SNUG-2019
Silicon Valley, CA
Voted Best Presentation
1st Place

World Class SystemVerilog & UVM Training

Finite State Machine (FSM) Design & Synthesis using SystemVerilog - Part I

Clifford E. Cummings
Sunburst Design, Inc.
Provo, UT, USA
www.sunburst-design.com

Heath Chambers
HMC Design Verification, Inc.
Albuquerque, NM, USA

ABSTRACT

There are at least seven different Finite State Machine (FSM) design techniques that are commonly taught, one with combinatorial outputs and six with registered outputs. This paper will describe four of the FSM design techniques: (1) 1-Always Block Style with registered outputs, (2) 2-Always Block Style with combinatorial outputs, (3) 3-Always Block Style with registered outputs, and (4) 4-Always Block Style with registered outputs

This paper establishes measurement techniques to determine good coding styles and also shows synthesis results for ASIC designs.

Multiple benchmark FSM designs are used to measure coding style and synthesis efficiency.

The other three FSM design styles will be described in follow-on papers.

Table of Contents

Table of Figures

Table of Examples

Table of Tables

1. Introduction

What could possibly be new about Finite State Machine (FSM) design? Hasn't this topic been completely covered by existing publications?

Sunburst Design has been teaching six different FSM coding styles, plus a few minor variants, for more than two decades and the last paper that Cliff did on this topic was presented in 2003. Cliff & Heath have made many new observations and refinements to the FSM coding techniques and we have observed synthesis improvements using FSM coding styles. Also, FSM synthesis can have interesting differences when synthesizing for ASICs and FPGAs. This paper shares fundamental FSM coding styles along with newer FSM design techniques that we have refined in the 15 years since Cliff's last FSM conference paper.

FSM design is actually a very large topic. The seven major FSM coding styles include one FSM coding style with combinatorial outputs and six FSM coding styles with registered outputs. Registering module outputs is typically recommended by synthesis tool vendors as it helps meet timing goals more easily without using multiple different input and output timing design constraints. Registered outputs are also glitch-free. There is nothing inherently wrong with FSM combinatorial outputs when glitching outputs are used internally to an ASIC or FPGA design and settle before the next active clock edge to meet register setup times, which can be proven with Static Timing Analysis (STA) tools, so we do show one reasonable FSM coding style with combinatorial outputs.

Heath and Cliff generally recommend doing FSM design with registered outputs and there are multiple FSM coding styles that achieve that goal.

The seven different coding styles that we commonly teach are:

- 2-always block coding style with combinatorial outputs.
- 1-always block coding style with registered outputs.
- 3-always block coding style with registered outputs.
- 4-always block coding style with registered outputs (new style not previously shown).
- Indexed OneHot coding style with registered outputs.
- Parameter OneHot coding style with registered outputs.
- Output encoded coding style with registered outputs.

This topic is large enough that it is our intent to break documentation on these styles into three papers with this being the first. This paper will cover the first four coding styles listed above with the remaining three coding styles and sub-variations to be covered in subsequent papers.

In the process of writing this paper, we discovered that the 1-always block coding style typically infers a design that is somewhat smaller and faster than our preferred 3-always block coding style. We found that creating a 4-always block FSM design could achieve the same synthesis results as the 1-always block style while still being a much more concise coding style than the 1-always block style. The reasons for these claims and the 4-always block coding style are included in this paper.

2. Important design goals

Harry Foster of Mentor, A Siemens Business, has conducted industry trend studies using the Wilson Research Group, which we believe are the best and most reliable studies in our industry. These studies have shown design and verification trends for ASIC & FPGA design since the year 2010. In the year 2010, Cliff Cummings and Harry Foster conducted Assertion Based Verification seminars worldwide and in those seminars, Harry consistently claimed that the activity that was most responsible for putting a project behind schedule was debug time.

Activity	2010	2012	2014-FPGA	2014-ASIC	2016-FPGA	2016-ASIC	2018-FPGA
Debugging	37%	36%	43%	37%	43%	39%	42%
Creating & Running Test	27%	23%	19%	24%	21%	22%	19%
Testbench Development	28%	22%	20%	22%	20%	22%	20%
Other	14%	20%	17%	17%	16%	17%	18%

Due to rounding, numbers do not always sum to 100%

Source: Harry Foster / Wilson Research Group Studies

Since 2014, Debugging has taken ~95% more effort than any other single activity

Table 1 - Average time verification engineers spend in various tasks

The Wilson Research Group Studies from 2010[6], 2012[7], 2014[8][9], 2016[10][11] and 2018[12] all showed that debug time consistently consumed the most verification engineering time. Since 2014 as shown in Table 1, debugging has taken on average ~95% more effort than any other verification task.

It is clear from these studies that any coding habit that helps to minimize debug time is an important habit to develop. The RTL coding guidelines shared in this paper help to reduce debug time.

2.1 More code = more bugs

It has often been said that more lines of code equals more bugs. A quick online search could not find definitive sources for this claim but in all of the sources referenced, none made the counter claim and many sources gave reasoned arguments why the claim that more_lines = more_bugs makes sense [19][20][21][22][23][24][25][26].

For these reasons, we believe that ***concise coding styles*** that follow ***defensive coding guidelines to either avoid bugs*** or to enable early detection are of greatest value in RTL design in general and FSM design in particular. Throughout this paper we will emphasize where we have used concise coding styles and styles that help to easily identify bugs.

3. FSM Coding Style Metrics

In order to judge what makes a good coding style, we selected the following goals that can be used to judge the various coding styles:

1. The FSM coding style should be *easily modifiable to change state encodings and FSM styles.*
2. The coding style should be *concise.*
3. The coding style should be *easy to code and understand.*
4. The coding style should help *facilitate debugging*
5. The coding style should *yield efficient synthesis results*
6. The coding style should be *easy to change due to FSM modifications* including modifying the number of inputs and outputs.

These metrics will be evaluated for each coding style and tabulated at the end of this paper.

4. State machine types

There are two principle state machine classifications that apply to all state machine designs.

4.1 Moore -vs- Mealy

A Moore state machine is classified as an FSM where the outputs are only a function of the present state, while Mealy state machines have one or more outputs that are a function of the present state and one or more FSM inputs.

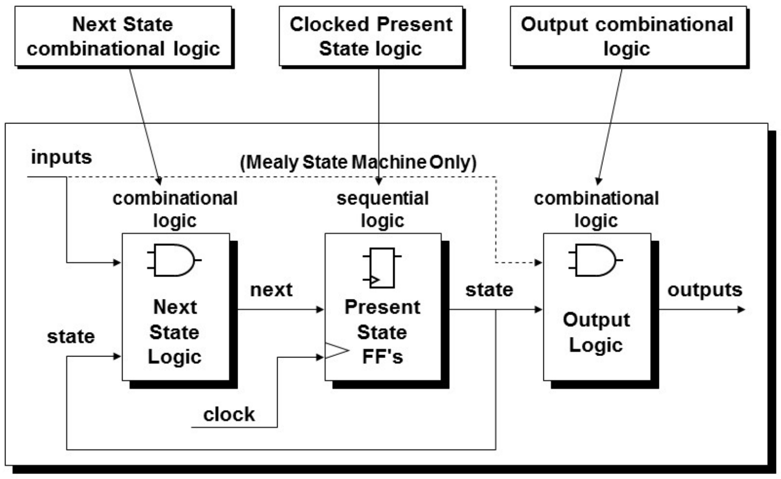


Figure 1 - Moore & Mealy State Machine block diagram

Moore state machines are favored in industry because the outputs have a full cycle to settle through the combinatorial logic and are therefore easier to meet required cycle times. Mealy outputs allow an input to appear after the cycle has started, and the input must still traverse the combinatorial logic and meet setup time for the Mealy output. If the design absolutely requires an input to make it on-chip after the active clock edge, pass through logic and appear on the output all within one cycle, those are the designs that typically use a Mealy output.

Using a play on words, it is said that, "Moore is Less" meaning that Moore state machines are only dependent on the current state while Mealy State Machines are dependent on the current state and one or more inputs.

In general, we avoid Mealy FSM designs unless absolutely necessary.

4.2 Binary -vs- OneHot encoding

The second major classification is whether the state encoding of the FSM designs are binary (also referred to as highly encoded) or if they are OneHot.

There are multiple binary encoded styles but what they have in common is that they typically use fewer flip-flops to create unique binary encodings for each state in the FSM design than OneHot FSM designs.

OneHot encoding uses one flip-flop for each state in the state machine and when the state machine is in that state, that flip-flop is "hot" or equal to "1" while the rest of the states are equal to "0." Since transitioning from one state to another only requires that the previous hot flip-flop be set to 0 and the new hot flip-flop be set to 1, the combinatorial logic to transition between states is typically very simple. As will be explained in Section 5. OneHot FSM designs are typically inferred automatically by FPGA synthesis tools.

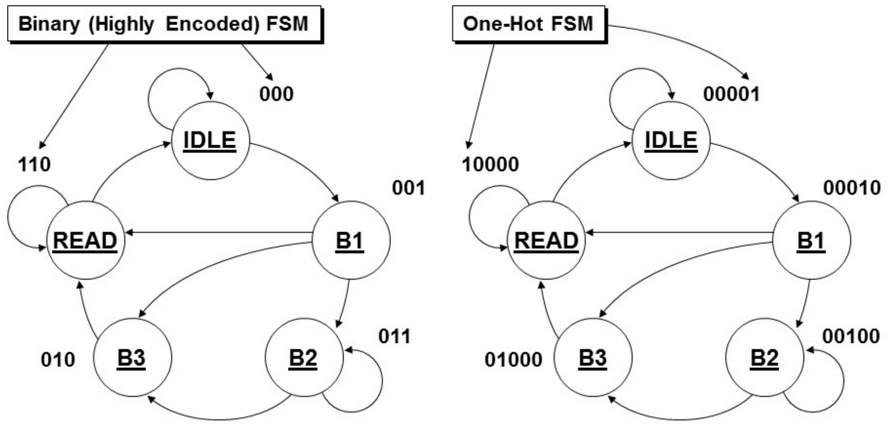


Figure 2 - Binary -vs- OneHot state diagrams

5. ASIC -vs- FPGA Synthesis

There are two important differences between ASIC & FPGA design.

For ASICs, the general rule is that "silicon is free and flip-flops cost." Adding another nand gate to a combinatorial path typically does not add much delay to the path, but every flip-flop adds blocks of clocked logic, which do consume space.

For FPGAs, the general rule is that "flip-flops are free and combinatorial logic costs." FPGAs typically have more flip-flops than are needed for most designs so using the extra flip-flops does not add more silicon to the design, while the speed of an FPGA is largely determined by the size of the combinatorial logic. If the combinatorial logic fits into one Look Up Table (LUT) the logic is fast. If the combinatorial logic requires two LUTs to implement the combinatorial function, then there is one lookup delay, one or more trace delays, and a second lookup delay, which typically causes the combinatorial logic delay to be more than twice as long as a one-LUT combinatorial implementation.

Since OneHot FSM designs typically require smaller combinatorial logic (as discussed in Section 4.2), FPGA synthesis tools synthesize OneHot State Machines for smaller FSM designs by default, regardless of the state encoding included in the RTL code. For this reason, FPGA synthesis tools are sometimes more forgiving when it comes to optimizing synthesis coding styles. As long as the FPGA synthesis tool recognizes that the RTL code will infer an FSM, the synthesis tool creates a very good OneHot FSM design and largely ignores the coding style used by the RTL designer.

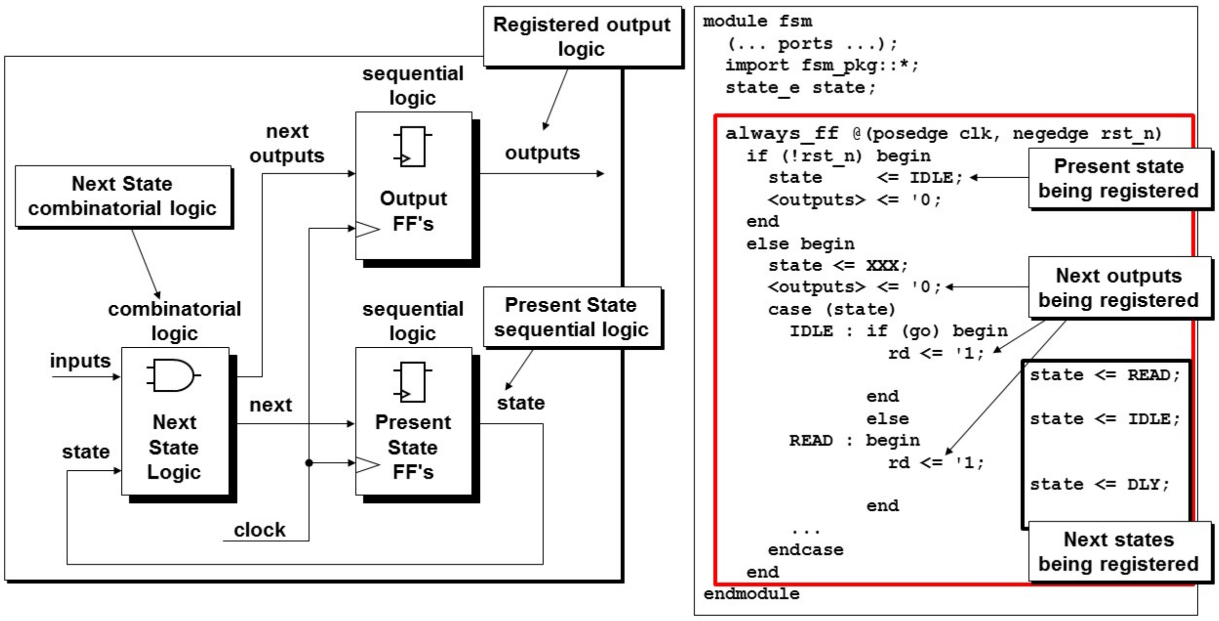
FPGA synthesis tools frequently have a special setting (typically a GUI switch) to turn off the automatic creation of OneHot FSM designs. OneHot FPGA designs will be described in the Part II FSM paper.

6. SystemVerilog FSM coding styles

The four FSM coding styles discussed in this paper are the 1-always block with registered outputs, 2-always block with combinatorial outputs, 3-always block with registered outputs and 4-always block with registered outputs. OneHot and Output Encoded FSM coding styles will be discussed in subsequent publications.

6.1 One Always Block FSM coding style - registered outputs

The 1-always block FSM coding style can be viewed as a single `always_ff` procedure (shown in Figure 3) that handles the state register, the next `state` assignments and the outputs that belong to the next `state` for each transition arc. The trick used in this coding style is to recognize that the output assignments are for the state you are going to next and not for the current state being tested in the `case` statement. The 1-always block coding style requires designers to set the outputs for each transition arc to that `state` and not just once for that `state`. This is why this coding style is so verbose. On the positive side, the synthesis results are typically better than the 2-always block and 3-always block coding styles since the next `state` and next-outputs are being generated in parallel in the single `always_ff` procedure. The fix for the 3-always block synthesis inefficiency is to split the final `always_ff` into separate `always_comb` for next-outputs and to register those outputs in a final `always_ff` procedure, thus creating the 4-always block coding style as shown in section 6.4 .

