State Machine Coding Styles for Synthesis

Clifford E. Cummings

Sunburst Design, Inc.

ABSTRACT

This paper details efficient Verilog coding styles to infer synthesizable state machines. HDL considerations such as advantages and disadvantages of one-always block FSMs Vs. two-always block FSMs are described.

Introduction

Steve Golson's 1994 paper, "State Machine Design Techniques for Verilog and VHDL" [1], is a great paper on state machine design using Verilog, VHDL and Synopsys tools. Steve's paper also offers in-depth background concerning the origin of specific state machine types.

This paper, "State Machine Coding Styles for Synthesis," details additional insights into state machine design including coding style approaches and a few additional tricks.

State Machine Classification

There are two types of state machines as classified by the types of outputs generated from each. The first is the Moore State Machine where the outputs are only a function of the present state, the second is the Mealy State Machine where one or more of the outputs are a function of the present state and one or more of the inputs.

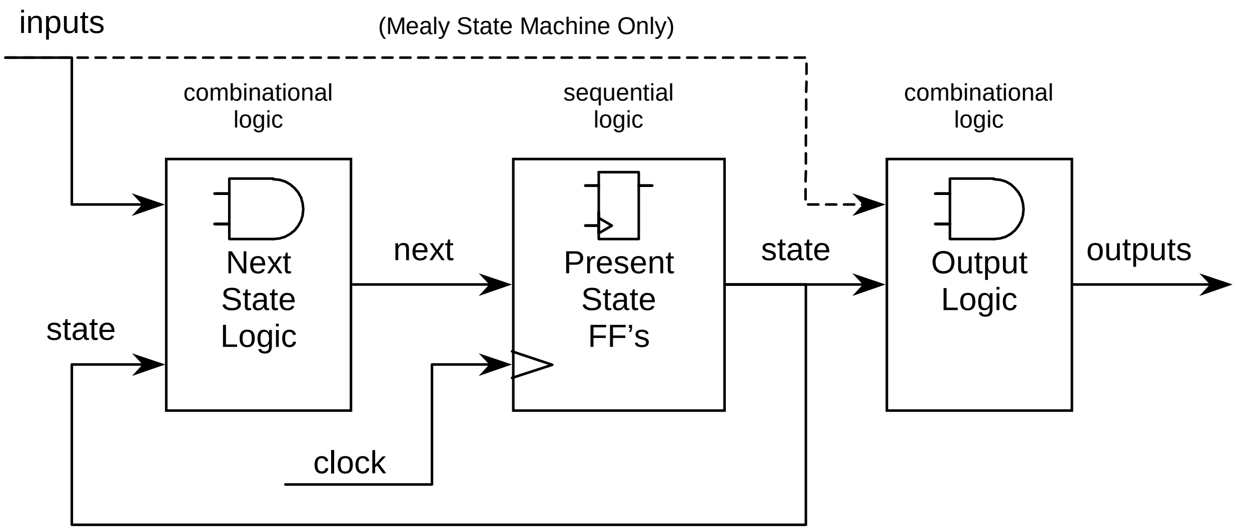


Figure 1 - FSM Block Diagram

In addition to classifying state machines by their respective output-generation type, state machines are also often classified by the state encoding employed by each state machine. Some of the more common state encoding styles include [1] [2] [3]: highly-encoded binary (or binary-sequential), gray-code, Johnson, one-hot, almost one-hot and one-hot with zero-idle (note: in the absence of a known official designation for the last encoding-style listed, the author selected the "one-hot with zero-idle" title. A more generally accepted name may exist).

Using the Moore FSM state diagram shown in Figure 2, this paper will detail synthesizable Verilog coding styles for highly-encoded binary, one-hot and one-hot with zero-idle state machines. This paper also details usage of the Synopsys FSM Tool to generate binary, gray and one-hot state machines. Coded examples of the three coding styles for the state machine in Figure

2, plus an example with the correct Synopsys FSM Tool comments, have been included at the end of this paper.

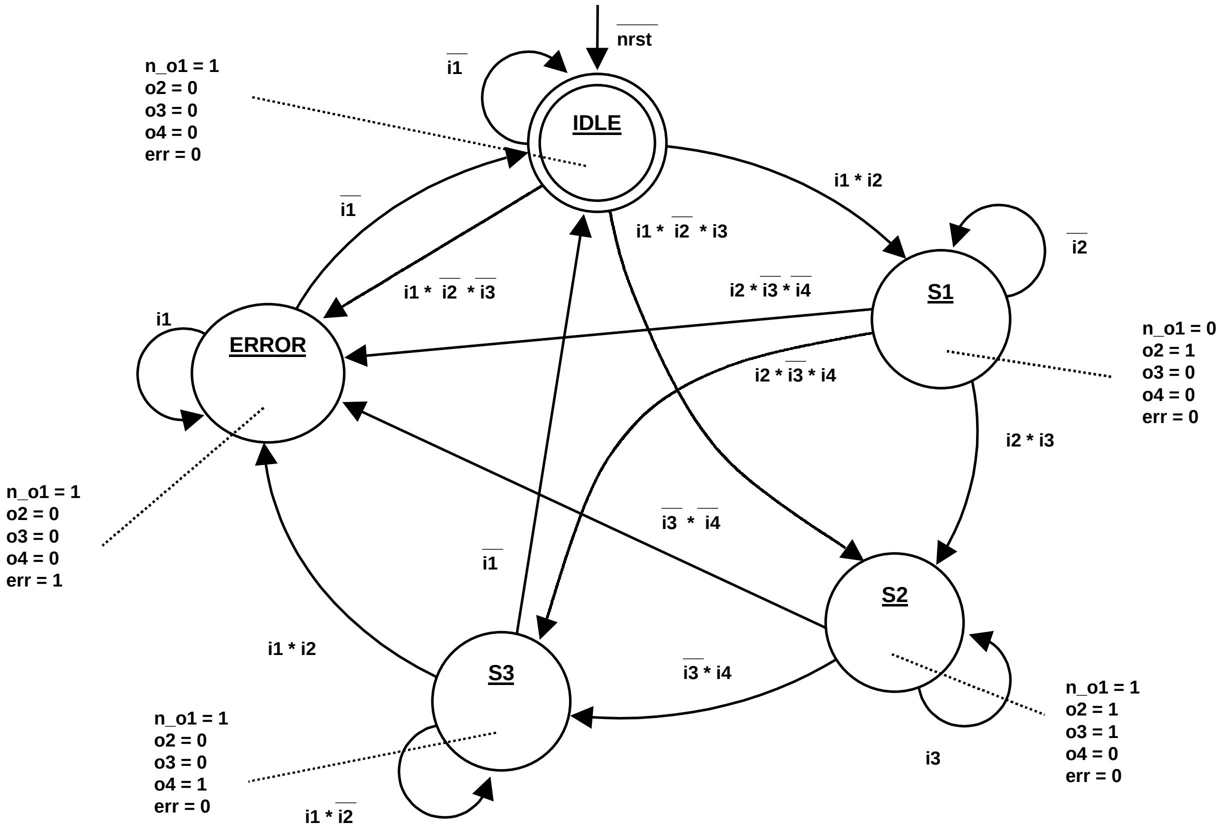


Figure 2 - Benchmark 1 (bm1) State Diagram

FSM Verilog Modules

Guideline: make each state machine a separate Verilog module.

Keeping each state machine separate from other synthesized logic simplifies the tasks of state machine definition, modification and debug. There are also a number of EDA tools that assist in the design and documentation of FSMs, but in general they only work well if the FSM is not mingled with other logic-code.

State Assignments

Guideline: make state assignments using parameters with symbolic state names.

