm8 - three always blocks SystemVerilog style (Good style)

```
module fsm_sv8b_3
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  enum reg [3:0] {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,
                  XX  = `x     } state, next;

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

  always @* begin
    next = XX;
    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, 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)
        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 28 - fsm8 - three always block SystemVerilog coding style (Recommended)

17.0 prep4 Verilog & SystemVerilog code

17.1 prep4 - one always blocks style (Avoid this style!)

```
module prep4_1
  (output reg [7:0] out,
   input      [7:0] in,
   input            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,
            S10 = 4'b1010,
            S11 = 4'b1011,
            S12 = 4'b1100,
            S13 = 4'b1101,
            S14 = 4'b1110,
            S15 = 4'b1111;

  reg [3:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                                                  state <= S0;
      out <= 8'h00;
    end
    else begin
      state <= 'bx;
      case (state)
        S0 : if (in == 0) begin
               out <= 8'h00;
                                                  state <= S0;
             end
             else if (in <  4) begin
               out <= 8'h06;
                                                  state <= S1;
             end
             else if (in < 32) begin
               out <= 8'h18;
                                                  state <= S2;
             end
             else if (in < 64) begin
               out <= 8'h60;
                                                  state <= S3;
             end
             else begin
               out <= 8'h80;
                                                  state <= S4;
```

```
      end
S1 : if (in[0] & in[1]) begin
       out <= 8'h00;
                                          state <= S0;
      end
      else begin
       out <= 8'h60;
                                          state <= S3;
      end
S2 : begin
       out <= 8'h60;
                                          state <= S3;
      end
S3 : begin
       out <= 8'hF0;
                                          state <= S5;
      end
S4 : if (in[0] | in[2] | in[4]) begin
       out <= 8'hF0;
                                          state <= S5;
      end
      else begin
       out <= 8'h1F;
                                          state <= S6;
      end
S5 : if (in[0]) begin
       out <= 8'h3F;
                                          state <= S7;
      end
      else begin
       out <= 8'hF0;
                                          state <= S5;
      end
S6 : if ( in[6] &  in[7]) begin
       out <= 8'h06;
                                          state <= S1;
      end
      else if (!in[6] &  in[7]) begin
       out <= 8'hFF;
                                          state <= S9;
      end
      else if ( in[6] & !in[7]) begin
       out <= 8'h7F;
                                          state <= S8;
      end
      else begin
       out <= 8'h1F;
                                          state <= S6;
      end
S7 : if ( in[6] &  in[7]) begin
       out <= 8'h80;
                                          state <= S4;
      end
      else if (!in[6] & !in[7]) begin
       out <= 8'h60;
```

```
                                        state <= S3;
      end
      else begin
        out <= 8'h3F;
                                        state <= S7;
      end
S8 : if (in[4] ^ in[5]) begin
        out <= 8'hFF;
                                        state <= S11;
      end
      else if (in[7]) begin
        out <= 8'h06;
                                        state <= S1;
      end
      else begin
        out <= 8'h7F;
                                        state <= S8;
      end
S9 : if (in[0]) begin
        out <= 8'hFF;
                                        state <= S11;
      end
      else begin
        out <= 8'hFF;
                                        state <= S9;
      end
S10: begin
        out <= 8'h06;
                                        state <= S1;
      end
S11: if (in == 64) begin
        out <= 8'h7F;
                                        state <= S15;
      end
      else begin
        out <= 8'h7F;
                                        state <= S8;
      end
S12: if (in == 255) begin
        out <= 8'h00;
                                        state <= S0;
      end
      else begin
        out <= 8'hFD;
                                        state <= S12;
      end
S13: if (in[1] ^ in[3] ^ in[5]) begin
        out <= 8'hFD;
                                        state <= S12;
      end
      else begin
        out <= 8'hDF;
                                        state <= S14;
      end
S14: if (in == 0) begin
```

```
                out <= 8'hDF;
                                                state <= S14;
              end
              else if (in < 64) begin
                out <= 8'hFD;
                                                state <= S12;
              end
              else begin
                out <= 8'hFF;
                                                state <= S10;
              end
          S15: if (!in[7]) begin
                out <= 8'h7F;
                                                state <= S15;
              end
              else case (in[1:0])
                2'b00: begin
                         out <= 8'hDF;
                                                state <= S14;
                       end
                2'b01: begin
                         out <= 8'hFF;
                                                state <= S10;
                       end
                2'b10: begin
                         out <= 8'hF7;
                                                state <= S13;
                       end
                2'b11: begin
                         out <= 8'h00;
                                                state <= S0;
                       end
              endcase
        endcase
      end
endmodule
```

Example 29 - prep4 - one always block coding style (NOT recommended!)

17.2 prep4 - three always blocks style (Good style)

```
module prep4_3
  (output reg [7:0] out,
   input      [7:0] in,
   input            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,
```

```
           S10 = 4'b1010,
           S11 = 4'b1011,
           S12 = 4'b1100,
           S13 = 4'b1101,
           S14 = 4'b1110,
           S15 = 4'b1111;

reg [3:0] state, next;