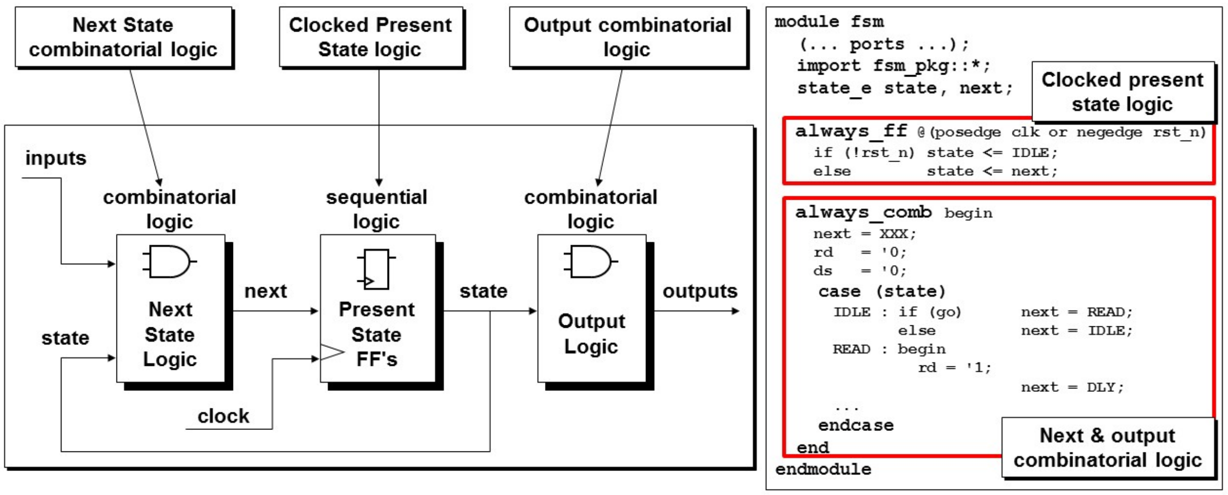


```
module fsm
  (... ports ...);
  import fsm_pkg::*;
  state_e state;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      state     <= IDLE;
      <outputs> <= '0;
    end
    else begin
      state <= XXX;
      <outputs> <= '0;
      case (state)
        IDLE : if (go) begin
                 rd <= '1;         state <= READ;
               end
               else                state <= IDLE;
        READ : begin
                 rd <= '1;
                                   state <= DLY;
               end
        ...
      endcase
    end
endmodule
```

Figure 3 - Block diagram for 1-always block coding style

6.2 Two Always Block FSM coding style - combinatorial outputs

The 2-always block FSM coding style can be viewed as an `always_ff` state register (just 3 lines of code) followed by an **always_comb** procedure to represent the combined **next** state logic and combinatorial output logic. Optionally, the outputs can be separated from the **always_comb** procedure and placed into a separate **always_comb** procedure or into one or more continuous assignment statements.

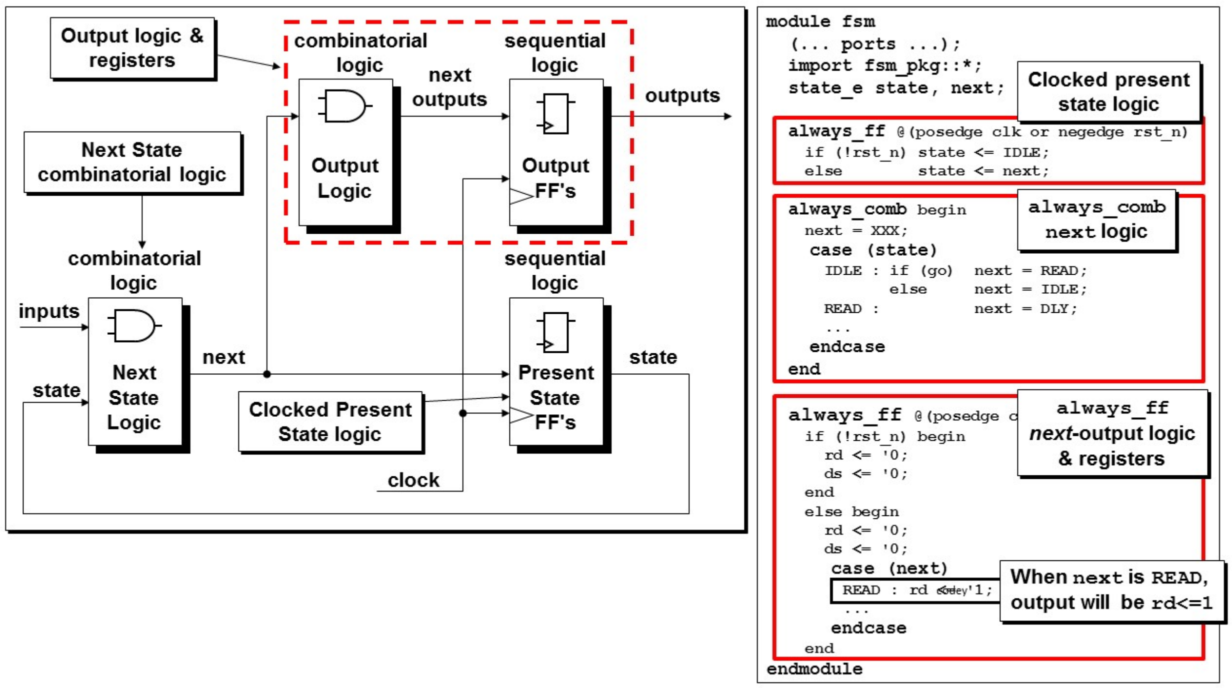

Next State combinatorial logic
Clocked Present State logic
Output combinatorial logic
inputs
combinatorial logic
sequential logic
combinatorial logic
next
state
outputs
Next State Logic
Present State FF's
Output Logic
state
clock
Clocked present state logic
Next & output combinatorial logic

```
module fsm
  (... ports ...);
  import fsm_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;
    rd   = '0;
    ds   = '0;
     case (state)
      IDLE : if (go)      next = READ;
             else         next = IDLE;
      READ : begin
               rd = '1;
                          next = DLY;
      ...
     endcase
  end
endmodule
```

Figure 4 - Block diagram for 2-always block coding style

6.3 Three Always Block FSM coding style - registered outputs

The 3-always block FSM coding style can be viewed as an **always_ff** state register (just 3 lines of code) followed by an **always_comb** procedure to represent the **next** state combinatorial logic and an **always_ff** procedure to calculate and register the next outputs.



```
module fsm
  (... ports ...);
  import fsm_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;
     case (state)
      IDLE : if (go)   next = READ;
             else      next = IDLE;
      READ :           next = DLY;
      ...
     endcase
  end

  always_ff @(posedge c
    if (!rst_n) begin
      rd <= '0;
      ds <= '0;
    end
    else begin
      rd <= '0;
      ds <= '0;
       case (next)
        READ : rd <='1;
        ...
       endcase
    end
endmodule
```

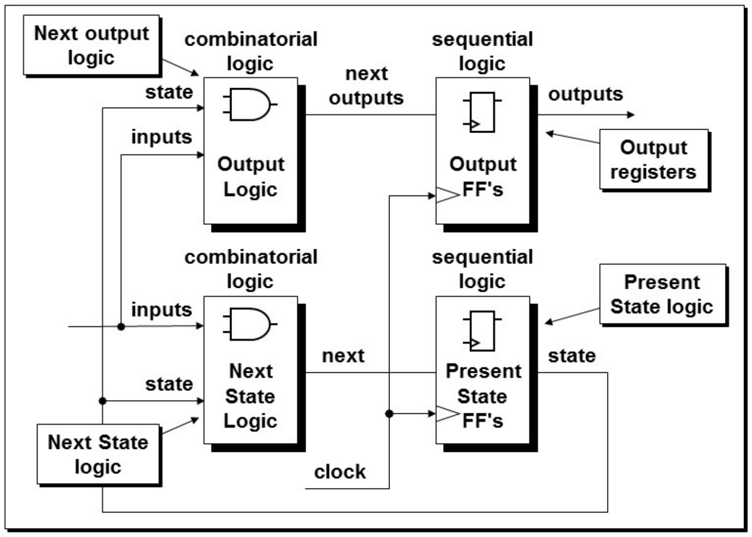
Figure 5 - Block diagram for 3-always block coding style

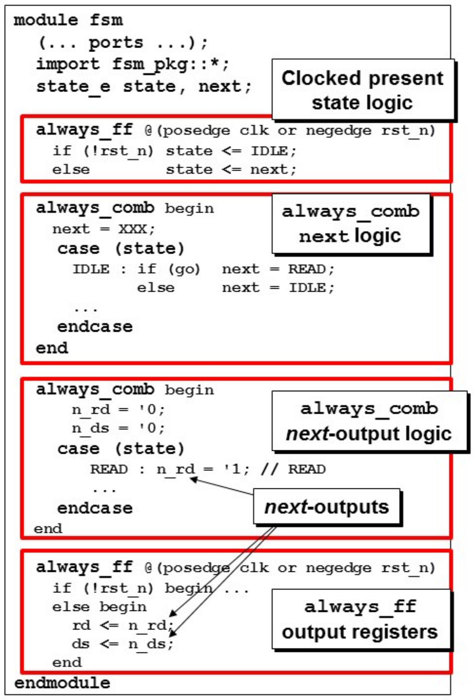
The trick used in this coding style is to recognize that the output assignments are the "next" output assignments and not the output assignments for the current state, so the final `always_ff` procedure will use a `case` statement to test the `next` state, not the current state. Since the outputs are calculated from the `next` state logic, this can add extra logic when synthesized as the `next` state was already calculated from input conditions. This means that there is one block of combinatorial logic that calculates the `next` state and that feeds a second block of combinatorial logic to calculate the next outputs, which can create larger and slower combinatorial logic. In the 1-always block coding style, the inputs and `state` variables are used to calculate both the `next` state and next outputs in parallel, which can reduce the size and delay through the combinatorial next output logic. This inefficiency will be addressed in the 4-always block coding style described in the next section.

6.4 Four Always Block FSM coding style - registered outputs

While writing this paper, we discovered that the 1-always block coding style required much more code but typically gave better synthesis results than the 3-always block coding style. This was a bit of a surprise until we realized that the next-output combinatorial logic of the 3-always block coding style is fed by another block of combinatorial logic that was used to calculate the `next` state. The 1-always block coding style generated the `next` state and next-outputs in parallel, in the same combinatorial block of logic.

We found that creating a 4-always block coding style where the combinatorial nextout values are created from an `always_comb` procedure that simultaneously examines the registered state outputs along with the FSM inputs, would generally give the same optimized synthesis results as the 1-always block coding style with far fewer lines of code. The first `always_ff` and `always_comb` procedures are identical to the 3-always block coding style. The third `always_ff` procedure from the 3-always block coding style is split into an `always_comb` procedure to calculate the next-output logic and an `always_ff` procedure to register the next-outputs. These optimizations improved synthesis results to match the results observed with the 1-always block style.





```
module fsm
  (... ports ...);
  import fsm_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;
     case (state)
       IDLE : if (go)  next = READ;
              else     next = IDLE;
       ...
     endcase
  end

  always_comb begin
      n_rd = '0;
      n_ds = '0;
     case (state)
        READ : n_rd = '1; // READ
        ...
     endcase
  end

  always_ff @(posedge clk or negedge rst_n)
    if (!rst_n) begin ...
    else begin
      rd <= n_rd;
      ds <= n_ds;
    end
endmodule
```

Clocked present state logic
always_comb next logic
always_comb *next*-output logic
next-outputs
always_ff output registers

Figure 6 - Block diagram for 4-always block coding style

7. SystemVerilog FSM Coding Tips & Tricks

As of the year 2019 and after 15 years of SystemVerilog FSM coding, we have compiled some of our favorite FSM coding tips and tricks. You do not have to follow any of these tips and tricks to implement a working and synthesis-efficient FSM design, but we have found them to be useful and advantages of using each trick will be explained. Users are encouraged to implement their favorite tricks from this list and feel free to share back with us some of your favorite tricks.

Many of these tricks fall into the category of "less is better." We find it easier to visualize a design that is concise and where more of the design can be viewed without additional scrolling on a computer screen or flipping of printed pages. For us, a well formatted and concise design earns extra points.

To help explain the tips & tricks, we will use the following very simple Moore state machine with four states, two inputs and two outputs.

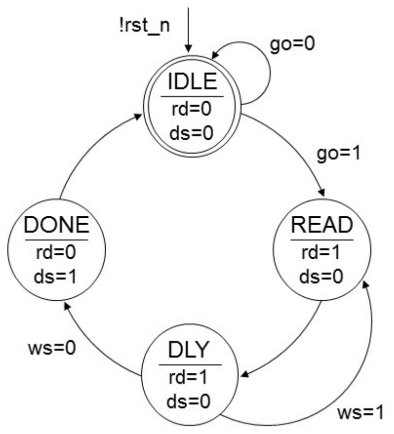


Figure 7 - FSM1 state diagram - used for example code

7.1 Logic data type

For SystemVerilog RTL coding, designers should follow this guideline:

> *Declare all data types to be of type* `logic` *unless there are multiple drivers on the signal, then use the* `wire` *data type.*

The three most common sources for valid multi-driver-signal designs are bus crossbars, OneHot multiplexers and bidirectional buses, all of which are becoming increasingly scarce in modern hardware design. Any logic that includes three-state logic is strongly discouraged in contemporary ASIC designs and almost non-existent in modern FPGA designs.

The strong advantage of the `logic` data type is that if a designer ever mistakenly makes multiple assignments to the same signal, the simulation *compiler* will catch and report the error before an engineer ever simulates or synthesizes the design. Catching the error before simulation saves debug time.

Checklist item: Engineers should generally declare all FSM ports and all FSM internal signals to be of type `logic`.

7.2 Assignments using '0 / '1 / 'x -vs- 1'b0 / 1'b1 / 1'bx

SystemVerilog introduced a very convenient shorthand to assign all 0's, all 1's or all X's to a scalar signal or a vector bus. The notation is to assign `'0` (all zeros), `'1` (all ones) and `'x` (all X's). Although this is a small enhancement, it is useful for multiple reasons.

First, assigning `'0` instead of `1'b0` is more concise, albeit by only two characters so this is not the compelling reason to use this style.

A better reason is that the notation `1'b0` is somewhat confusing and when the code is quickly scanned it is easy to mistake this assignment for a "1" while the shorter `'0` conveys no such confusion.