Defining and using symbolic state names makes the Verilog code more readable and eases the task of redefining states if necessary. Examples 1-3 show binary, one-hot and one-hot with zero-idle parameter definitions for the FSM state diagram in Figure 2.

```
parameter [2:0] // synopsys enum code
                  IDLE = 3'd0,
                    S1 = 3'd1,
                    S2 = 3'd2,
                    S3 = 3'd3,
                 ERROR = 3'd4;
```

Example 1 - Parameter definitions for binary encoding

```
parameter [4:0] IDLE = 5'b00001,
                    S1 = 5'b00010,
                    S2 = 5'b00100,
                    S3 = 5'b01000,
                 ERROR = 5'b10000;
```

Example 2 - Parameter definitions for verbose one-hot encoding

```
parameter [4:0] IDLE = 5'd0,
                    S1 = 5'd1,
                    S2 = 5'd2,
                    S3 = 5'd3,
                 ERROR = 5'd4;
```

Example 3 - Parameter definitions for simplified one-hot encoding

The simplified one-hot encoding shown Example 3 uses decimal numbers to index into the state register. This technique permits comparison of single bits as opposed to comparing against the entire state vector using the full state parameters shown in Example 2.

```
parameter [4:1] // ERROR is 4'b0000
                  IDLE = 4'd1,
                    S1 = 4'd2,
                    S2 = 4'd3,
                    S3 = 4'd4;
```

Example 4 - Parameter definitions for one-hot with zero-idle encoding

The one-hot with zero-idle encoding can yield very efficient FSMs for state machines that have many interconnections with complex equations, including a large number of connections to one particular state. Frequently, multiple transitions are made either to an IDLE state or to another common state (such as the ERROR-state in this example).

One could also define symbolic state names using the macro-definition compiler directives (`define), but `define creates a global definition (from the point where the definition is read in the Verilog-code input stream). Unlike `define constants, parameters are constants local to the module where they are declared, which allows a design to have multiple FSMs with duplicate state names, such as IDLE or READ, each with a unique state encoding.

Occasionally, FSM code is written with parameter-defined state definitions, but subsequent code still includes explicit binary state encodings elsewhere in the module. This defeats the purpose of using symbolically labeled parameters. Only use the pre-defined parameter names for state testing and next-state assignment.

Additional notes on experimenting with different state definitions using Synopsys generated binary, gray and one-hot encodings are detailed in the section, "Synopsys FSM Tool."

Two-Always Block State Machine

A synthesizable state machine may be coded many ways. Two of the most common, easily understood and efficient methods are two-always block and one-always block state machines.

The easiest method to understand and implement is the two-always block state machine with output assignments included in either the combinational next-state always block or separate continuous-assignment outputs. This method partitions the Verilog code into the major FSM blocks shown in Figure 1: clocked present state logic, next state combinational logic and output combinational logic.

Sequential Always Block

Guideline: only use Verilog nonblocking assignments in the sequential always block.

Guideline: only use Verilog nonblocking assignments in all always blocks used to generate sequential logic.

For additional information concerning nonblocking assignments, see reference [4].

Verilog nonblocking assignments mimic the pipelined register behavior of actual hardware and eliminate many potential Verilog race conditions. Many engineers make nonblocking assignments using an intra-assignment timing delay (as shown in Example 5). There are two good reasons and one bad reason for using intra-assignment timing delays with nonblocking assignments.

Good reasons: (1) gives the appearance of a clk->q delay on a clocked register (as seen using a waveform viewer); (2) helps avoid hold-time problems when driving most gate-level models from an RTL model.

Bad reason: "we add delays because Verilog nonblocking assignments are broken!" - This is not true.

When implementing either a binary encoded or a verbose one-hot encoded FSM, on reset the state register will be assigned the IDLE state (or equivalent) (Example 5).

```
always @(posedge clk or posedge rst)
    if (rst)   state <= #1 IDLE;
    else       state <= #1 next;
```

Example 5 - Sequential always block for binary and verbose one-hot encoding

When implementing a simplified one-hot encoded FSM, on reset the state register will be assigned all zeros followed immediately by reassigning the IDLE bit of the state register (Example 6). Note, there are two nonblocking assignments assigning values to the same bit. This is completely defined by the IEEE Verilog Standard [5] and in this case, the last nonblocking assignment supercedes any previous nonblocking assignment (updating the IDLE bit of the state register).

```
always @(posedge clk or posedge rst)
    if (rst) begin
               state <= 5'b0;
               state[IDLE] <= 1'b1;
             end
    else       state <= next;
```

Example 6 - Sequential always block for simplified one-hot encoding

When implementing a one-hot with zero-idle encoded FSM, on reset the state register will be assigned all zeros (Example 7).

```
always @(posedge clk or posedge rst)
    if (rst) state <= 4'b0;
    else     state <= next;
```

Example 7 - Sequential always block one-hot with zero-idle encoding

Combinational Always Block

Guideline: only use Verilog blocking assignments in combinational always blocks.

Code a combinational always block to update the next state value. This always block is triggered by a sensitivity list that is sensitive to the state register from the synchronous always block and all of the inputs to the state machine.

Place a default next state assignment on the line immediately following the always block sensitivity list. This default assignment is updated by next-state assignments inside the case statement. There are three types of default next-state assignments that are commonly used: (1) next is set to all x's, (2) next is set to a predetermined recovery state such as IDLE, or (3) next is just set to the value of the state register.

By making a default next state assignment of x's, pre-synthesis simulation models will cause the state machine outputs to go unknown if not all state transitions have been explicitly assigned in the case statement. This is a useful technique to debug state machine designs, plus the x's will be treated as "don't cares" by the synthesis tool.

Some designs require an assignment to a known state as opposed to assigning x's. Examples include: satellite applications, medical applications, designs that use the FSM flip-flops as part of a diagnostic scan chain and designs that are equivalence checked with formal verification tools. Making a default next state assignment of either IDLE or all 0's typically satisfy these design requirements and making the initial default assignment might be easier than coding all of the explicit next-state transition assignments in the case statement.

Making the default next-state assignment equal to the present state is a coding style that has been used by PLD designers for years.