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

always @(state or in) begin
  next = 'bx;
  case (state)
    S0 : if      (in == 0)                 next = S0;
         else if (in <  4)                 next = S1;
         else if (in < 32)                 next = S2;
         else if (in < 64)                 next = S3;
         else                              next = S4;
    S1 : if      (in[0] && in[1])          next = S0;
         else                              next = S3;
    S2 :                                   next = S3;
    S3 :                                   next = S5;
    S4 : if      (in[0] || in[2] || in[4]) next = S5;
         else                              next = S6;
    S5 : if      (!in[0])                  next = S5;
         else                              next = S7;
    S6 : case (in[7:6])
           2'b00:                          next = S6;
           2'b01:                          next = S8;
           2'b10:                          next = S9;
           2'b11:                          next = S1;
         endcase
    S7 : case (in[7:6])
           2'b00:                          next = S3;
           2'b01, 2'b10:                   next = S7;
           2'b11:                          next = S4;
         endcase
    S8 : if      (in[4] ^ in[5])           next = S11;
         else if (in[7])                   next = S1;
         else                              next = S8;
    S9 : if      (!in[0])                  next = S9;
         else                              next = S11;
    S10:                                   next = S1;
    S11: if      (in == 64)                next = S15;
         else                              next = S8;
    S12: if      (in == 255)               next = S0;
         else                              next = S12;
    S13: if      (in[5] ^ in[3] ^ in[1])   next = S12;
         else                              next = S14;
    S14: if      (in == 0)                 next = S14;
         else if (in < 64)                 next = S12;
         else                              next = S10;
```

```
      S15: if (!in[7])                          next = S15;
           else case (in[1:0])
             2'b00:                             next = S14;
             2'b01:                             next = S10;
             2'b10:                             next = S13;
             2'b11:                             next = S0;
           endcase
      default:                                  next = 'bx;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) out <= 8'h00;
    else begin
      out = 'bx;
      case (next)
        S0:       out <= 8'h00;
        S1:       out <= 8'h06;
        S2:       out <= 8'h18;
        S3:       out <= 8'h60;
        S4:       out <= 8'h80;
        S5:       out <= 8'hF0;
        S6:       out <= 8'h1F;
        S7:       out <= 8'h3F;
        S8:       out <= 8'h7F;
        S9:       out <= 8'hFF;
        S10:      out <= 8'hFF;
        S11:      out <= 8'hFF;
        S12:      out <= 8'hFD;
        S13:      out <= 8'hF7;
        S14:      out <= 8'hDF;
        S15:      out <= 8'h7F;
      endcase
    end
endmodule
```

Example 30 - prep4 - three always block coding style (Recommended)

17.3 prep4 - three always blocks SystemVerilog style (Good style)

```
module prep4b_3
  (output reg [7:0] out,
   input      [7:0] in,
   input            clk, rst_n);

  enum reg [3:0] {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,
                  S10 = 4'b1010,
```

```
                S11 = 4'b1011,
                S12 = 4'b1100,
                S13 = 4'b1101,
                S14 = 4'b1110,
                S15 = 4'b1111,
                XX  = 'x     } state, next;

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

always @* begin
  next = XX;
  case (state)
    S0 : if      (in == 0)                 next = S0;
         else if (in <  4)                 next = S1;
         else if (in < 32)                 next = S2;
         else if (in < 64)                 next = S3;
         else                              next = S4;
    S1 : if      (in[0] && in[1])          next = S0;
         else                              next = S3;
    S2 :                                   next = S3;
    S3 :                                   next = S5;
    S4 : if      (in[0] || in[2] || in[4]) next = S5;
         else                              next = S6;
    S5 : if      (!in[0])                  next = S5;
         else                              next = S7;
    S6 : case (in[7:6])
           2'b00:                          next = S6;
           2'b01:                          next = S8;
           2'b10:                          next = S9;
           2'b11:                          next = S1;
         endcase
    S7 : case (in[7:6])
           2'b00:                          next = S3;
           2'b01, 2'b10:                   next = S7;
           2'b11:                          next = S4;
         endcase
    S8 : if      (in[4] ^ in[5])           next = S11;
         else if (in[7])                   next = S1;
         else                              next = S8;
    S9 : if      (!in[0])                  next = S9;
         else                              next = S11;
    S10:                                   next = S1;
    S11: if      (in == 64)                next = S15;
         else                              next = S8;
    S12: if      (in == 255)               next = S0;
         else                              next = S12;
    S13: if      (in[5] ^ in[3] ^ in[1])   next = S12;
         else                              next = S14;
    S14: if      (in == 0)                 next = S14;
         else if (in < 64)                 next = S12;
         else                              next = S10;
    S15: if (!in[7])                       next = S15;
         else case (in[1:0])
```

```
            2'b00:                          next = S14;
            2'b01:                          next = S10;
            2'b10:                          next = S13;
            2'b11:                          next = S0;
          endcase
    endcase
  end

  always @(posedge clk, negedge rst_n)
    if (!rst_n) out <= 8'h00;
    else begin
      out = 'bx;
      case (next)
        S0:       out <= 8'h00;
        S1:       out <= 8'h06;
        S2:       out <= 8'h18;
        S3:       out <= 8'h60;
        S4:       out <= 8'h80;
        S5:       out <= 8'hF0;
        S6:       out <= 8'h1F;
        S7:       out <= 8'h3F;
        S8:       out <= 8'h7F;
        S9:       out <= 8'hFF;
        S10:      out <= 8'hFF;
        S11:      out <= 8'hFF;
        S12:      out <= 8'hFD;
        S13:      out <= 8'hF7;
        S14:      out <= 8'hDF;
        S15:      out <= 8'h7F;
      endcase
    end
endmodule
```

Example 31 - prep4 - three always block SystemVerilog coding style (Recommended)