So why not just use `0` or `1` to make an assignment? Assigning `0` or `1` is acceptable and common in Verilog designs and they typically work just fine, but assigning `0` is really assigning 32-bits of `0` while assigning `1` is really assigning 31-bits of `0` with LSB equal to `1` and Verilog quietly truncates the leading 31 bits when this is assigned to a 1-bit variable. This is fine unless the code is run through a linting tool, which frequently reports a warning that 32 bits are being assigned to a 1-bit variable. The assignment works fine but the warning is both annoying and distracting so engineers often add waivers to their linting tools to ignore this warning.

Coding `'0` / `'1` / `'x` assigns as many 0's, 1's or X's as are required to fill the left-hand-side (LHS) variable or wire that is being assigned, so there is never any linting tool warning related to mismatched sizes when making these new SystemVerilog assignments.

Also when assigning X's using the older Verilog style, one must indicate how many X-bits are required for the assignment while `'x` will fill the assigned variable with as many X's as are needed. This avoids bugs that might be introduced when widths are changed later in the design.

We use `'0` / `'1` / `'x` for all of our FSM designs, and for that matter, for all of our SystemVerilog designs that do not require a non-repeated fill pattern. For more unusual assignment patterns, we still use the older Verilog style to indicate the bit-pattern and how many bits are required, for example `8'b0110_1100` or `8'h6C`.

Checklist item: Where ever possible, use the SystemVerilog `'0` / `'1` / `'x` to make assignments.

7.2.1 Caution using '0 / '1 / 'x

Although we recommend using these new assignment styles, there are two points of caution that engineers should understand.

First, assigning `'1` will assign all 1's and not all 0's followed by a single 1. If you assign `data[7:0]='1` then data will be equal to `8'hFF`, not `8'h01`. `'1` does indeed assign all 1's.

Second, when `'0` / `'1` / `'x` are placed inside of concatenation, they are only 1-bit assignments and they do not fill the remaining bits with the pattern. If you assign `data[7:0]={'1, 4'hA}` the result will be that data is equal to `8'h1A` and NOT `8'hFA`. The reason that the `'1` does not fill the leading bits is because what if multiple `'1` / `'0` assignments are included in the same concatenation? Where is the fill operation supposed to occur? Consider the following examples:

```
data[15:0] = {'1, 4'hA, '0};
// Which bits should be leading 1's and which bits should be trailing 0's?
data[15:0] = {'0, '1, '0};
// Which bits should be leading 1's, middle 0's and trailing 1's?
```

There were too many opportunities for confusion by adding `'0` / `'1` / `'x` / `'z` to concatenation, so the SystemVerilog Standards Group decided that when this shorthand notation was used in concatenation that the notation would only expand to a single bit each. In the above examples, the values expand as shown below.

```
data[15:0] = {'1, 4'hA, '0};
// is equal to 16'b0000_0000_0011_0100 or 16'h0034
data[15:0] = {'0, '1, '0};
// is equal to 16'b0000_0000_0000_0010 or 16'h0002
```

7.3 FSM module header & port list

The FSM module header and port list should be coded using the Verilog-2001 concise port list style. All ports should be declared to be of type `logic` for the reasons discussed in Section 7.1 We order the port list as outputs followed by inputs followed by control inputs. The outputs-inputs order is not required by Verilog or SystemVerilog and many engineers prefer to list inputs followed by outputs. Since Verilog gate primitives can only be instantiated using positional ports and since the order is always outputs followed by inputs followed by control inputs, we follow that same convention with our own modules. We have also been told by engineering colleagues that most module changes are related to adding or modifying the inputs and that it is easier to modify the end of the list as opposed to the front or center of the list.

The module header and port list for the FSM1 design is shown in Example 1.

```
module fsm1_3 (
  output logic rd, ds,
  input  logic go, ws, clk, rst_n);
```

Example 1 - FSM1 module header and port list example

Many engineers list each port on a separate line and there is nothing specifically wrong with that practice, but we prefer to group multiple signals into each `output` and `input` declaration as shown in Example 1, to reduce the number of lines of code in the FSM design and to allow more of the FSM design to be visible per page when examining the code.

Checklist item: Use the Verilog-2001 concise port declaration style and declare all ports to be of type `logic`. Extra points for listing outputs followed by inputs followed by control inputs. Extra points for grouping multiple signals into each `output` and `input` port list declaration.

7.4 Enumerated state types

Enumerated types are used to make the `state` and `next` state declarations and then the enumerated state names are used throughout the FSM design.

While working on this paper, we discovered that there were multiple advantages to putting the enumerated declarations into a package as `typedef`[s] and then importing the package into the FSM design.

We used the naming convention of `<fsm-name>_pkg` and we actually maintained two different FSM packages, one with abstract enumerated declaration `typedef` with filename `<fsm-name>_pkg_a.sv` (`_a` for abstract) and one with binary encoded enumerated declaration `typedef` with filename `<fsm-name>_pkg_b.sv` (`_b` for binary).

Using the FSM `enum typedef` packages offered the following advantages:

(1) Each FSM coding style would import the same FSM packages and ensure that all the FSM styles used the exact same FSM `typedef`[s]. Fewer opportunities to make mistakes.

(2) By maintaining `pkg_a` and `pkg_b` files, we could easily switch from abstract to binary encoded `enum` `typedef`[s] without maintaining two copies of each FSM file.
(3) Simulation of either abstract or binary-encoded `enum`[s] was as easy as selecting and compiling the appropriate package before compiling the other simulation files since each package had the same identical package name. This gave us much easier code maintenance.
(4) Similarly, synthesis of either abstract or binary-encoded `enum`[s] was as easy as compiling the appropriate package before compiling the other synthesis files since each package had the same identical package name. Again, much easier code maintenance.
(5) If we wanted to try different enumerated assignments, we only had to change the `pkg_b` file and all styles would use the exact same new encodings. We did not have to touch the FSM files themselves.
(6) The testbench could easily use the same `state_e typedef` to implement reference models to track expected FSM states, if desired.
(7) The testbench could easily use `bind` -files for assertions or covergroups by using the `typedef` to input the `state` and `next` variables from the FSM design.

After we implemented the separate package files, we were able to cut the number of FSM benchmark files in half and ensured a more error-free coding style.

The enumerated declarations for the `state_e typedef` can either be made using abstract state encodings as shown in

Example 2, or can be made using the `logic` data type, specifying the number of state bits as a range, and include user-defined encodings for each state as shown in Example 3.

```
package fsm1_pkg;
  typedef enum {IDLE,
                READ,
                DLY,
                DONE,
                XXX } state_e;
endpackage
```

Example 2 - FSM1: fsm1_pkg_a.sv with abstract enumerated state_e typedef

```
package fsm1_pkg;
  typedef enum logic [1:0] {IDLE = 2'b00,
                            READ = 2'b01,
                            DLY  = 2'b11,
                            DONE = 2'b10,
                            XXX  = 'x   } state_e;
endpackage
```

Example 3 - FSM1: fs1_pkg_b.sv encoded state_e typedef w/logic data type and bit-range

```
  import fsm1_pkg::*;
  state_e state, next;
```

Example 4 - FSM1: importing fsm1_pkg::*; (abstract or binary) with state/next declaration

It is typical for RTL coders to start with the abstract `state` and `next` declarations and to later add state encodings if desired. Since the state names are used throughout the FSM RTL code, adding state encodings later does not require any additional RTL modifications to the FSM design.

The enumerated state names also show up automatically in a waveform display to help track state transitions and to help debug the state machine. It is often useful to show two copies of the enumerated names in the waveform display, the first showing the state names (which is the default mode for all waveform viewers) and the second copy with the radix changed to be binary, decimal or hex to see the state encodings (often useful for debugging). There is nothing in the IEEE SystemVerilog Standard[15] that requires a waveform viewer to show the state names by default, but all waveform viewers that we have used from all of the major EDA vendors show the state names by default.

It is a good practice to include an extra `XX` or `XXX` state in the list of declared state names as shown in both Example 2 and Example 3. This extra `XXX` state will be used to code the `always_comb` procedure and this practice can help quickly debug an FSM design, plus if one chooses to include state encodings, the `XXX` state will be set to all X's, which are treated as "don't cares" by the synthesis tool to help optimize the synthesized design.

Checklist item: Declare the `state` and `next` variables using enumerated types. Add an `XX` or `XXX` state to help debug the design and to help optimize the synthesized result.

7.5 FMS RTL code should use SystemVerilog always_ff and always_comb procedures

The RTL code for the FSM design should use `always_ff` and `always_comb` procedures to infer clocked and combinatorial logic. Do not use the older Verilog `always` procedures. The `always_ff` and `always_comb` procedures show designer intent and include built-in checking for bad RTL coding styles as described in the paper "SystemVerilog Logic Specific Processes for Synthesis - Benefits and Proper Usage[3]."

Checklist item: Use `always_ff` and `always_comb` procedures to infer clocked and combinatorial logic. Do not use the older Verilog `always` procedures.

7.6 FSM State Register

The state register should be an `always_ff` procedure and should be 3 lines of code, not 5 lines and not 6 lines. This state register is placed at the top of the FSM design, immediately following the FSM declarations, and requires no comment to note the obvious state register code. Since the state register code is at the top of the module, there is no need to search the remainder of the FSM design to see that the state register code was added and implemented properly.

With a `posedge clk` and a low-true asynchronous reset, our state registers are almost always exactly the following three lines of code:

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

Example 5 - FSM state register declaration

There is no need to put the `if`-statement and the `state` assignment on separate lines and there is no need to put `begin` - `end` on the `always_ff` procedure. Also, we place the reset state assignment and `next` state assignment into a nice, neat column for easy scanning and reading.

Note that adding `begin` - `end` to the `always_ff` procedure is not only unnecessary, but introduces the opportunity to add code before the first `if (!rst_n)` statement, which violates coding requirements for some synthesis tools including Design Compiler. In short, the `always_ff begin` - `end` allows an RTL designer to code and simulate something that might not be synthesizable and as such, we consider this to be a poor coding practice.

Consider the legal RTL code in Example 6.

```
always_ff @(posedge clk, negedge rst_n) begin
  testbad <= go;
  if (!rst_n) state <= IDLE;
  else        state <= next;
end
```

Example 6 - Bad usage of always_ff begin-end statements

This example code is legal SystemVerilog code and will simulate without error, but once this design is read into the DesignVision GUI, the following error messages are reported in a pop-up window:

```
1: Error:
<filename><line number>:
The statements in this 'always' block are outside the scope of the
synthesis policy. Only an 'if' statement is allowed at the top level in
this always block. (ELAB-302)

2: Error: Cant' read 'sverilog' file
<filename>
(UID-59)
```

Figure 8 - DesignVision error messages from bad usage of always_ff begin-end statements

In short, the `begin` - `end` is unnecessary, needlessly verbose and introduces the opportunity to add bad code that might not be discovered until the design is read for synthesis.

Checklist item: Declare the state register using just 3 lines of code and place it at the top of the design after the enumerated type declaration.

7.7 Default next='x -vs- default next=state

There are two common ways to code the `always_comb` procedure for an FSM design. The two methods are to either make a pre-default-X `next` assignment or use a pre-default `next=state` assignment[16] with implied loopback assignments. While running synthesis benchmarks on both styles, and to our surprise, we discovered that the `next='x` pre-default-X assignment consistently gave better synthesis results over the pre-default `next=state` assignment style. The reason seems to be that the pre-default-X tends to fill out a synthesis-equivalent of the Karnaugh Map (K-Map) with X's, especially when there are fewer than 2^n states in the FSM design. Even when adding case-default-X assignments to both styles, the pre-default-X assignment style consistently gave better synthesis results.

```
always_comb begin
  next = XXX;                     // Pre-default-X assignment
  case (state)
    IDLE : if (go)   next = READ;
           else      next = IDLE; //@ loopback
    READ :           next = DLY;
    DLY  : if (!ws)  next = DONE;
           else      next = READ;
```

```
      DONE :          next = IDLE;
      default:        next = XXX;  // case-default-X assignment
    endcase
  end
```

Example 7 - Default next='x combinatorial always_comb procedure for FSM designs

Anyone who has done extensive RTL coding and simulation recognizes that, when it comes to RTL debugging, catastrophic RTL coding failures are the easiest to find, identify and fix. Conversely, subtle RTL coding errors can often be very time consuming to find and frequently require the RTL coder to add numerous debug printing messages to finally discover the subtle problem in the code. Making X-assignments can help cause catastrophic simulation failures when there is an RTL design problem, while implied loopback errors can often allow an FSM design to appear to be working for multiple clock cycles before the simulation reports an FSM simulation error.

So how does this work and what procedure do we recommend?

We strongly encourage engineers to make the pre-default `next='x` assignment in the FSM design. The pre-default X-assignments have two RTL simulation debug advantages and one RTL synthesis advantage.

The RTL simulation advantage is that assigning X's frequently causes the simulation to fail catastrophically if there is a missing `next` assignment. At the point where there is a missing `next` assignment, we say that the waveform display "starts to bleed red!" Wherever the `next` state assignment is missing, the waveform display will show that `next` is all X's and that is typically the exact point where the `next`-assignment is missing. This is a catastrophic simulation error that quickly identifies the missing assignment and, from our experience, is fixed very quickly.

The second RTL simulation advantage is that making a pre-default-X assignment requires the RTL coder to code an equation for each transition arc *from* each state. After completing the FSM design, a designer can double-check the code by counting the number of transition equations for each state in the RTL code and match that to the number of transition arcs in the state diagram. There is a one-to-one correspondence between the number of transition equations and transition arcs. Using the `next=state` default assignment allows designers to remove loopback transition assignments from the RTL code wherever there is a feedback loop on that state so there will be fewer RTL equations than transition arcs in the state diagram.

The RTL synthesis advantage is that making X-assignments is like putting X's into a K-Map for unused states and all digital design engineers recognize that adding X's to a K-Map allows engineers or tools to make larger groupings, enabling smaller sums-of-products, which can infer smaller combinatorial logic (at least in theory).

Making a `next=state` pre-default assignment only has one minor advantage over the `next='x` pre-default assignment. Any state with a loopback transition can be omitted from the RTL code since that state will not take another transition branch and will remain in the same state; thus potentially removing a few lines of code.

Since debug time is a primary concern in RTL design, we recommend using the pre-default `next='x` assignment style.

Checklist item: Use the pre-default `next='x` assignment at the top of the `always_comb` procedure.

7.8 Next state naming convention

We prefer to use the identifiers `state` and `next`, not `state` and `nextstate`. There is nothing wrong with using `nextstate` except that it is a needlessly long identifier that lengthens the code on all of the `next` assignments in the combinatorial `always_comb` procedure.

We know that `next` is a keyword in VHDL so those with a VHDL background tend to gravitate towards using the `nextstate` identifier, but `next` is not a keyword in SystemVerilog and SystemVerilog coders can make their designs more concise by using the `next` identifier instead of using `nextstate`. As previously mentioned, we prefer concise code.

Checklist item: Extra points for using `next` instead of using `nextstate` in the FSM design, but both work fine.