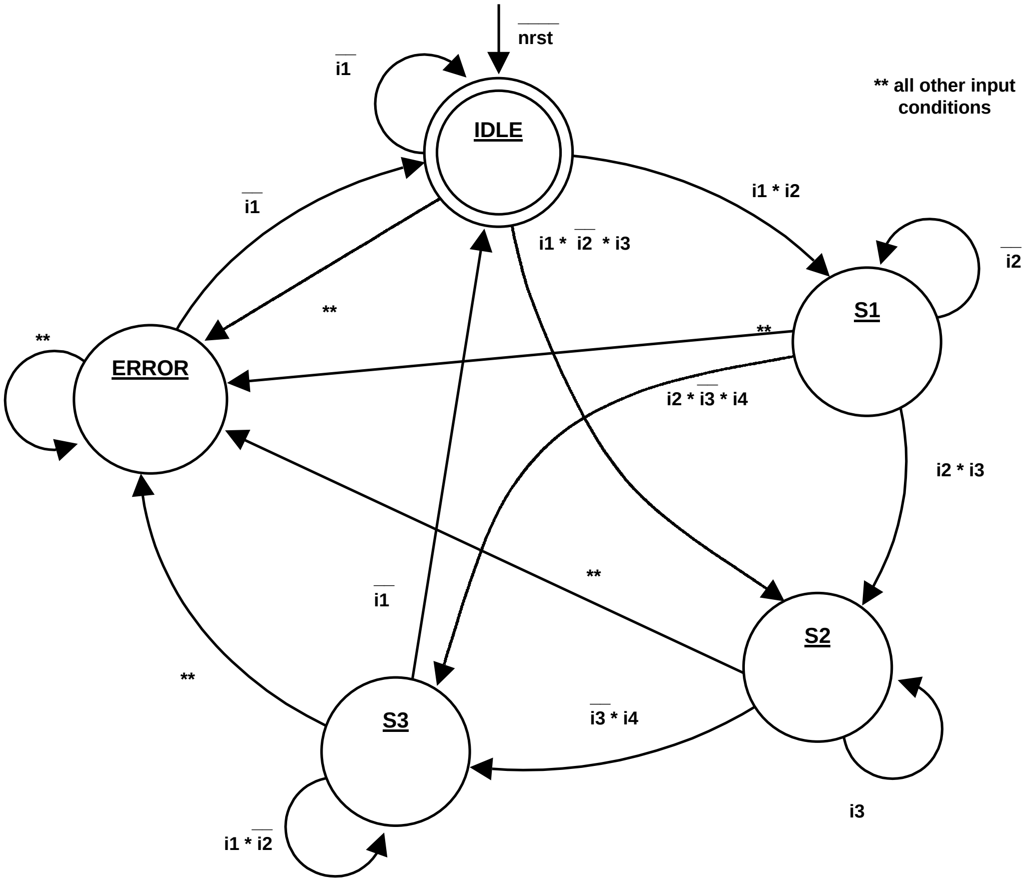


Figure 3 - Next State Transitions

Next state assignments are efficiently updated from within a case statement.

```
always @(state or i1 or i2 or i3 or i4) begin
    next = 3'bx;
    case (state)
    IDLE: begin
                              next = ERROR;
          if (!i1)            next = IDLE;
          if (i1 & i2)        next = S1;
          if (i1 & !i2 & i3)  next = S2;
        end

    S1: ...
```

Example 8 - Next state assignments for binary and verbose one-hot encoding

```
always @(state or i1 or i2 or i3 or i4) begin
    next = 5'b0;
    case (1'b1)  // synopsys full_case parallel_case
    state[IDLE]: begin
          if      (!i1) next[IDLE]  = 1'b1;
          else if ( i2) next[S1]    = 1'b1;
          else if ( i3) next[S2]    = 1'b1;
          else          next[ERROR] = 1'b1;
        end
    state[S1]: ...
```

Example 9 - Next state assignments for simplified one-hot encoding

```
always @(state or i1 or i2 or i3 or i4) begin
    next = 4'b0;
    case (1'b1)  // synopsys full_case parallel_case
    ~|state: begin // IDLE
          if ( i1 &  i2)       next[S1]    = 1'b1;
          if ( i1 & !i2 &  i3) next[S2]    = 1'b1;
          if ( i1 & !i2 & !i3) next[ERROR] = 1'b1;
        end
    state[S1]: ...
```

Example 10 - Next state assignments for simplified one-hot with zero-idle encoding

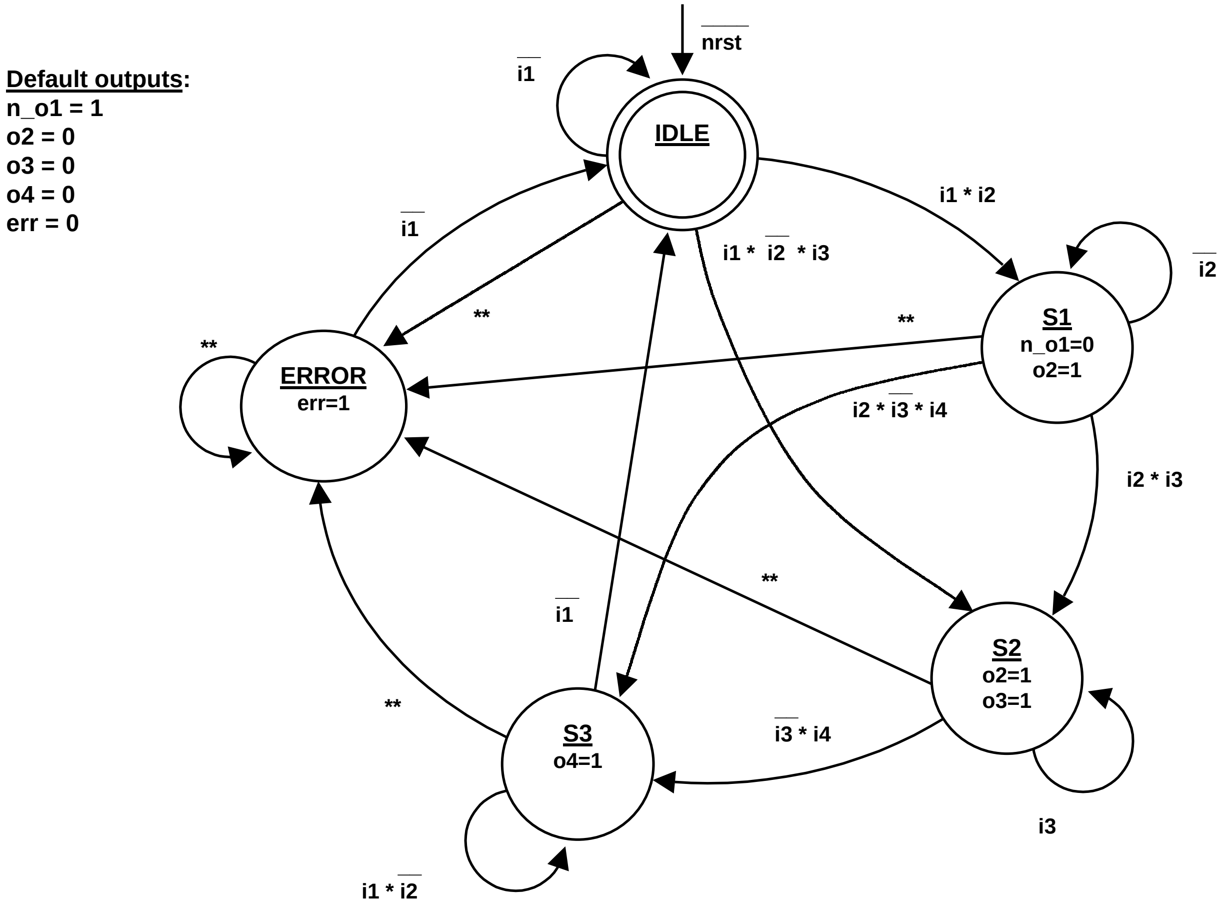


Figure 4 - State Diagram

FSM Output Generation

Code the output logic either as a separate block of continuous assignments or within the combinational logic always block. If the output assignments are coded as part of the combinational always block, output assignments could also be put into Verilog tasks with meaningful names, as shown in Figure 5. The tasks are called from within each state in the case statement.

Isolation of the output assignments makes changes to the output logic easy if modification is required. It also helps to avoid the creation of additional unwanted latches by the synthesis tool.

When placing output assignments inside the combinational always block of a Two-Always Block State Machine, make default output assignments at the top of the always block, then modify the appropriate output assignments in the case statement.