7.9 Next assignments placed in a column

Since readability and the ability to quickly identify RTL coding errors are very important, we also place all of the `always_comb next` assignments neatly in a column positioned towards the right margin of the RTL code as shown in Example 8. We find it easier to scan the `next` assignments when they are in a neat column as opposed to `next` assignments that follow the contour of the code as shown in Example 9. This style also has the advantage that if an engineer ever mistakenly uses a mixture of blocking and nonblocking assignments in the `next` assignments, is it visually very obvious in the slightly misaligned column of `next` assignments.

```
always_comb begin
  next = XXX;
  case (state)
    IDLE : if (go)  next = READ;
           else     next = IDLE; //@ loopback
    READ :          next = DLY;
    DLY  : if (!ws) next = DONE;
           else     next = READ;
    DONE :          next = IDLE;
    default:        next = XXX;
  endcase
end
```

Example 8 - always_comb next assignments in a neat column - Recommended

```
always_comb begin
  next = XXX;
  case (state)
    IDLE : if (go) next = READ;
           else next = IDLE; //@ loopback
    READ : next = DLY;
    DLY  : if (!ws) next = DONE;
           else next = READ;
    DONE : next = IDLE;
    default: next = XXX;
  endcase
end
```

Example 9 - always_comb next assignments following contour of the code - NOT Recommended

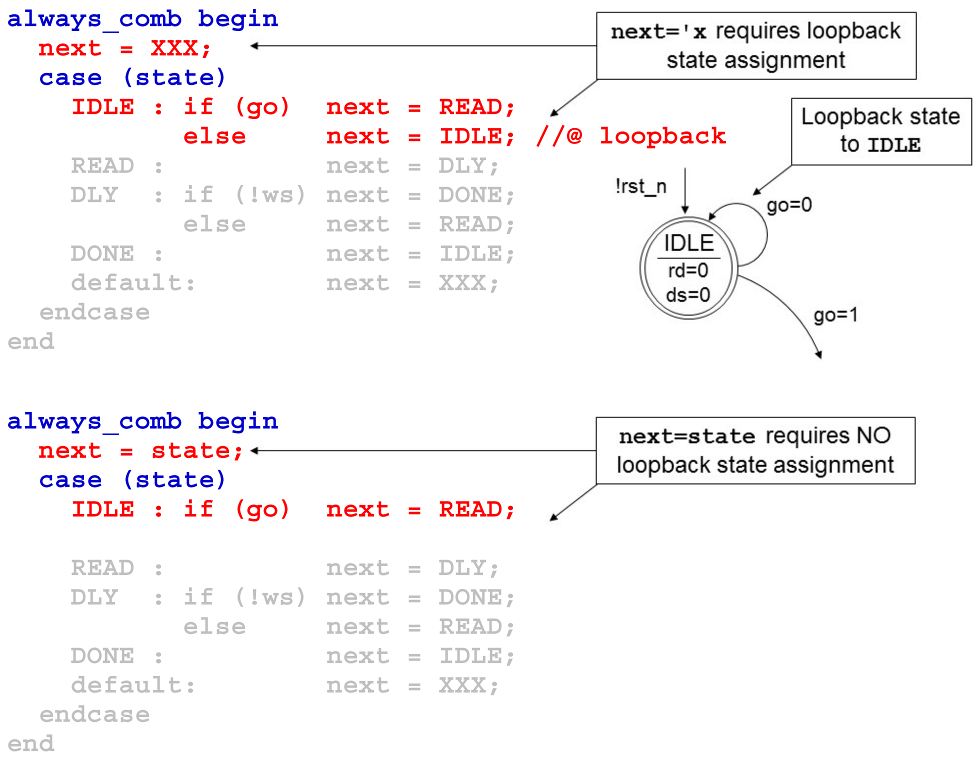
Of course the simulation and synthesis of both of the above examples will be identical. The column alignment is used to quickly understand and debug the `always_comb next` assignments.

Checklist item: Extra points for placing the `next` assignments in a neat column in the FSM RTL code.

7.10 Loopback next state assignments

Any state in the state diagram with a loopback state must be coded into the `always_comb` portion of the design. There are two ways to handle the loopback assignments depending on how the default `next` assignment was implemented.

If the pre-default `next='x` was used (which is what we recommend), the loopback assignment must be coded, but we recommend that the loopback be implemented as the `else` -clause of an `if-else-if` statement. We further recommend that a comment be added to the loopback `else`-assignment of the form `//@ loopback`. The reason for this comment-style will be explained shortly.



```
always_comb begin
  next = XXX;
  case (state)
    IDLE : if (go)  next = READ;
           else     next = IDLE; //@ loopback
    READ :          next = DLY;
    DLY  : if (!ws) next = DONE;
           else     next = READ;
    DONE :          next = IDLE;
    default:        next = XXX;
  endcase
end
```

```
always_comb begin
  next = state;
  case (state)
    IDLE : if (go)  next = READ;

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

next='x requires loopback state assignment
Loopback state to IDLE
!rst_n
go=0
IDLE
rd=0
ds=0
go=1
next=state requires NO loopback state assignment

Figure 9 - FSM loopback state assignment styles

If the pre-default `next=state` is used, the loopback assignment can be omitted from the `if-else-if` statement since the `next` state is already assigned to stay in the current state.

The reason for using the `//@ loopback` comment and placing the loopback into the final `else` -clause is that it makes it very easy to convert from the `next='x` pre-default to the `next=state` pre-default style of `always_comb` procedure (although we do not recommend the latter). The RTL coder simply replaces `next='x` with `next=state` and then deletes all lines that have the `//@ loopback` comment.

7.11 Default output assignments

When coding the outputs in an FPGA design, we have observed that some engineers make all of the output assignments for each state of the `case` statement. We discourage this practice for the following reasons:

It is easy to miss an output assignment for one of the states, which means that the missed output must not change for that state and that will infer a latch when synthesized.

Listing all of the outputs for each state causes the FSM RTL design code to grow in size very quickly.

Listing all of the outputs for each state makes it confusing for an engineer to determine which outputs have changed in which states. Engineers often try to examine the output assignments for the preceding and next states in an attempt to try to figure out what has changed in *this* state.

A better coding practice is to place default output assignments before the output-assignment `case` statement and then to update the appropriate outputs for each state where the outputs change as shown in Example 10.

```
    always_ff @(posedge clk, negedge rst_n)
      if (!rst_n) begin
        rd <= '0;
        ds <= '0;
      end
      else begin
        rd <= '0;
        ds <= '0;
        case (next)
          IDLE : ;
          READ : rd <= '1;
          DLY  : rd <= '1;
          DONE : ds <= '1;
          default: {rd,ds} <= 'x;
        endcase
      end
  endmodule
```

Example 10 - FSM registered output assignments using an always_ff procedure

This coding style offers two significant advantages over making all output assignments for each state.

First, the code becomes more concise. Second an engineer can plainly see which outputs change for each state, thus making the code more readable, easier to understand and easier to debug.

This style requires less code (one of our goals) and makes the design easier to debug (also one of our goals).

Checklist item: Place pre-default output assignments before the output-assignment `case` statement and then update the appropriate outputs for each state where the outputs change.

7.12 Nonblocking Assignment race conditions?

The code shown in Example 10 shows default output assignments using nonblocking assignments followed by more nonblocking assignments to the same outputs in the `case` statement. We are often asked if this is a race condition in Verilog and SystemVerilog? The answer is no and the coding style shown is both proper and recommended.

The question arises due to an error in the 1995 Verilog Standard[13]. Section 5.4.1 of the 1995 Verilog Standard correctly states that two nonblocking assignments to the same variable are queued in the order that they are executed such that the last assignment wins (in more official verbiage). In the same 1995 Verilog Standard, Section 9.2.2 includes an example with a mistake where two assignments are sequentially executed and the description states that the "final value ... is indeterminate." That description was wrong.

The 2001 Verilog Standard[14] fixed the example in section 9.2.2 to have parallel execution of the two assignments, which indeed is a race condition; thus, Section 5.4.1 clearly governs the behavior of two sequentially executed nonblocking assignments and last assignment wins.

There is no race condition in Example 10.

8. Benchmark FSM designs

This paper examines coding efforts and Design Compiler synthesis results for four different FSM benchmark designs: FSM1, FSM7, FSM8 and Prep4. Each design includes a state diagram and a description of the RTL code using the four different FSM coding styles, 1-always block with registered outputs, 2-always block with combinatorial outputs, 3-always block with registered outputs and 4-always block with registered outputs. The full RTL code for each example is included in Appendix 2.

For benchmark purposes, we have noted the number of lines of code required to complete each of the four coding styles that incorporated the pre-default-X assignments and included all of the loopback assignments. We believe the pre-default-X with explicit loopback code to be indicative of the coding effort for each FSM design. We also assigned our evaluation for each of the original coding goals for each style as described in Section 3. We have included coding observations for the designs in each of the four FSM sections.

The synthesis results were tested using the 1-always block, 2-always block, 3-always block and 4-always block coding styles. Each of these four coding styles was also tested with slight variations, specifically, each was tested ***without*** using `case-default`-X assignments (No case-default-X), ***with*** `case-default`-X assignments (case-default X) and then used case-default-X ***with*** pre-default-X (required "Explicit loopback" assignments) and ***with*** `next=state` (used "Implicit loopback" assignments).

To see if there have been improvements between a 2015 version and a 2018 version of Design Compiler (DC), the "No Default X" and "Default X" variations were synthesized using both versions (explicit DC versions are noted in 0), but when we saw absolutely no difference in synthesized results, we quit using the 2015 version and synthesized all of the coding variations only using the 2018 version of DC.

Synthesis compilation was done using two available ASIC libraries, the LSI 10K library, which has been included in all of the DC releases, but which is also a rather old library with limited modern capabilities. We also used an SAED32 (Synopsys Armenia Educational Department 32nm) library that is included with a Synopsys BitCoin online example.

Default synthesis was done without any clocking goals and using abstract enumerated types (no assigned state encodings). Then a `create_clock clk -period 0` was executed after compiling the design and before doing `report_area` and `report_timing` to show the default delay (negative slack) through each design. Note that abstract enumerated types are assigned integer values by SystemVerilog starting with the first state listed equal to zero and each successive state incrementing by 1. These in-order binary counts are typically not very efficient when synthesized.

Based on the default delays, a `create_clock` command was issued with a realistic clock period plus all inputs were given 80% of the clock style while all outputs were given 20% of the clock cycle.

The second iteration of each design used semi-intelligent binary encoded values assigned to each state and some effort was made to choose encodings that would give Gray-like transitions as much as possible. Then the designs were again synthesized, first with no clock goal and then with clock input and output design constraints.

None of the designs up to this point used `case-default` values set to all X's (`case-default`-X). In the tables for each FSM design, these are referred to as the "No case-default-X" values.

Since adding default X's to a `case` statement is similar adding X's to a K-Map, default X-values were assigned to next states and outputs in the `case` statements and the previous synthesis exercises were repeated. These are referred to as the "Default X" values in the tables.

The next synthesis iteration was to change the `next = 'x` assignments to `next = state` and to remove all of the loopback code from the `case` statements. The theory being tested is that synthesis might improve if the state does not transition and if the extra code is removed from each design. Then the synthesis runs were executed again. These are referred to as the "Default X - No Loopback" values in the tables.

8.1 FSM1 with 4 states, 2 outputs and 2 inputs

The FSM1 design, shown in Figure 10, has an asynchronous low-true **`rst_n`**, two inputs **`go`** and **`ws`** (wait state), and two outputs **`rd`** (read) and **`ds`** (done strobe).

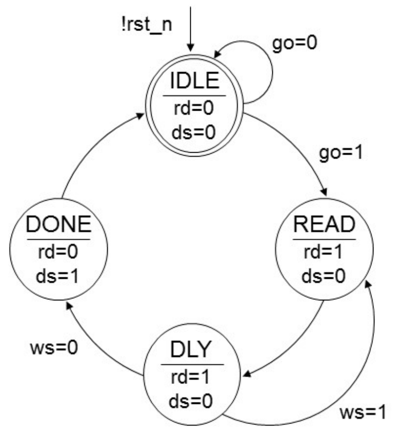
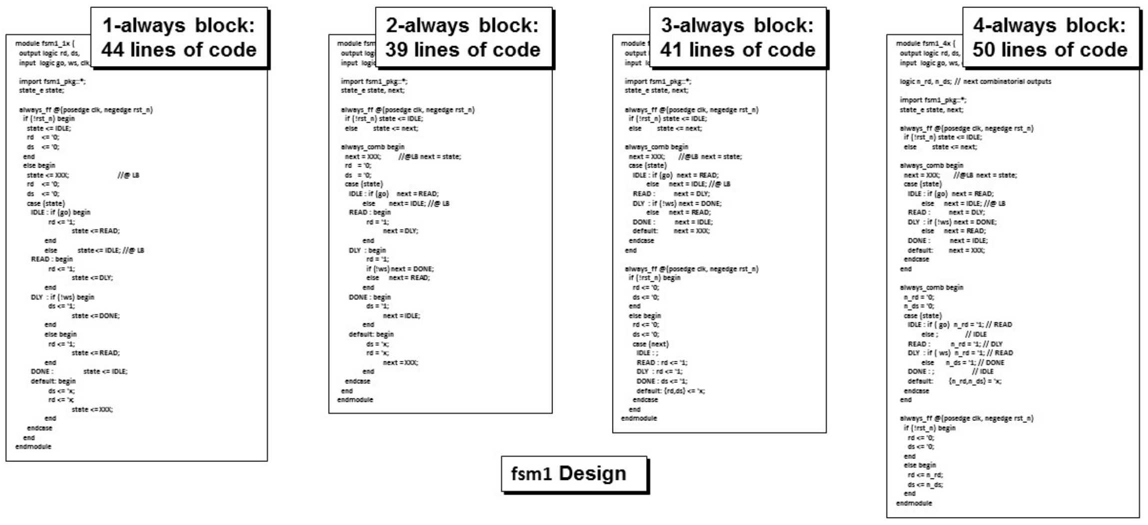


Figure 10 - FSM1 state diagram

There is one loopback state in the FSM1 design and that is in the **`IDLE`** state.

The coding effort for small FSM designs using the different styles does not differ significantly, but it will differ dramatically as the number of states and outputs increases.



```
module fsm1_1x (
  output logic rd, ds,
  input  logic go, ws, clk

  import fsm1_pkg::*;
  state_e state;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      state <= IDLE;
      rd    <= '0;
      ds    <= '0;
    end
    else begin
      state <= XXX;                   //@ LB
      rd    <= '0;
      ds    <= '0;
      case (state)
        IDLE : if (go) begin
                 rd <= '1;
                       state <= READ;
               end
               else        state <= IDLE; //@ LB
        READ : begin
                 rd <= '1;
                       state <= DLY;
               end
        DLY  : if (!ws) begin
                 ds <= '1;
                       state <= DONE;
               end
               else begin
                 rd <= '1;
                       state <= READ;
               end
        DONE :            state <= IDLE;
        default: begin
                 ds <= 'x;
                 rd <= 'x;
                       state <= XXX;
               end
      endcase
    end
endmodule
```

```
module fsm
  output logi
  input  logi

  import fsm1_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;       //@LB next = state;
    rd   = '0;
    ds   = '0;
    case (state)
      IDLE : if (go)    next = READ;
             else       next = IDLE; //@ LB
      READ : begin
               rd = '1;
                    next = DLY;
             end
      DLY  : begin
               rd = '1;
               if (!ws) next = DONE;
               else     next = READ;
             end
      DONE : begin
               ds = '1;
                    next = IDLE;
             end
      default: begin
               ds = 'x;
               rd = 'x;
                    next = XXX;
             end
    endcase
  end
endmodule
```

```
module f
  output
  input  l

  import fsm1_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;      //@LB next = state;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE; //@ LB
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
      default:        next = XXX;
    endcase
  end

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      rd <= '0;
      ds <= '0;
    end
    else begin
      rd <= '0;
      ds <= '0;
      case (next)
        IDLE : ;
        READ : rd <= '1;
        DLY  : rd <= '1;
        DONE : ds <= '1;
        default: {rd,ds} <= 'x;
      endcase
    end
endmodule
```

```
module fsm1_4x (
  output logic rd, ds,
  input  logic go, ws,

  logic n_rd, n_ds; // next combinatorial outputs

  import fsm1_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;      //@LB next = state;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE; //@ LB
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
      default:        next = XXX;
    endcase
  end

  always_comb begin
    n_rd = '0;
    n_ds = '0;
    case (state)
      IDLE : if ( go) n_rd = '1; // READ
             else ;          // IDLE
      READ :          n_rd = '1; // DLY
      DLY  : if ( ws) n_rd = '1; // READ
             else     n_ds = '1; // DONE
      DONE : ;               // IDLE
      default:        {n_rd,n_ds} = 'x;
    endcase
  end

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      rd <= '0;
      ds <= '0;
    end
    else begin
      rd <= n_rd;
      ds <= n_ds;
    end
endmodule
```

Figure 11 - fsm1 - Coding styles effort comparison

Figure 11 shows the relative coding difference between the four coding styles. The full fsm1 RTL code for all four coding styles used in Figure 11 are shown in Appendix 2.2 - Appendix 2.5. The **`fsm1_pkg`** files with both abstract and binary encoded **`typedef`**[s] that were used by all of the fsm1 designs are shown in Appendix 2.1.

Design Compiler O-2018.06-SP4 **LSI 10K Library - lsk_10k**

		No case-default-X / Explicit loopback					case-default-X / Explicit loopback					case-default-X / Implicit loopback				
		RTL	No Clock Goal		Clock Goal 7.5		RTL	No Clock Goal		Clock Goal 7.5		RTL	No Clock Goal		Clock Goal 7.5	
	Enums	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack
fsm1_1	Abstract	39	52	-5.57	52	0.04	44	50	-5.19	49	0.04	42	50	-5.36	51	0.04
fsm1_2	Abstract	34	32	-4.89	38	(1.68)	39	31	-4.94	35	(1.37)	38	28	-5.18	35	(1.37)
fsm1_3	Abstract	38	49	-6.37	49	0.03	41	49	-6.37	49	0.03	40	49	-6.37	57	0.04
fsm1_4	Abstract	48	52	-5.57	52	0.04	50	52	-5.57	52	0.04	49	53	-5.10	52	0.04
fsm1_1	Encoded	39	44	-4.08	44	0.04	44	44	-4.08	44	0.04	42	44	-4.08	44	0.04
fsm1_2	Encoded	34	26	-4.09	28	(1.16)	39	26	-4.09	28	(1.16)	38	26	-4.09	27	(1.16)
fsm1_3	Encoded	38	44	-5.06	45	0.04	41	44	-5.06	45	0.04	40	44	-5.06	45	0.04
fsm1_4	Encoded	48	44	-4.08	44	0.04	50	44	-4.08	44	0.04	49	44	-4.08	44	0.04

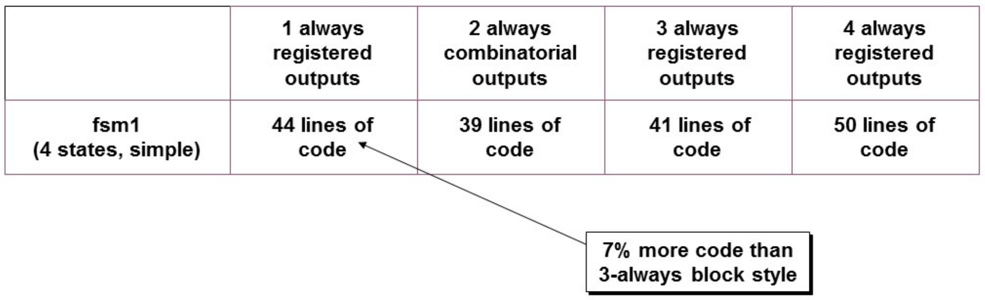
LOC* - Lines Of Code (not including imported enums)

Design Compiler O-2018.06-SP4 **SAED Library - saed32rvt_ss0p95v125c.db**

		No case-default-X / Explicit loopback					case-default-X / Explicit loopback					case-default-X / Implicit loopback				
		RTL	No Clock Goal		Clock Goal 0.65		RTL	No Clock Goal		Clock Goal 0.65		RTL	No Clock Goal		Clock Goal 0.65	
	Abstract	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack
fsm1_1	Abstract	39	51	-0.42	54	0.00	44	51	-0.40	52	0.00	42	52	-0.39	54	0.00
fsm1_2	Abstract	34	34	-0.38	54	(0.12)	39	34	-0.40	42	(0.09)	38	32	-0.40	43	(0.09)
fsm1_3	Abstract	38	48	-0.50	49	0.00	41	48	-0.50	49	0.00	40	48	-0.50	48	0.00
fsm1_4	Abstract	48	51	-0.42	54	0.00	50	51	-0.42	54	0.00	49	50	-0.40	52	0.00
fsm1_1	Encoded	39	42	-0.38	44	0.00	44	42	-0.38	44	0.00	42	42	-0.37	44	0.00
fsm1_2	Encoded	34	26	-0.35	31	(0.06)	39	26	-0.35	31	(0.06)	38	26	-0.35	29	(0.06)
fsm1_3	Encoded	38	42	-0.41	45	0.00	41	42	-0.41	45	0.00	40	42	-0.41	45	0.00
fsm1_4	Encoded	48	42	-0.37	44	0.00	50	42	-0.37	44	0.00	49	42	-0.38	44	0.00

Table 2 - fsm1 - Synthesis efficiency comparison table

When the fsm1 design was synthesized, there were almost no differences when state encodings and pre-default-X values were added. Also there was no noticeable difference when the loopback code was deleted. In this design, the 1-always block coding style had slightly better synthesis results than the 3-always block style, and the 4-always block style synthesis results matched the 1-always block synthesis results. Since this FSM design has four states and since two bits were used to create all possible unique encodings for each state, it is not surprising that adding default-X assignments did not noticeably improve the synthesis results. We expect pre-default-X assignments will prove more useful when the number of states is not a power of 2.



	1 always registered outputs	2 always combinatorial outputs	3 always registered outputs	4 always registered outputs
fsm1 (4 states, simple)	44 lines of code	39 lines of code	41 lines of code	50 lines of code

7% more code than 3-always block style

Figure 12 - fsm1 - Lines of code

fsm1 Coding Goals	1 always	2 always	3 always	4 always
(1) Easy to change state encodings	yes	yes	yes	yes
(2) Concise coding style	yes	yes	yes	yes
(3) Easy to code and understand	~no	yes	yes	~yes
(4) Facilitate debugging	~yes	yes	yes	yes
(5) Yield efficient synthesis results	yes+	~no	~yes	yes+
(6) Easy to change due to FSM changes	~yes	yes	yes	~yes

Figure 13 - fsm1 - Coding goals summary

It is apparent from Figure 12 that the coding styles are rather equal.

8.2 FSM7 with 10 states, 1 output and 2 inputs

The FSM7 design has an asynchronous low-true `rst_n`, two inputs `go` and `jmp` (jump), and just one output `y1`.

There are loopback states in the fsm7 diagram in states `S0` and `S3`.

The coding effort for this FSM design using the different styles does start to show that the 1-always block coding effort is starting to get verbose. This design is somewhat contrived because most FSM designs with 10 states would have multiple outputs and multiple transition arcs, so we do not yet see the full impact of the 1-always block coding style.

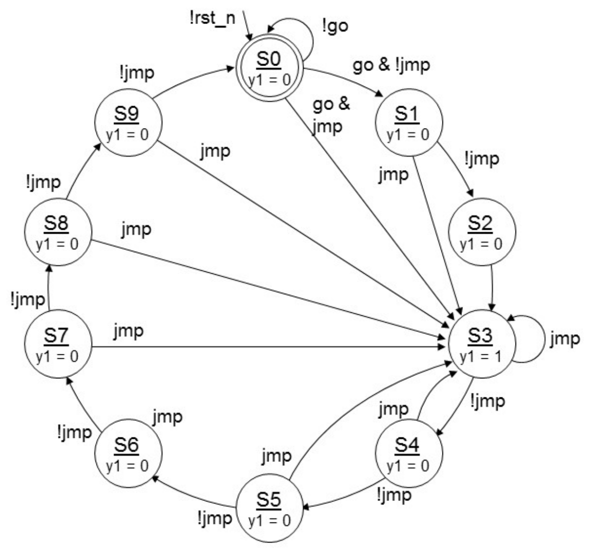


Figure 14 - FSM7 state diagram

As can be seen from the fsm7 design, the 1-always block coding style required 46% more code than the equivalent 3-always block coding style.

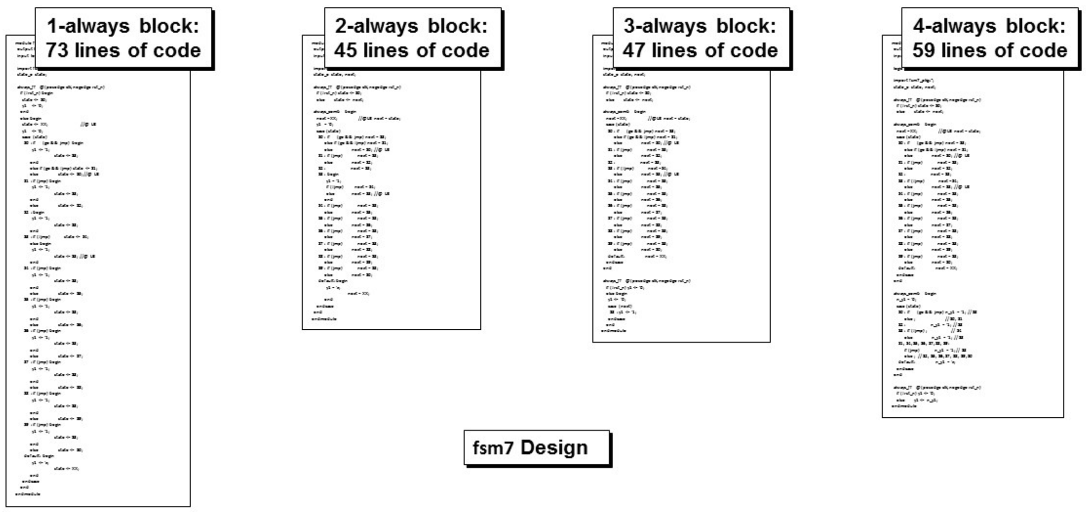


Figure 15 - fsm7 - Coding styles effort comparison

When the fsm7 design was synthesized, there was some area and performance advantage using the 1-always block coding style over the 3-always block coding style as shown in Table 3. This is presumably due to the fact that the registered outputs for the 3-always block coding style are calculated from the **`next`** state values as opposed to using the same inputs tests to calculate the clocked outputs for the 1-always block style. Noteworthy is the fact that the 4-always block style synthesis results were actually slightly better than the 3-always block synthesis results. The full fsm7 RTL code for the coding styles used in Figure 15 are shown in Appendix 2.7 - Appendix 2.10. The **`fsm7_pkg`** files with both abstract and binary encoded **`typedef`**[s] that were used by all of the fsm7 designs are shown in Appendix 2.6.

Design Compiler O-2018.06-SP4 LSI 10K Library - lsk_10k

		No case-default-X / Explicit loopback					case-default-X / Explicit loopback					case-default-X / Implicit loopback				
		RTL	No Clock Goal		Clock Goal 8.5		RTL	No Clock Goal		Clock Goal 8.5		RTL	No Clock Goal		Clock Goal 8.5	
	Enums	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack
fsm7_1	Abstract	69	100	-7.60	153	0.00	73	94	-7.48	105	0.11	70	89	-7.42	97	0.06
fsm7_2	Abstract	41	76	-8.10	118	(1.48)	45	78	-6.87	112	(1.15)	43	78	-8.90	95	(1.15)
fsm7_3	Abstract	46	93	-8.92	164	(0.91)	47	89	-8.14	145	0.01	45	87	-7.82	127	0.02
fsm7_4	Abstract	57	95	-8.38	139	0.07	59	91	-7.79	118	0.03	57	87	-7.96	88	0.12
fsm7_1	Encoded	69	78	-6.54	89	0.11	73	73	-6.15	77	0.24	70	79	-7.37	93	0.07
fsm7_2	Encoded	41	61	-5.79	83	(1.44)	45	61	-6.50	68	(1.15)	43	66	-7.68	78	(1.29)
fsm7_3	Encoded	46	73	-5.20	71	0.24	47	73	-5.20	71	0.24	45	77	-7.30	93	0.06
fsm7_4	Encoded	57	77	-5.32	78	0.19	59	73	-5.20	71	0.24	57	82	-7.69	84	0.14

LOC* - Lines Of Code (not including imported enums)

Design Compiler O-2018.06-SP4 SAED Library - saed32rvt_ss0p95v125c.db

		No case-default-X / Explicit loopback					case-default-X / Explicit loopback					case-default-X / Implicit loopback				
		RTL	No Clock Goal		Clock Goal 0.65		RTL	No Clock Goal		Clock Goal 0.65		RTL	No Clock Goal		Clock Goal 0.65	
	Enums	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack
fsm7_1	Abstract	69	122	-0.77	126	0.00	73	101	-0.63	107	0.00	70	96	-0.68	102	0.00
fsm7_2	Abstract	41	80	-0.68	108	(0.11)	45	87	-0.61	109	(0.07)	43	82	-0.71	95	(0.07)
fsm7_3	Abstract	46	100	-0.73	113	0.00	47	95	-0.65	101	(0.01)	45	92	-0.62	122	0.00
fsm7_4	Abstract	57	105	-0.70	125	0.00	59	103	-0.65	107	0.00	57	96	-0.68	107	0.00
fsm7_1	Encoded	69	89	-0.60	96	0.00	73	80	-0.48	85	0.00	70	86	-0.62	87	0.00
fsm7_2	Encoded	41	65	-0.49	86	(0.08)	45	70	-0.57	70	(0.07)	43	74	-0.73	91	(0.07)
fsm7_3	Encoded	46	80	-0.47	79	0.00	47	80	-0.47	79	0.00	45	90	-0.57	89	0.00
fsm7_4	Encoded	57	86	-0.44	84	0.00	59	80	-0.47	79	0.00	57	90	-0.74	90	0.00

LOC* - Lines Of Code (not including imported enums)

Table 3 - fsm7 - Synthesis efficiency comparison table

	1 always registered outputs	2 always combinatorial outputs	3 always registered outputs	4 always registered outputs
fsm7 (10 states, 1 output)	73 lines of code	45 lines of code	47 lines of code	59 lines of code

55% more code than 3-always block style

Figure 16 - fsm7 - Lines of code

fsm7 Coding Goals	1 always	2 always	3 always	4 always
(1) Easy to change state encodings	yes	yes	yes	yes
(2) Concise coding style	no	yes	yes	yes
(3) Easy to code and understand	~no	yes	yes	~yes
(4) Facilitate debugging	yes	yes	yes	yes
(5) Yield efficient synthesis results	yes+	~no	~yes	yes+
(6) Easy to change due to FSM changes	~no	yes	yes	~yes

Figure 17 - fsm7 - Coding goals summary

It is apparent from Figure 16 that the 1-always block coding style is significantly more verbose than the 3-always block coding style and that the 1-always block code is getting harder to follow all of the next-output assignments.

We have again noted that the 1-always block coding style might not be so easy to change due to FSM changes.

8.2.1 Why is 1 always block more verbose than 3 always block?

When we originally examined different FSM coding styles more than 15 years ago, we were surprised to observe that larger 1-always block FSM RTL coded examples were significantly more verbose than 3-always block coded examples. Based on the names of the styles, this seemed counter intuitive. After all, which sounds bigger? 1-always block or 3-always blocks?

Upon closer examination we noted that the 2-always block and 3-always block RTL coding styles assign the outputs once per state, while the 1-always block style assigned the outputs once per transition arc to each state.

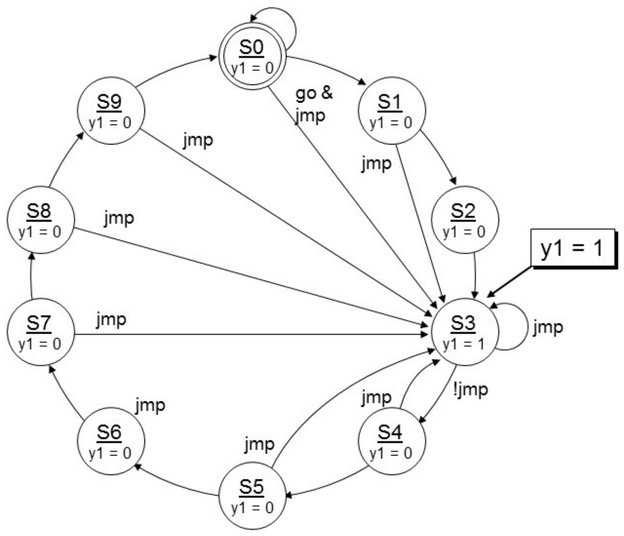


Figure 18 - FSM7 state diagram - why is the 1-always block style so verbose?

In the FSM7 design, as shown in Figure 18, the worst case state is `S3`. Every state in this FSM design that transitions to state `S3` must set `state <= S3` and also change the output assignment by setting `y1 <= '1`. These two assignments must be encapsulated within `begin` - `end` statements. This means that each transition arc to state `S3` must include these four lines of code. This contributes to the rapid code-explosion that is so common when using the 1-always block coding style.

In the RTL code shown in Example 11, it can be seen that there are four lines of code for each state in the `case` statement that are nearly identical. For each transition to state `S3`, the assignment `y1 <= '1` is required and the `state <= S3` assignment is also required. These two assignments must also be surrounded by `begin` - `end`. These four lines of code, added to each case item, quickly add many lines of code to the `case` statement.