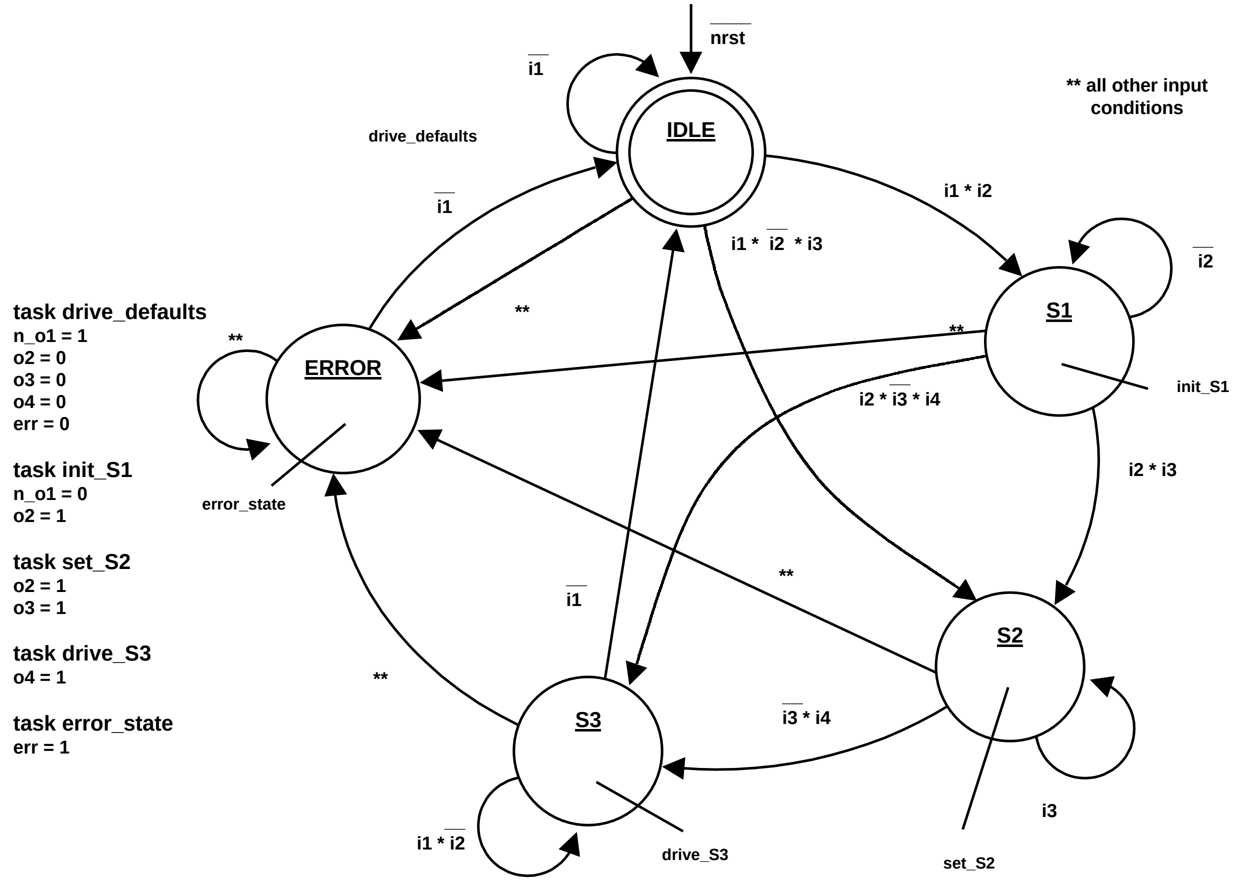


Figure 5 - Task State Outputs

In general this method requires less coding than making all output assignments for each state (case item) and highlights when outputs are supposed to change.

Mealy and Registered Outputs

Mealy outputs are easily added to the Verilog code either by qualifying an output continuous assignment:

```
assign rd_out = (state == READ) & !rd_strobe_n;
```

or by qualifying an output assignment in the combinational always block:

```
case (state)
...
READ: if (!rd_strobe_n) rd_out = 1'b1;
```

Registered outputs may be added to the Verilog code making assignments to an output using nonblocking assignments in a sequential always block. The FSM can be coded as one sequential always block or a second sequential always block can be added to the design.

One-Always Block State Machine

In general, the one-always block state machine is slightly more simulation-efficient than the two-always block state machine since the inputs are only examined on clock changes; however, this state machine can be more difficult to modify and debug.

When placing output assignments inside the always block of a one-always block state machine, one must consider the following:

Placing output assignments inside of the always block will infer output flip-flops. It must also be remembered that output assignments placed inside of the always block are "next output" assignments which can be more error-prone to code.

Note: output assignments inside of a sequential always block cannot be Mealy outputs.

Full_case / parallel_case

A case statement is a "select-one-of-many" construct in both Verilog and VHDL. A case statement is composed of the keyword, case, followed by a case expression that is compared to subsequent case items. The case items are tested against the case expression, one by one, in sequential order and when a match between the case expression and one of the case items is detected, the corresponding actions executed, the rest of the case items are skipped and program execution resumes with the first statement after the endcase statement.

```
case (case_expression (with 2^n possible combinations))
   case_item1 : <action #1>;
   case_item2 : <action #2>;
   case_item3 : <action #3>;
   ...
```

```
    case_item2$^{n-1}$: <action #2$^{n-1}$>;
    case_item2$^{n}$: <action #2$^{n}$>;
    default: <default action>;
endcase
```

A full case statement is defined to be a case statement where every possible input pattern is explicitly defined. A parallel case statement is defined to be a case statement with no overlapping conditions in the case items.

VHDL case statements are required to be "full," which means that every possible case item shall either be explicitly listed as a case item, or there must be an "others =>" clause after the last-defined case item. In practice, almost all VHDL case statements utilizing non bit-type data types include an "others =>" statement to cover the non-binary data patterns.

VHDL case statements are also required to be "parallel," which means that no case item shall overlap any other in the list of case items.

Verilog case statements are not required to be either "full" or "parallel."

Adding "// synopsys full_case" to the end of a case statement (before any case items are declared) informs the synthesis tool that all outputs from non-explicitly declared case items should be treated as "don't-cares" for synthesis purposes.

Adding "// synopsys parallel_case" to the end of a case statement (before any case items are declared) informs the synthesis tool that all case items should be tested individually, even if the case items overlap.

Adding either or both "// synopsys full_case parallel_case" directives to the Verilog FSM source code is generally beneficial when coding one-hot or one-hot with zero-idle FSMs. In these cases, it is given that only one bit of the state vector is set and that all other bit-pattern combinations should be treated as "don't cares." It is also given that there should be no overlap in the list of case items.

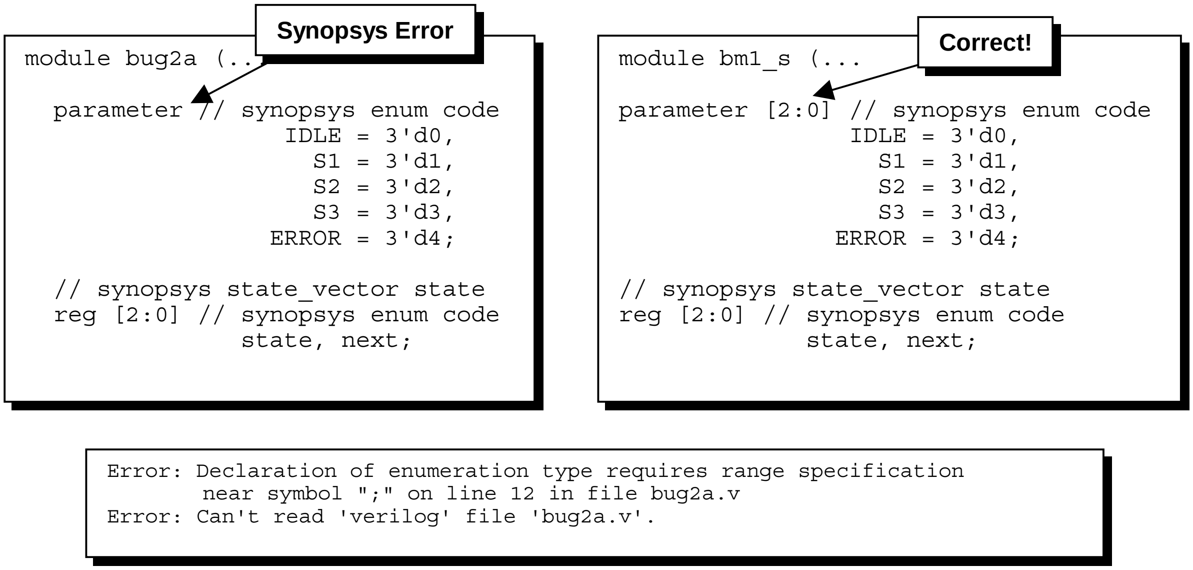
Note that the usage of full_case parallel case may cause pre-synthesis design simulations to differ from post-synthesis design simulations because these directives are effectively giving Synopsys tools information about the design that was not included in the original Verilog model.

Adding full_case parallel_case to every case statement in a design is not recommended. The practice can change the functionality of a design, and can also cause some binary encoded FSM designs to actually get larger and slower.

Synopsys FSM Tool

The Synopsys FSM tool can be used to experiment with different state encodings styles, such as binary, gray and one-hot codes. In order to use the FSM tool, the Verilog code must include Synopsys synthetic comments, plus a few unusual Verilog code statements. The Synopsys FSM tool is very strict about how these comments and code segments are ordered and it is very easy to code this incorrectly for the FSM tool.

First, the parameter must include a range (very unusual Verilog coding style). If no range is included in the parameter declaration, the error message "Declaration of enumeration type requires range specification" will be reported.



Synopsys Error

```
module bug2a (...

  parameter // synopsys enum code
                  IDLE = 3'd0,
                    S1 = 3'd1,
                    S2 = 3'd2,
                    S3 = 3'd3,
                 ERROR = 3'd4;

  // synopsys state_vector state
  reg [2:0] // synopsys enum code
               state, next;
```

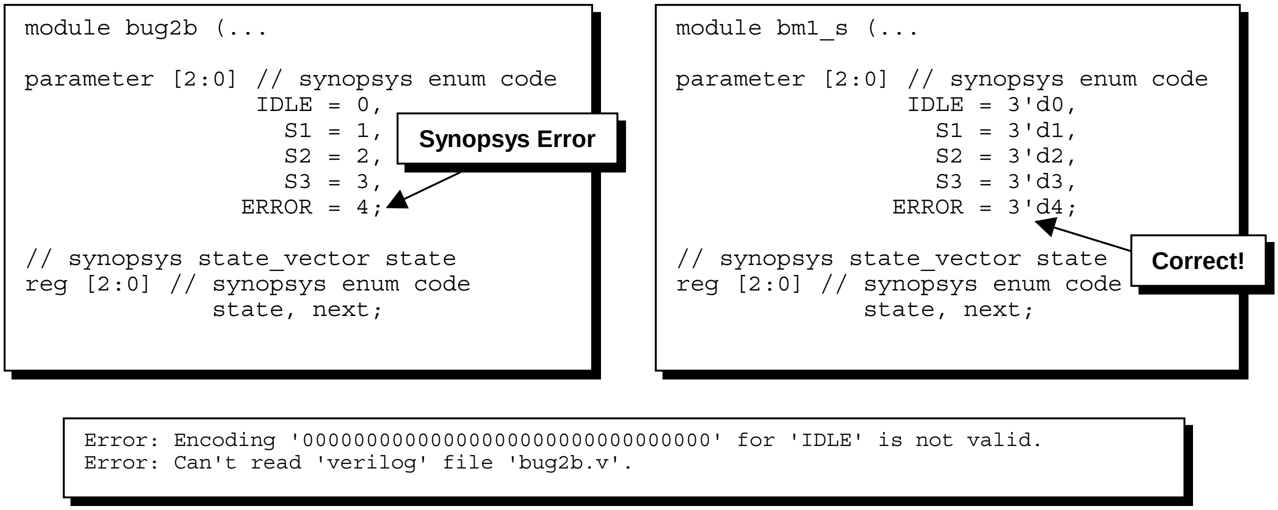
Correct!

```
module bm1_s (...

parameter [2:0] // synopsys enum code
                IDLE = 3'd0,
                  S1 = 3'd1,
                  S2 = 3'd2,
                  S3 = 3'd3,
               ERROR = 3'd4;

// synopsys state_vector state
reg [2:0] // synopsys enum code
             state, next;
```

```
Error: Declaration of enumeration type requires range specification
       near symbol ";" on line 12 in file bug2a.v
Error: Can't read 'verilog' file 'bug2a.v'.
```

Figure 6 - FSM Tool - parameter range

Second, numeric parameter definitions must be sized, otherwise the FSM tool interprets all numbers as 32-bit numbers and reports an invalid encoding error.



```
module bug2b (...

parameter [2:0] // synopsys enum code
                IDLE = 0,
                  S1 = 1,
                  S2 = 2,
                  S3 = 3,
               ERROR = 4;

// synopsys state_vector state
reg [2:0] // synopsys enum code
             state, next;
```

Synopsys Error

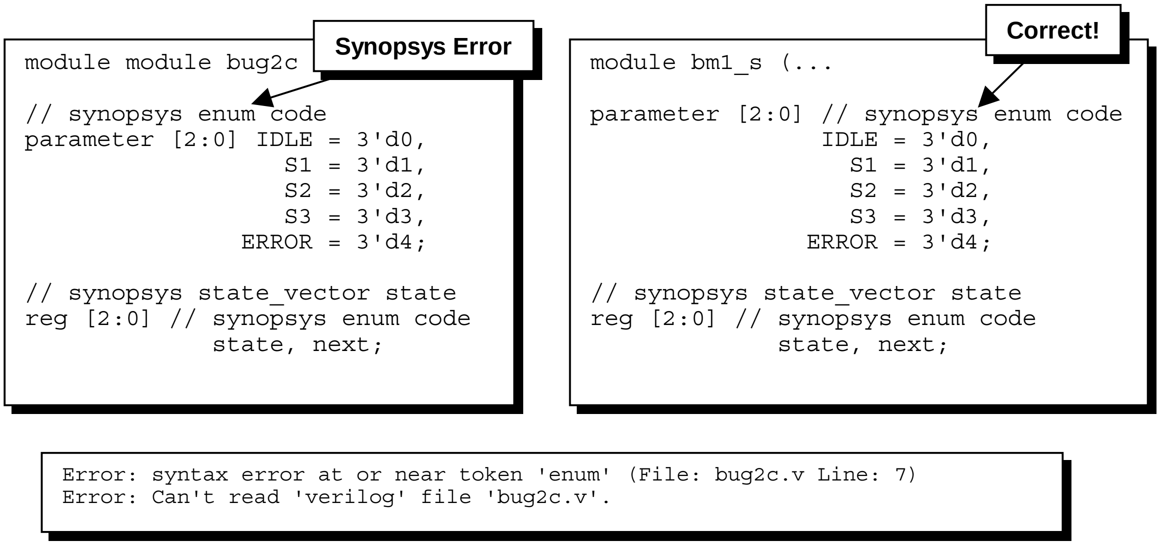
```
module bm1_s (...

parameter [2:0] // synopsys enum code
                IDLE = 3'd0,
                  S1 = 3'd1,
                  S2 = 3'd2,
                  S3 = 3'd3,
               ERROR = 3'd4;

// synopsys state_vector state
reg [2:0] // synopsys enum code
             state, next;
```

Correct!

```
Error: Encoding '00000000000000000000000000000000' for 'IDLE' is not valid.
Error: Can't read 'verilog' file 'bug2b.v'.
```

Figure 7 - FSM Tool - sized numbers

Third, the required placement of the Synopsys synthetic comments is exactly as shown. The "// synopsys enum <name>" must be placed after the parameter range declaration and before any of the parameters are declared,


Synopsys Error
Correct!