```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n) begin
    state <= S0;
    y1    <= '0;
  end
  else begin
```

```
state <= XX;
y1    <= '0;
case (state)
  S0 : if      (go &&  jmp) begin
         y1 <= '1;
                            state <= S3;
       end
       else if (go && !jmp) state <= S1;
       else                 state <= S0; //@ looopback
  S1 : if (jmp) begin
         y1 <= '1;
                            state <= S3;
       end
       else                 state <= S2;
  S2 : begin
         y1 <= '1;
                            state <= S3;
       end
  S3 : if (!jmp)            state <= S4;
       else begin
         y1 <= '1;
                            state <= S3; //@ loopback
       end
  S4 : if (jmp) begin
         y1 <= '1;
                      state <= S3;
       end
       else           state <= S5;
  S5 : if (jmp) begin
         y1 <= '1;
                      state <= S3;
       end
       else           state <= S6;
  S6 : if (jmp) begin
         y1 <= '1;
                      state <= S3;
       end
       else           state <= S7;
  S7 : if (jmp) begin
         y1 <= '1;
                      state <= S3;
       end
       else           state <= S8;
  S8 : if (jmp) begin
         y1 <= '1;
                      state <= S3;
       end
       else           state <= S9;
  S9 : if (jmp) begin
         y1 <= '1;
                      state <= S3;
       end
       else           state <= S0;
  default: begin
         y1 <= 'x;
                      state <= XX;
       end
endcase
```