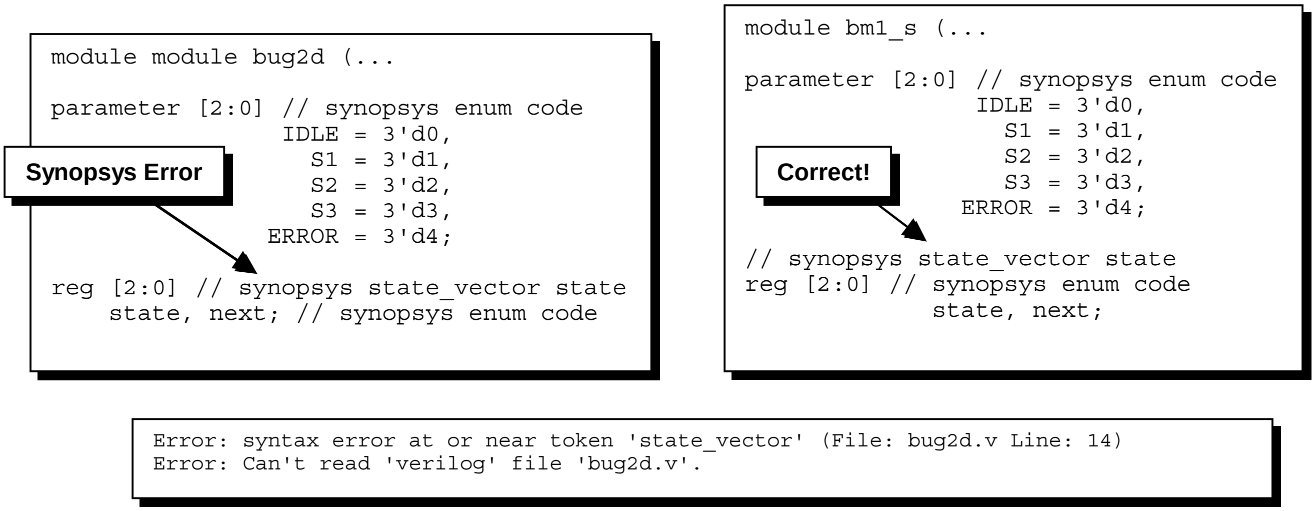
```
module module bug2c

// synopsys enum code
parameter [2:0] IDLE = 3'd0,
                  S1 = 3'd1,
                  S2 = 3'd2,
                  S3 = 3'd3,
               ERROR = 3'd4;

// synopsys state_vector state
reg [2:0] // synopsys enum code
             state, next;
```

```
module bm1_s (...

parameter [2:0] // synopsys enum code
                IDLE = 3'd0,
                  S1 = 3'd1,
                  S2 = 3'd2,
                  S3 = 3'd3,
               ERROR = 3'd4;

// synopsys state_vector state
reg [2:0] // synopsys enum code
             state, next;
```

```
Error: syntax error at or near token 'enum' (File: bug2c.v Line: 7)
Error: Can't read 'verilog' file 'bug2c.v'.
```

Figure 8 - FSM Tool - synopsys enum

a "/ synopsys state_vector <state_vector_name>" Synopsys comment must be placed immediately before the state-reg declaration and the exact same "// synopsys enum <name>" comment, used above, must be placed after the reg range declaration but before the state (and next) declarations. "


Synopsys Error
Correct!

```
module module bug2d (...

parameter [2:0] // synopsys enum code
                IDLE = 3'd0,
                  S1 = 3'd1,
                  S2 = 3'd2,
                  S3 = 3'd3,
               ERROR = 3'd4;

reg [2:0] // synopsys state_vector state
    state, next; // synopsys enum code
```

```
module bm1_s (...

parameter [2:0] // synopsys enum code
                IDLE = 3'd0,
                  S1 = 3'd1,
                  S2 = 3'd2,
                  S3 = 3'd3,
               ERROR = 3'd4;

// synopsys state_vector state
reg [2:0] // synopsys enum code
             state, next;
```

```
Error: syntax error at or near token 'state_vector' (File: bug2d.v Line: 14)
Error: Can't read 'verilog' file 'bug2d.v'.
```

Figure 9 - FSM Tool - synopsys state_vector

Below are example dc_shell commands that are used to invoke the Synopsys FSM tools on a state machine design.

```
(read the design)
compile
extract
set_fsm_encoding_style binary
compile
write -f db -hier -output "db/" + DESIGN + "_fsm_binary.db"
report_area > "rpt/" + DESIGN + "_fsm_binary.rpt"
create_clock -p 0 clk
report_timing >> "rpt/" + DESIGN + "_fsm_binary.rpt"

(read the design)
compile
extract
set_fsm_encoding_style gray
compile
write -f db -hier -output "db/" + DESIGN + "_fsm_gray.db"
report_area > "rpt/" + DESIGN + "_fsm_gray.rpt"
create_clock -p 0 clk
report_timing >> "rpt/" + DESIGN + "_fsm_gray.rpt"

(read the design)
compile
extract
set_fsm_encoding_style one_hot
compile
write -f db -hier -output "db/" + DESIGN + "_fsm_onehot.db"
report_area > "rpt/" + DESIGN + "_fsm_onehot.rpt"
create_clock -p 0 clk
report_timing >> "rpt/" + DESIGN + "_fsm_onehot.rpt"
```

Example 11 - FSM Tool - dc_shell script

Acknowledgements

I would like to thank both Mike McNamara of Silicon Sorcery and Steve Golson of Trilobyte Systems for information and tips they have shared with me concerning Finite State Machine design. For more information about coding State Machines in both Verilog and VHDL, I highly recommend reading Steve's paper, "State Machine Design Techniques for Verilog and VHDL" [1].

References

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

[2] Z. Kohavi, "Switching and Finite Automata Theory," McGraw-Hill Book Company, New York, 1978, pp. 275-321.

[3] D.J. Smith, "HDL Chip Design," Doone Publications, Madison, Alabama, 1997, pp. 193-270.

[4] C.E. Cummings, "Verilog Nonblocking Assignments Demystified," International Verilog HDL Conference Proceedings 1998.

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

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 19 years of ASIC, FPGA and system design experience and nine years of Verilog, synthesis and methodology training experience.

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

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

E-mail Address: cliffc@sunburst-design.com
This paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of September 7th, 2001)

```
module bm1_s (err, n_o1, o2, o3, o4,
            i1, i2, i3, i4, clk, rst);
  output    err, n_o1, o2, o3, o4;
  input     i1, i2, i3, i4, clk, rst;
  reg       err, n_o1, o2, o3, o4;

  parameter [2:0] // synopsys enum code
                  IDLE = 3'd0,
                    S1 = 3'd1,
                    S2 = 3'd2,
                    S3 = 3'd3,
                 ERROR = 3'd4;

  // synopsys state_vector state
  reg [2:0] // synopsys enum code
               state, next;

  always @(posedge clk or posedge rst)
    if (rst)   state <= IDLE;
    else       state <= next;

  always @(state or i1 or i2 or i3 or i4) begin
    next = 3'bx;
    err = 0; n_o1 = 1;
    o2 = 0;  o3 = 0;  o4 = 0;
    case (state)
    IDLE: begin
                               next = ERROR;
          if (!i1)             next = IDLE;
          if (i1 & i2)         next = S1;
          if (i1 & !i2 & i3)   next = S2;
        end

    S1: begin
                               next = ERROR;
          if (!i2)             next = S1;
          if (i2 & i3)         next = S2;
          if (i2 & !i3 & i4)   next = S3;
          n_o1 = 0;
          o2 = 1;
        end

    S2: begin
                               next = ERROR;
          if (i3)              next = S2;
          if (!i3 & i4)        next = S3;
          o2 = 1;
          o3 = 1;
        end

    S3: begin
                               next = S3;
          if (!i1)             next = IDLE;
          if (i1 & i2)         next = ERROR;
          o4 = 1;
        end

    ERROR: begin
                               next = IDLE;
          if (i1)              next = ERROR;
          err = 1;
        end
    endcase
  end
endmodule
```

Synopsys FSM Tool synthetic comment

Highly encoded state-parameter definitions

Synopsys FSM Tool synthetic comments

next = 2'bx (synthesis "don't care" assignment)

Initial default output assignments

Default assignment followed by parallel if statements

Only update output assignments that change in each state

Figure 10 - FSM Tool synthetic comments

```
module bm1_1afp (err, n_o1, o2, o3, o4,
               i1, i2, i3, i4, clk, rst);
  output  err, n_o1, o2, o3, o4;
  input   i1, i2, i3, i4, clk, rst;
  reg     err, n_o1, o2, o3, o4;

  parameter [4:0] IDLE = 5'b00001,
                    S1 = 5'b00010,
                    S2 = 5'b00100,
                    S3 = 5'b01000,
                 ERROR = 5'b10000;

  reg [4:0]  state, next;

  always @(posedge clk or posedge rst)
    if (rst) state <= IDLE;
    else     state <= next;

  always @(state or i1 or i2 or i3 or i4) begin
    next = 5'bx;
    err = 0; n_o1 = 1;
    o2 = 0;    o3 = 0;  o4 = 0;
    case (state)  // synopsys full_case parallel_case
    IDLE: begin
          if      (!i1) next = IDLE;
          else if ( i2) next = S1;
          else if ( i3) next = S2;
          else          next = ERROR;
        end

    S1: begin
          if      (!i2) next = S1;
          else if ( i3) next = S2;
          else if ( i4) next = S3;
          else          next = ERROR;
          n_o1 = 0;
          o2 = 1;
        end

    S2: begin
          if      ( i3) next = S2;
          else if ( i4) next = S3;
          else          next = ERROR;
          o2 = 1;
          o3 = 1;
        end

    S3: begin
          if      (!i1) next = IDLE;
          else if ( i2) next = ERROR;
          else          next = S3;
          o4 = 1;
        end

    ERROR: begin
          if (i1)       next = ERROR;
          else          next = IDLE;
          err = 1;
        end
    endcase
  end
endmodule
```

Verbose one-hot state-parameter definitions

next = 5'bx (synthesis "don't care" assignment)

Initial default output assignments

If/else-if statements

Final else statements

Only update output assignments that change in each state

Figure 11 - Verbose one-hot FSM

```
module bm1_1fpt (err, n_o1, o2, o3, o4,
               i1, i2, i3, i4, clk, rst);
  output  err, n_o1, o2, o3, o4;
  input   i1, i2, i3, i4, clk, rst;
  reg     err, n_o1, o2, o3, o4;

  parameter [4:0] IDLE = 5'd0,
                    S1 = 5'd1,
                    S2 = 5'd2,
                    S3 = 5'd3,
                 ERROR = 5'd4;

  reg [4:0]  state, next;

  always @(posedge clk or posedge rst)
    if (rst) begin
               state <= 5'b0;
               state[IDLE] <= 1'b1;
             end
    else       state <= next;

  always @(state or i1 or i2 or i3 or i4) begin
    next = 5'b0;
    drive_defaults;
    case (1'b1)  // synopsys full_case parallel_case
    state[IDLE]: begin
          if      (!i1) next[IDLE]  = 1'b1;
          else if ( i2) next[S1]    = 1'b1;
          else if ( i3) next[S2]    = 1'b1;
          else          next[ERROR] = 1'b1;
        end

    state[S1]: begin
          if      (!i2) next[S1]    = 1'b1;
          else if ( i3) next[S2]    = 1'b1;
          else if ( i4) next[S3]    = 1'b1;
          else          next[ERROR] = 1'b1;
          init_S1;
        end

    state[S2]: begin
          if      ( i3) next[S2]    = 1'b1;
          else if ( i4) next[S3]    = 1'b1;
          else          next[ERROR] = 1'b1;
          set_S2;
        end

    state[S3]: begin
          if      (!i1) next[IDLE]  = 1'b1;
          else if ( i2) next[ERROR] = 1'b1;
          else          next[S3]    = 1'b1;
          drive_S3;
        end

    state[ERROR]: begin
          if (i1)       next[ERROR] = 1'b1;
          else          next[IDLE]  = 1'b1;
          error_state;
        end

    endcase
  end

  task drive_defaults;
    begin
      err = 0;
      n_o1 = 1;
      o2 = 0;
      o3 = 0;
      o4 = 0;
    end
  endtask
```

Simplified one-hot state-parameter definitions
(to index into the state vector)

On reset, state <= 5'b0
followed by state[IDLE] <= 1'b1

next is set to all 0's

Synopsys full_case parallel_case helps infer a more efficient one-hot implementation

Only set the "one-hot" bit in the next register

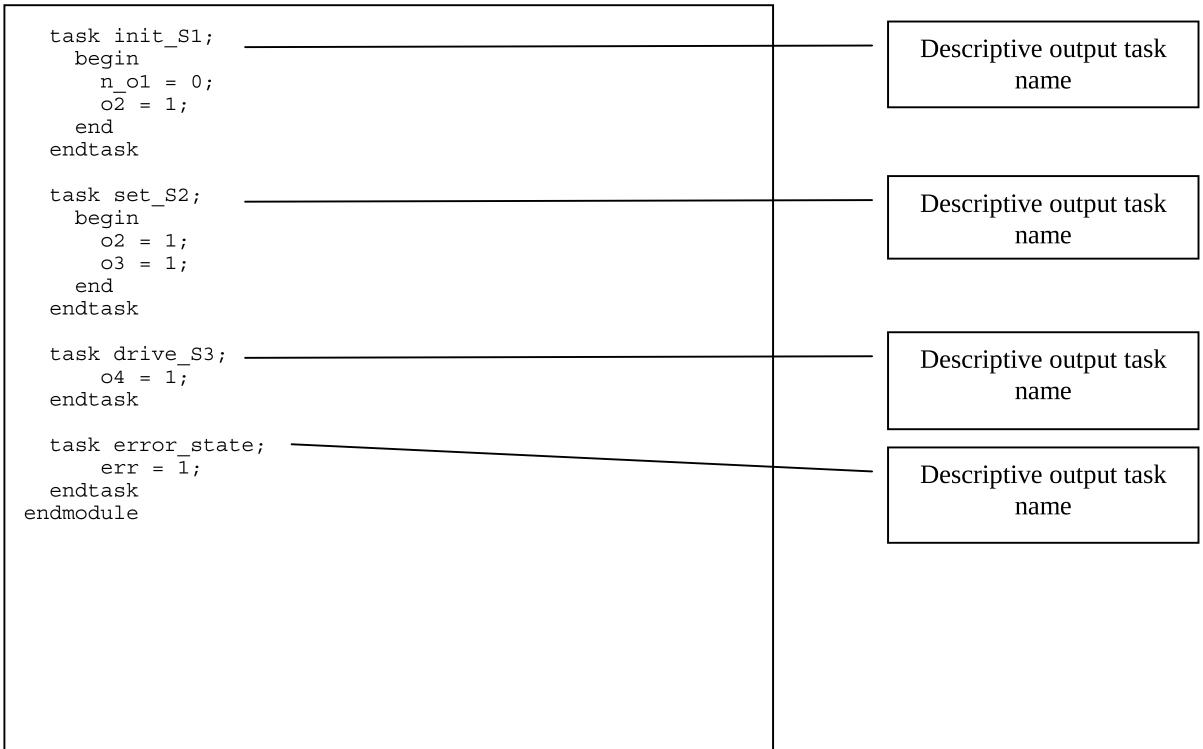
Case "if true" (1'b1) ...

... match a single state bit

Output task call

Descriptive output task names

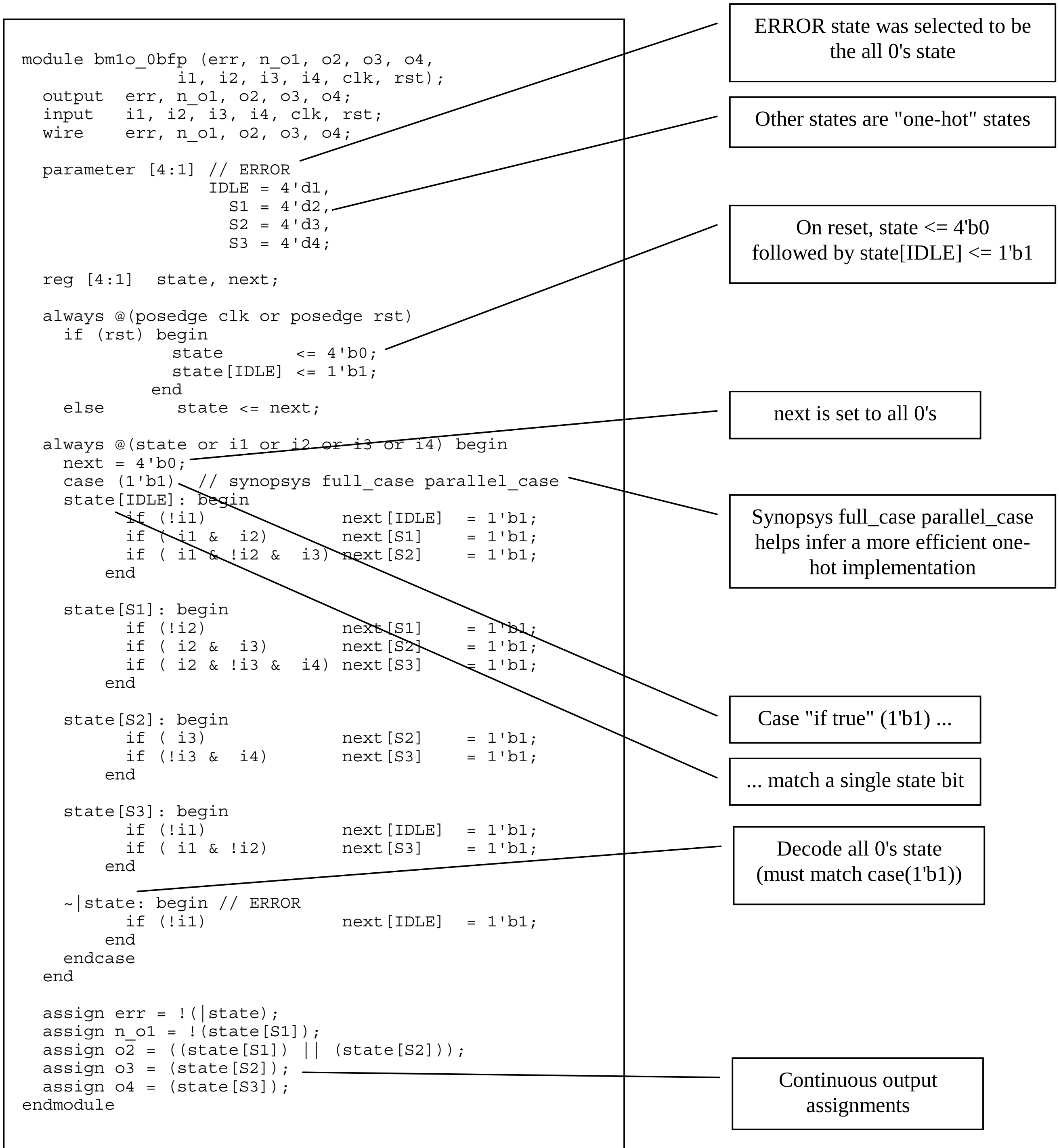
Figure 12 - Simplified one-hot FSM w/task outputs



```
  task init_S1;
    begin
      n_o1 = 0;
      o2 = 1;
    end
  endtask

  task set_S2;
    begin
      o2 = 1;
      o3 = 1;
    end
  endtask

  task drive_S3;
      o4 = 1;
  endtask

  task error_state;
      err = 1;
  endtask
endmodule
```

Descriptive output task name
Descriptive output task name
Descriptive output task name
Descriptive output task name

Figure 13 - Simplified one-hot FSM w/task outputs (cont.)



```
module bm1o_0bfp (err, n_o1, o2, o3, o4,
               i1, i2, i3, i4, clk, rst);
  output  err, n_o1, o2, o3, o4;
  input   i1, i2, i3, i4, clk, rst;
  wire    err, n_o1, o2, o3, o4;

  parameter [4:1] // ERROR
                  IDLE = 4'd1,
                    S1 = 4'd2,
                    S2 = 4'd3,
                    S3 = 4'd4;

  reg [4:1]  state, next;

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

  always @(state or i1 or i2 or i3 or i4) begin
    next = 4'b0;
    case (1'b1)  // synopsys full_case parallel_case
    state[IDLE]: begin
          if (!i1)             next[IDLE]  = 1'b1;
          if ( i1 &  i2)       next[S1]    = 1'b1;
          if ( i1 & !i2 &  i3) next[S2]    = 1'b1;
        end

    state[S1]: begin
          if (!i2)             next[S1]    = 1'b1;
          if ( i2 &  i3)       next[S2]    = 1'b1;
          if ( i2 & !i3 &  i4) next[S3]    = 1'b1;
        end

    state[S2]: begin
          if ( i3)             next[S2]    = 1'b1;
          if (!i3 &  i4)       next[S3]    = 1'b1;
        end

    state[S3]: begin
          if (!i1)             next[IDLE]  = 1'b1;
          if ( i1 & !i2)       next[S3]    = 1'b1;
        end

    ~|state: begin // ERROR
          if (!i1)             next[IDLE]  = 1'b1;
        end
    endcase
  end

  assign err = !(|state);
  assign n_o1 = !(state[S1]);
  assign o2 = ((state[S1]) || (state[S2]));
  assign o3 = (state[S2]);
  assign o4 = (state[S3]);
endmodule
```

ERROR state was selected to be the all 0's state

Other states are "one-hot" states

On reset, state <= 4'b0 followed by state[IDLE] <= 1'b1

next is set to all 0's

Synopsys full_case parallel_case helps infer a more efficient one-hot implementation

Case "if true" (1'b1) ...

... match a single state bit

Decode all 0's state (must match case(1'b1))

Continuous output assignments

Figure 14 - One-hot with zero-idle FSM