```
      end
  endmodule
```

Example 11 - fsm7_1 - 1-always block style output assignments for each transition arc

The difference becomes more pronounced when there are more states, multiple output assignments and more transition arcs as shown in the FSM8 design of Section 8.3 .

8.3 FSM8 with 10 states, 3 outputs and 2 inputs

The FSM8 design has an asynchronous low-true **`rst_n`**, four inputs **`go`**, **`sk1`**, **`sk0`** and **`jmp`** (jump), and three outputs **`y3`**, **`y2`** and **`y1`**.

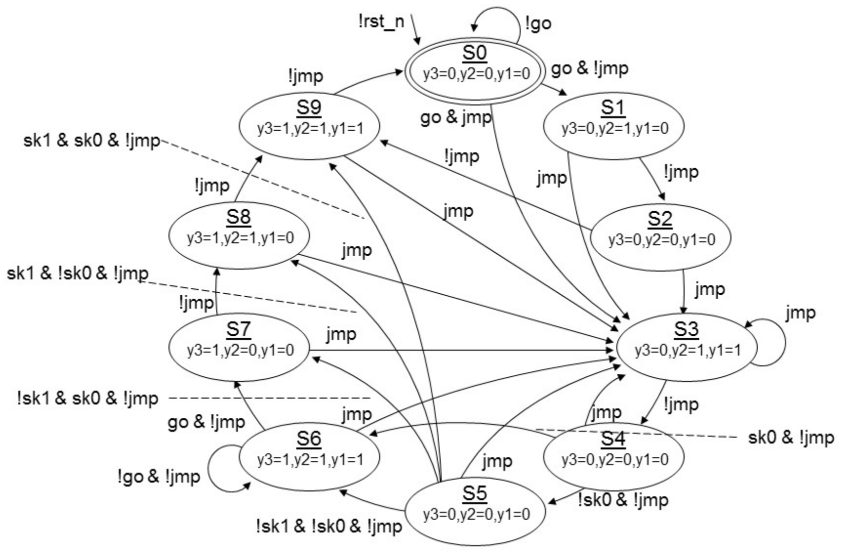


Figure 19 - FSM8 state diagram

As can be seen from the fsm8 design, the 1-always block coding style required 84% more code than the equivalent 3-always block coding style.

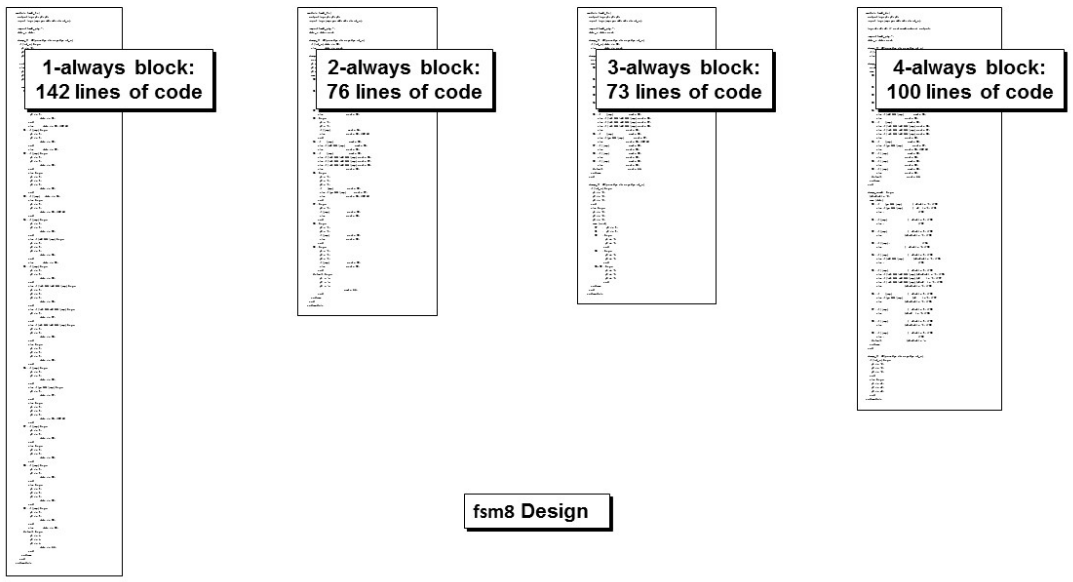


Figure 20 - fsm8 - Coding styles effort comparison

Design Compiler O-2018.06-SP4 LSI 10K Library - lsk_10k

		No case-default-X / Explicit loopback					case-default-X / Explicit loopback					case-default-X / Implicit loopback				
		RTL	No Clock Goal		Clock Goal 9.0		RTL	No Clock Goal		Clock Goal 9.0		RTL	No Clock Goal		Clock Goal 9.0	
	Enums	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack
fsm8_1	Abstract	136	147	-10.66	220	0.03	142	163	-8.63	207	0.00	138	149	-8.37	179	0.15
fsm8_2	Abstract	70	105	-10.57	190	(2.99)	76	100	-7.88	150	(1.95)	73	98	-8.62	141	(1.87)
fsm8_3	Abstract	72	140	-14.81	284	(0.72)	73	133	-11.50	229	(1.38)	70	139	-12.62	246	(1.39)
fsm8_4	Abstract	98	155	-10.11	261	(0.03)	100	157	-9.02	190	0.02	97	162	-10.45	210	0.01
fsm8_1	Encoded	136	140	-11.38	213	(0.09)	142	164	-7.85	195	0.02	138	175	-8.73	198	0.03
fsm8_2	Encoded	70	94	-7.95	154	(2.70)	76	85	-7.19	130	(2.16)	73	95	-9.57	141	(1.69)
fsm8_3	Encoded	72	128	-11.55	224	0.01	73	128	-11.55	224	0.01	70	146	-11.90	270	(0.95)
fsm8_4	Encoded	98	143	-8.64	180	0.01	100	164	-7.85	195	0.02	97	174	-8.02	220	0.00

LOC* - Lines Of Code (not including imported enums)

Design Compiler O-2018.06-SP4 SAED Library - saed32rvt_ss0p95v125c.db

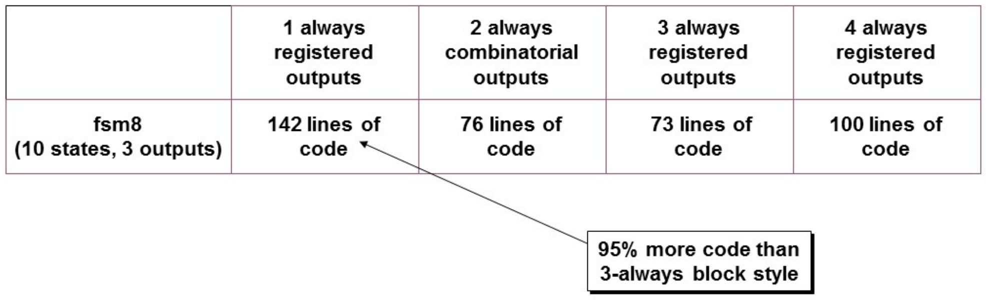
		No case-default-X / Explicit loopback					case-default-X / Explicit loopback					case-default-X / Implicit loopback				
		RTL	No Clock Goal		Clock Goal 0.8		RTL	No Clock Goal		Clock Goal 0.8		RTL	No Clock Goal		Clock Goal 0.8	
	Enums	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack
fsm8_1	Abstract	136	170	-0.84	185	0.00	142	180	-0.75	195	0.00	138	177	-0.90	186	0.00
fsm8_2	Abstract	70	119	-0.80	166	(0.17)	76	120	-0.74	142	(0.12)	73	124	-0.66	148	(0.09)
fsm8_3	Abstract	72	154	-1.15	210	(0.05)	73	150	-1.07	173	(0.01)	70	152	-1.19	177	0.00
fsm8_4	Abstract	98	178	-0.84	199	0.00	100	177	-0.84	187	0.00	97	185	-0.82	181	0.00
fsm8_1	Encoded	136	160	-0.86	177	0.00	142	193	-0.69	197	0.00	138	212	-0.71	211	0.00
fsm8_2	Encoded	70	116	-0.72	145	(0.16)	76	97	-0.53	150	(0.15)	73	116	-0.85	141	(0.11)
fsm8_3	Encoded	72	140	-0.93	176	0.00	73	140	-0.93	176	0.00	70	171	-0.97	243	(0.01)
fsm8_4	Encoded	98	164	-0.74	172	0.00	100	193	-0.69	197	0.00	97	220	-0.69	222	0.00

LOC* - Lines Of Code (not including imported enums)

Table 4 - fsm8 - Synthesis efficiency comparison table

Surprisingly, when the fsm8 design was synthesized, there was some area and performance advantage using the 3-always block coding style over the 1-always block and 4-always block coding styles as shown inTable 4. This observation suggests that first coding FSM designs using the concise and efficient 3-always block style and then copying the code to incorporate the 4-always block modifications is a good strategy. The copied 4-always block style typically will have better synthesis results than the 3-always block style, but not always.

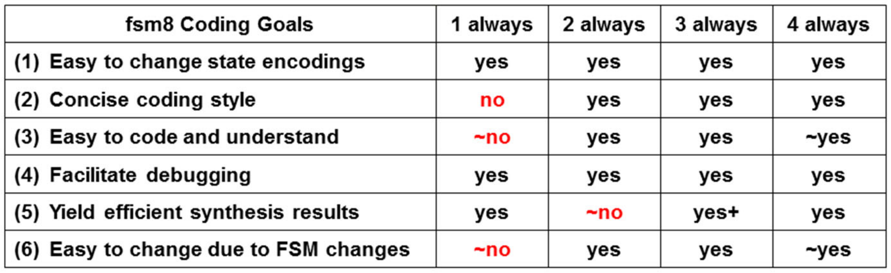
The full fsm8 RTL code for the coding styles used in Figure 20 are shown in Appendix 2.11 - Appendix 2.11. The `fsm8_pkg` files with both abstract and binary encoded `typedef`[s] that were used by all of the fsm8 designs are shown in Appendix 2.11.



	1 always registered outputs	2 always combinatorial outputs	3 always registered outputs	4 always registered outputs
fsm8 (10 states, 3 outputs)	142 lines of code	76 lines of code	73 lines of code	100 lines of code

95% more code than 3-always block style

Figure 21 - fsm8 - Lines of code



fsm8 Coding Goals	1 always	2 always	3 always	4 always
(1) Easy to change state encodings	yes	yes	yes	yes
(2) Concise coding style	no	yes	yes	yes
(3) Easy to code and understand	~no	yes	yes	~yes
(4) Facilitate debugging	yes	yes	yes	yes
(5) Yield efficient synthesis results	yes	~no	yes+	yes
(6) Easy to change due to FSM changes	~no	yes	yes	~yes

Figure 22 - fsm8 - Coding goals summary

We see in Figure 21 and Figure 22 that the 1-always block coding style is significantly more verbose than the 3-always block coding style and that the 1-always block code is getting even harder to understand all of the next-output assignments, while again noting that the 1-always block and 4-always block coding styles were not necessarily more efficient when synthesized.

We have also noted that the 1-always block coding style might not be very easy to change due to FSM changes.

8.4 Prep4 FSM design with 16 states, 8-bit output and 8-bit input

The Prep4 design has an asynchronous low-true `rst_n`, one 8-bit input `i`, and one 8-bit output `o`.

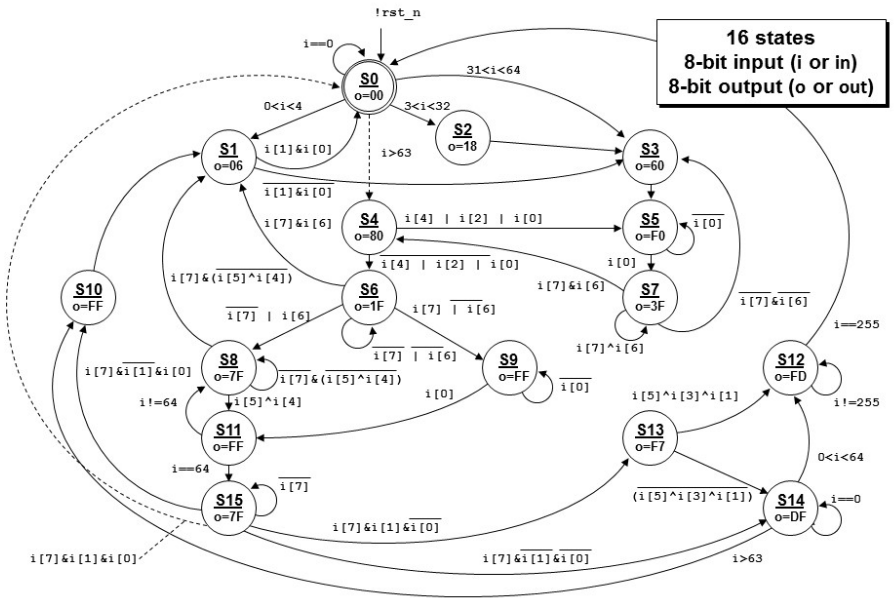


Figure 23 - Prep4 state diagram

As can be seen in Figure 24 for the prep4 design, the 1-always block coding style required more than twice as much code (114% more code) than the equivalent 3-always block coding style.

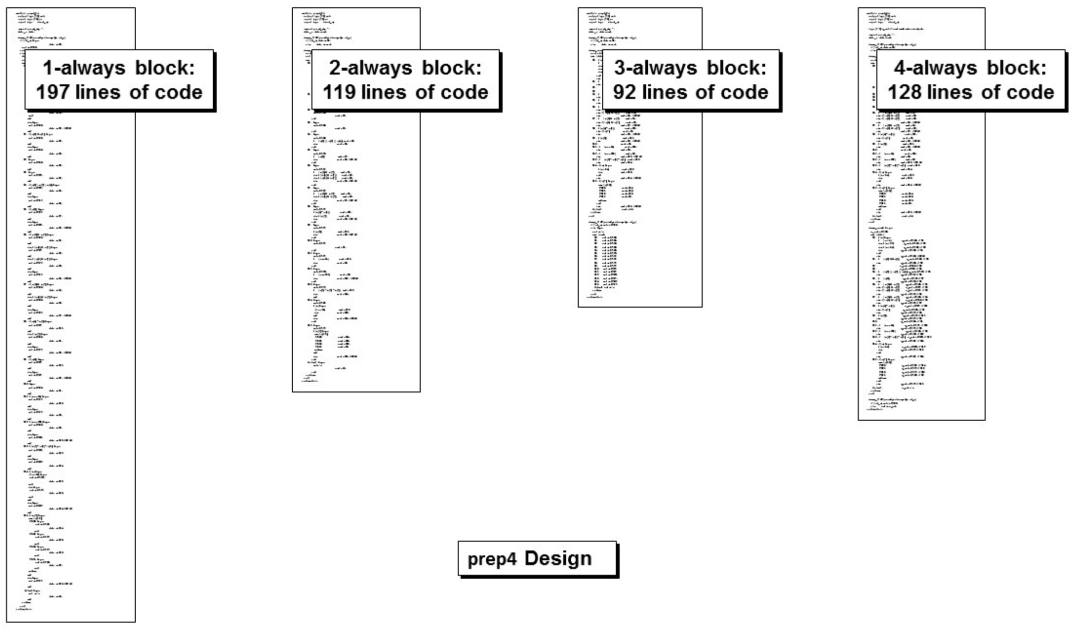

1-always block: 197 lines of code
2-always block: 119 lines of code
3-always block: 92 lines of code
4-always block: 128 lines of code
prep4 Design

Figure 24 - prep4 - Coding styles effort comparison

		Design Compiler O-2018.06-SP4					LSI 10K Library - lsk_10k									
		No case-default-X / Explicit loopback					case-default-X / Explicit loopback					case-default-X / Implicit loopback				
		RTL	No Clock Goal		Clock Goal 11		RTL	No Clock Goal		Clock Goal 11		RTL	No Clock Goal		Clock Goal 11	
	Enums	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack
prep4_1	Abstract	188	304	-10.96	421	0.00	192	303	-11.17	353	0.04	182	334	-11.03	451	0.00
prep4_2	Abstract	115	201	-9.44	345	(4.65)	119	199	-9.06	312	(4.91)	110	222	-13.57	360	(2.54)
prep4_3	Abstract	90	285	-16.32	476	(1.58)	92	281	-14.27	511	(0.62)	83	297	-16.30	497	(2.09)
prep4_4	Abstract	126	328	-12.09	453	(0.34)	128	304	-11.49	419	0.02	118	340	-12.64	487	0.00
prep4_1	Encoded	188	297	-10.69	363	0.01	192	297	-12.32	353	0.00	182	355	-14.89	423	0.02
prep4_2	Encoded	115	198	-9.01	357	(4.84)	119	202	-10.11	310	(5.08)	110	222	-13.22	360	(3.10)
prep4_3	Encoded	90	278	-15.89	491	(1.61)	92	291	-15.37	580	(1.14)	83	293	-17.42	491	(2.68)
prep4_4	Encoded	126	302	-11.61	346	0.04	128	297	-11.11	361	0.01	118	350	-11.62	458	0.01
		LOC*	- Lines Of Code		(not including imported enums)											
		Design Compiler O-2018.06-SP4					SAED Library - saed32rvt_ss0p95v125c.db									
		No case-default-X / Explicit loopback					case-default-X / Explicit loopback					case-default-X / Implicit loopback				
		RTL	No Clock Goal		Clock Goal 0.8		RTL	No Clock Goal		Clock Goal 0.8		RTL	No Clock Goal		Clock Goal 0.8	
	Enums	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack	LOC*	Area	Slack	Area	Slack
prep4_1	Abstract	188	369	-0.85	373	0.00	192	398	-0.78	395	0.00	182	441	-0.90	494	(0.01)
prep4_2	Abstract	115	276	-0.66	345	(0.34)	119	275	-0.69	327	(0.29)	110	314	-0.95	355	(0.20)
prep4_3	Abstract	90	354	-1.19	448	(0.09)	92	367	-1.21	408	(0.08)	83	377	-1.40	526	(0.11)
prep4_4	Abstract	126	409	-0.79	400	0.00	128	399	-0.81	401	0.00	118	438	-1.02	478	(0.08)
prep4_1	Encoded	188	383	-0.73	386	0.00	192	379	-0.87	368	0.00	182	456	-0.92	488	0.00
prep4_2	Encoded	115	267	-0.65	376	(0.33)	119	272	-0.69	339	(0.32)	110	315	-0.83	368	(0.26)
prep4_3	Encoded	90	358	-1.14	469	(0.14)	92	355	-1.19	426	(0.05)	83	368	-1.34	529	(0.09)
prep4_4	Encoded	126	383	-0.86	371	(0.01)	128	379	-0.87	399	0.00	118	463	-0.98	493	(0.01)
		LOC*	- Lines Of Code		(not including imported enums)											

Table 5 - prep4 - Synthesis efficiency comparison table

It should be noted that the outputs in this prep4 design are shown in vector form for each state, so the easiest way to code the outputs in this FSM design was to assign the output vector for each state and ***not*** make default-X output assignments. Since the outputs are calculated from each `next` state, the area increased significantly when synthesized. The full prep4 RTL code for the coding styles used in Figure 24 are shown in Appendix 2.17 - Appendix 2.20. The `prep4_pkg` files with both abstract and binary encoded `typedef`[s] that were used by all of the prep4 designs are shown in Appendix 2.16.

	1 always registered outputs	2 always combinatorial outputs	3 always registered outputs	4 always registered outputs
prep4 (16 states, complex)	192 lines of code	119 lines of code	92 lines of code	128 lines of code

109% more code than 3-always block style

Figure 25 - prep4 - Lines of code

prep4 Coding Goals	1 always	2 always	3 always	4 always
(1) Easy to change state encodings	yes	yes	yes	yes
(2) Concise coding style	no	yes	yes	yes
(3) Easy to code and understand	~no	yes	yes	~yes
(4) Facilitate debugging	yes	yes	yes	yes
(5) Yield efficient synthesis results	yes+	~no	~yes	yes+
(6) Easy to change due to FSM changes	~no	yes	yes	~yes

Figure 26 - prep4 - Coding goals summary

As seen in Figure 25, the 1-always block coding style is more than twice as verbose as the 3-always block coding style and the 1-always block code is getting much harder to follow when examining the next-output assignments.

We have again noted in Figure 26 that the 1-always block coding style is not so easy to change due to FSM changes.

9. Comparisons & Summary

SystemVerilog coding styles help facilitate proper simulation and understanding of the FSM coding styles. Below is a list of items that were identified as checklist items in this paper. After coding your FSM design, we recommend compare your code against the items in this checklist.

Checklist item: Engineers should generally declare all FSM ports and all FSM internal signals to be of type `logic`.

Checklist item: Where ever possible, use the SystemVerilog `'0` / `'1` / `'x` to make assignments.

Checklist item: Declare all ports to be of type `logic`. Extra points for listing outputs followed by inputs followed by control inputs. Extra points for grouping multiple signals into each `output` and `input` port list declaration.

Checklist item: Declare the `state` and `next` variables using enumerated types. Add an `XX` or `XXX` state to help debug the design and to help optimize the synthesized result.

Checklist item: Use `always_ff` and `always_comb` procedures to infer clocked and combinatorial logic. Do not use the older Verilog `always` procedures.

Checklist item: Declare the state register using just 3 lines of code and place it at the top of the design after the enumerated type declaration.

Checklist item: Use a default `next='x` or `next=state` default assignment at the top of the `always_comb` procedure. Start by first using the `next='x` style to help debug the FSM design.

Checklist item: Extra points for using `next` instead of using `nextstate` in the FSM design, but both work fine.

Checklist item: Extra points for placing the `next` assignments in a neat column in the FSM RTL code.

Checklist item: Place default output assignments before the output-assignment `case` statement and then to update the appropriate outputs for each state where the outputs change.

When comparing the coding effort for the four coding styles (1-always block, 2-always block, 3-always block and 4-always block) the 1-always block coding style is very verbose and increases in size quickly with more states and outputs. For the larger prep4 design, the coding effort was more than twice as hard as coding the 3-always block style.

It was also interesting to note that the 3-always block style frequently required less coding effort than the 2-always block style with combinatorial outputs.

The 1-always block and 4-always block coding styles typically gave slightly better synthesis area and timing performance over the 3-always block style, but we recommend using the efficient 3-always block style and then converting it to a 4-always block style if slightly better synthesis performance is needed.

In later publications, we will compare these styles to OneHot and output encoded FSM coding styles to see if the improved area and timing trend continues.

10. Acknowledgements

We are grateful to good friend and colleague Paul Zimmer, creator of the FizZim FSM diagram and code generation tool[16], for his review and valuable suggestions to help improve this paper.

We also thank our good friend and colleague, Steve Golson, for his recommendations to use newer technology libraries and to use input and output design constraints with our FSM designs.

WE also thank Jing Zhan, Robert Lombardi and Benny Widen for their valuable feedback and suggestions regarding the paper and presentation.

We thank Will Cummings of Synopsys for his help with using the Synplify Pro and Synplify Premier FPGA synthesis tools.

11. References

[1] Armenia Educational Programs - SAED: Synopsys Armenia Educational Department https://www.synopsys.com/company/contact-synopsys/office-locations/armenia/armenia-educational-program.html

[2] Clifford E. Cummings, *"Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements,"* SNUG http://www.sunburst-design.com/papers/CummingsSNUG2003SJ_SystemVerilogFSM.pdf

[3] Clifford E. Cummings, "SystemVerilog Assertions - Bindfiles & Best Known Practices for Simple SVA Usage," SNUG 2016 (Silicon Valley). http://www.sunburst-design.com/papers/CummingsSNUG2016SV_SVA_Best_Practices.pdf

[4] Clifford E. Cummings, *"SystemVerilog Logic Specific Processes for Synthesis - Benefits and Proper Usage," SNUG* http://www.sunburst-design.com/papers/CummingsSNUG2016SV_SVLogicProcs.pdf

[5] Dejan Marković, "Logic Synthesis - Lecture 11," EEM216A Lecture Slides, Fall 2012 http://icslwebs.ee.ucla.edu/dejan/researchwiki/images/0/0c/F2012-Lec-11_Logic-Synthesis.pdf

[6] Harry Foster, *"Part 5: The 2010 Wilson Research Group Functional Verification Study," (See Figure 2 - 32% Debug)* https://blogs.mentor.com/verificationhorizons/blog/2011/04/04/part-5-the-2010-wilson-research-group-functional-verification-study/

[7] Harry Foster, *"Part 6: The 2012 Wilson Research Group Functional Verification Study," (See Figure 2 - 36% Debug)* https://blogs.mentor.com/verificationhorizons/blog/2013/07/22/part-6-the-2012-wilson-research-group-functional-verification-study/

[8] Harry Foster, *"Part 3: The 2014 Wilson Research Group Functional Verification Study," (See Figure 2 - 43% Debug)* https://blogs.mentor.com/verificationhorizons/blog/2015/04/01/part-3-the-2014-wilson-research-group-functional-verification-study/

[9] Harry Foster, *"Part 8: The 2014 Wilson Research Group Functional Verification Study," (See Figure 5 - 37% Debug)* https://blogs.mentor.com/verificationhorizons/blog/2015/07/13/part-8-the-2014-wilson-research-group-functional-verification-study/

[10] Harry Foster, *"Part 3: The 2016 Wilson Research Group Functional Verification Study," (See Figure 2 - 43% Debug)* https://blogs.mentor.com/verificationhorizons/blog/2016/08/29/part-3-the-2016-wilson-research-group-functional-verification-study/

[11] Harry Foster, *"Part 8: The 2016 Wilson Research Group Functional Verification Study,"* *(See Figure 5 - 39% Debug)* https://blogs.mentor.com/verificationhorizons/blog/2016/10/04/part-8-the-2016-wilson-research-group-functional-verification-study/

[12] Harry Foster, *"Part 4: The 2018 Wilson Research Group Functional Verification Study,"* *(See Figure 4-2 - 42% Debug)* https://blogs.mentor.com/verificationhorizons/blog/2019/01/02/part-4-the-2018-wilson-research-group-functional-verification-study/

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

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

[15] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE, New York, NY, IEEE Std 1800™-2017

[16] Paul Zimmer, Michael Zimmer, Brian Zimmer, *"FizZim - an open-source FSM design environment,"* http://www.fizzim.com/mydownloads/fizzim_tutorial_20160423.pdf

[17] SAED Technology Libraries - SolvNet, "Bitcoin Low Power Case Study," (download Technology Libraries) https://solvnet.synopsys.com/retrieve/2630223.html

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

Lines of Code Web References:

[19] Amy Bowersox, *"How true is the following: "Every line of code is a bug"?,"* https://www.quora.com/How-true-is-the-following-Every-line-of-code-is-a-bug

[20] Chad Perrin, *"The danger of complexity: More code, more bugs,"* https://www.techrepublic.com/blog/it-security/the-danger-of-complexity-more-code-more-bugs/

[21] Charles Brian Quinn, *"Less Code is Better,"* https://www.bignerdranch.com/blog/less-code-is-better/

[22] Clifford E. Cummings, *"The Sunburst Design - "Where's Waldo" Principle of Verilog Coding,"* http://www.sunburst-design.com/papers/Wheres_Waldo_Coding.pdf

[23] Jack Ganssle, *"Keep It Small,"* http://www.ganssle.com/articles/keepsmall.htm

[24] Quora, *"Is there a correlation between the number of lines of code and the number of bugs in software?,"* https://www.quora.com/Is-there-a-correlation-between-the-number-of-lines-of-code-and-the-number-of-bugs-in-software

[25] Software Engineering, *"More code = more bugs,"* https://softwareengineering.stackexchange.com/questions/41949/more-code-more-bugs

[26] Steve Baker, *"What is the average ratio of bugs to a line of code?,"* https://www.quora.com/What-is-the-average-ratio-of-bugs-to-a-line-of-code

12. Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 38 years of ASIC, FPGA and system design experience and 28 years of SystemVerilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past 17 years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee from 1994-2012, and has presented more than 50 papers on SystemVerilog & SystemVerilog related design, synthesis and verification techniques.

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

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.

Email address: cliffc@sunburst-design.com

Heath Chambers is President of HMC Design Verification, Inc., a company that specializes in design and verification consulting and high tech training. Mr. Chambers is a consultant with 23 years of Verilog Experience 16 years of SystemVerilog experience, 19 years of consulting and verification lead experience for multiple projects and has been an instructor for Sunburst Design since the year 2000. Heath has 18 years of SystemVerilog, Verilog, synthesis and UVM Verification methodology training experience for Sunburst Design, Inc., and was previously a contract Specman Basic Training instructor for Verisity. Heath has ASIC and system verification, firmware, and self-test design experience and is capable of answering the very technical questions asked by experienced verification engineers.

Mr. Chambers, was a member of the IEEE 1364 Verilog and IEEE 1800 SystemVerilog Standards Groups from 2000 to 2012, and has helped to develop and improve Sunburst Design Verilog, SystemVerilog, UVM and synthesis training courses.

Mr. Chambers specializes in verification of ASICs and systems using top-down design methodologies and is proficient in SystemVerilog, Verilog, UVM, 'e', C, and Perl. Mr. Chambers specializes in the Questa, Cadence, Synopsys simulation tools.

Before becoming an independent Consultant, Mr. Chambers worked for Hewlett-Packard doing verification of multi-million gate ASICs and systems containing multiple chips. Mr. Chambers was the lead verification engineer for the last two projects he worked on before leaving the company.

Mr. Chambers holds a BSCS from New Mexico Institute of Mining and Technology.

Email address: hmcdvi@msn.com

Last Updated: February 2019

Appendix 1 **Tools and OS versions**

The ability of Verilog and SystemVerilog tools used to simulate FSM designs typically does not change with simulation tool versions. Simulation of FSM designs is a well-established task that has not changed in the foreseeable past.

On the other hand, synthesis results continue to evolve with different tool versions and has changed significantly over the years.

The examples in this paper were run using the following Linux and Synopsys tool versions:

64-bit Linux laptop: CentOS release 6.5

VCS version N-2017.12-SP1_Full64

> Running vcs and dve each required the command line switch -full64
>
> Without the -full64 command line switch, vcs compilation would fail with the message:

```
error while loading shared libraries: libelf.so.1: cannot open shared
object file: No such file or directory.
```

Design Compiler versions O-2018.06-SP4 and K-2015.06-SP4

Synplify Premier version O-2018.09-SP1

Synthesis Libraries:

LSI 10K - old synthesis library that ships with Design Compiler

SAED32

Appendix 2 **FSM benchmark source code**

The source code for the FSM1, FSM7, FSM8 and Prep4 source files are included in this appendix.

Appendix 2.1 **FSM1 - fsm1_pkgs for abstract and binary encoded enums**

The [a]bstract enumerated types package file is named `fsm1_pkg_a.sv` but the package name is `fsm1_pkg`.

```
package fsm1_pkg;
  typedef enum {IDLE,
                READ,
                DLY,
                DONE,
                XXX } state_e;
endpackage
```

Example 12 - File: fsm1_pkg_a.sv - fsm1_pkg enumerated typedef

The [b]inary encoded enumerated types package file is named `fsm1_pkg_b.sv` but the package name is still `fsm1_pkg`.

```
package fsm1_pkg;
  typedef enum logic [1:0] {IDLE = 2'b00,
                            READ = 2'b01,
                            DLY  = 2'b11,
                            DONE = 2'b10,
                            XXX  = 'x   } state_e;
endpackage
```

Example 13 - File: fsm1_pkg_b.sv - fsm1_pkg enumerated typedef

These packages include a typedef for the `state_e` type that is used by all of the `fsm1` designs.

Using the same package name makes it possible to read the separate package files first in a simulation and first in synthesis compilation to selectively choose abstract or binary encoded enumerated types without touching the `fsm1` files.

Appendix 2.2 **FSM1 - 1 always block RTL - NOT Recommended - Registered Outputs**

```
module fsm1_1x (
  output logic rd, ds,
  input  logic go, ws, clk, rst_n);

  import fsm1_pkg::*;
  state_e state;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      state <= IDLE;
      rd    <= '0;
      ds    <= '0;
    end
    else begin
      state <= XXX;                           //@ LB
      rd    <= '0;
      ds    <= '0;
      case (state)
        IDLE : if (go) begin
                 rd <= '1;
                             state <= READ;
               end
               else          state <= IDLE; //@ LB
        READ : begin
                 rd <= '1;
                             state <= DLY;
               end
        DLY  : if (!ws) begin
                 ds <= '1;
                             state <= DONE;
               end
               else begin
                 rd <= '1;
                             state <= READ;
               end
        DONE :               state <= IDLE;
        default: begin
                 ds <= 'x;
                 rd <= 'x;
                             state <= XXX;
               end
      endcase
    end
endmodule
```

Example 14 - fsm1_1x - 1-always block with registered outputs

Appendix 2.3 **FSM1 - 2 always block RTL - Recommended - Combinatorial Outputs**

```
module fsm1_2x (
  output logic rd, ds,
  input  logic go, ws, clk, rst_n);

  import fsm1_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;          //@LB next = state;
    rd   = '0;
    ds   = '0;
    case (state)
      IDLE : if (go)     next = READ;
             else        next = IDLE; //@ LB
      READ : begin
               rd = '1;
                         next = DLY;
             end
      DLY  : begin
               rd = '1;
               if (!ws)  next = DONE;
               else      next = READ;
             end
      DONE : begin
               ds = '1;
                         next = IDLE;
             end
      default: begin
               ds = 'x;
               rd = 'x;
                         next = XXX;
             end
    endcase
  end
endmodule
```

Example 15 - fsm1_2x - 2-always block with combinatorial outputs

Appendix 2.4 **FSM1 - 3 always block RTL- NOT Recommended - Registered Outputs**

```
module fsm1_3x (
  output logic rd, ds,
  input  logic go, ws, clk, rst_n);

  import fsm1_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;       //@LB next = state;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE; //@ LB
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
      default:        next = XXX;
    endcase
  end

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      rd <= '0;
      ds <= '0;
    end
    else begin
      rd <= '0;
      ds <= '0;
      case (next)
        IDLE : ;
        READ : rd <= '1;
        DLY  : rd <= '1;
        DONE : ds <= '1;
        default: {rd,ds} <= 'x;
      endcase
    end
endmodule
```

Example 16 - fsm1_3x - 3-always block style with registered outputs

Appendix 2.5 **FSM1 - 4 always block RTL - Recommended - Registered Outputs**

```
module fsm1_4x (
  output logic rd, ds,
  input  logic go, ws, clk, rst_n);

  logic n_rd, n_ds; // next combinatorial outputs

  import fsm1_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XXX;       //@LB next = state;
    case (state)
      IDLE : if (go)  next = READ;
             else     next = IDLE; //@ LB
      READ :          next = DLY;
      DLY  : if (!ws) next = DONE;
             else     next = READ;
      DONE :          next = IDLE;
      default:        next = XXX;
    endcase
  end

  always_comb begin
    n_rd = '0;
    n_ds = '0;
    case (state)
      IDLE : if ( go)  n_rd = '1; // READ
             else ;               // IDLE
      READ :           n_rd = '1; // DLY
      DLY  : if ( ws)  n_rd = '1; // READ
             else      n_ds = '1; // DONE
      DONE : ;                    // IDLE
      default:        {n_rd,n_ds} = 'x;
    endcase
  end

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      rd <= '0;
      ds <= '0;
    end
    else begin
      rd <= n_rd;
      ds <= n_ds;
    end
endmodule
```

Example 17 - fsm1_4x - 3-always block style with registered outputs

Appendix 2.6 **FSM7 - fsm7_pkgs for abstract and binary encoded enums**

The [a]bstract enumerated types package file is named `fsm7_pkg_a.sv` but the package name is `fsm7_pkg`.

```
package fsm7_pkg;
  typedef enum {S0,
                S1,
                S2,
                S3,
                S4,
                S5,
                S6,
                S7,
                S8,
                S9,
                XX } state_e;
endpackage
```

Example 18 - File: fsm7_pkg_a.sv - fsm7_pkg enumerated typedef

The [b]inary encoded enumerated types package file is named `fsm7_pkg_b.sv` but the package name is still `fsm7_pkg`.

```
package fsm7_pkg;
  typedef enum logic [3:0] {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,
                            XX  = 'x     } state_e;
endpackage
```

Example 19 - File: fsm7_pkg_b.sv - fsm7_pkg enumerated typedef

These packages include a typedef for the `state_e` type that is used by all of the `fsm7` designs.

Using the same package name makes it possible to read the separate package files first in a simulation and first in synthesis compilation to selectively choose abstract or binary encoded enumerated types without touching the `fsm7` files.

Appendix 2.7 **FSM7 - 1 always block RTL - NOT Recommended - Registered Outputs**

```
module fsm7_1x (
  output logic y1,
  input  logic jmp, go, clk, rst_n);

  import fsm7_pkg::*;
  state_e state;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      state <= S0;
      y1    <= '0;
    end
    else begin
      state <= XX;                              //@ LB
      y1    <= '0;
      case (state)
        S0 : if      (go &&  jmp) begin
               y1 <= '1;
                                  state <= S3;
             end
             else if (go && !jmp) state <= S1;
             else                 state <= S0; //@ LB
        S1 : if (jmp) begin
               y1 <= '1;
                                  state <= S3;
             end
             else                 state <= S2;
        S2 : begin
               y1 <= '1;
                                  state <= S3;
             end
        S3 : if (!jmp)            state <= S4;
             else begin
               y1 <= '1;
                                  state <= S3; //@ LB
             end
        S4 : if (jmp) begin
               y1 <= '1;
                                  state <= S3;
             end
             else                 state <= S5;
        S5 : if (jmp) begin
               y1 <= '1;
                                  state <= S3;
             end
             else                 state <= S6;
        S6 : if (jmp) begin
               y1 <= '1;
                                  state <= S3;
             end
             else                 state <= S7;
        S7 : if (jmp) begin
               y1 <= '1;
                                  state <= S3;
             end
             else                  state <= S8;
        S8 : if (jmp) begin
               y1 <= '1;
                                   state <= S3;
             end
             else                  state <= S9;
        S9 : if (jmp) begin
               y1 <= '1;
                                   state <= S3;
             end
             else                  state <= S0;
        default: begin
               y1 <= 'x;
                                   state <= XX;
             end
      endcase
    end
endmodule
```

Example 20 - fsm7_1x - 1-always block style with registered outputs

Appendix 2.8 **FSM7 - 2 always block RTL - Recommended - Combinatorial Outputs**

```
module fsm7_2x (
  output logic y1,
  input  logic jmp, go, clk, rst_n);

  import fsm7_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                     //@LB next = state;
    y1   = '0;
    case (state)
      S0 : if       (go &&  jmp) next = S3;
           else if (go && !jmp) next = S1;
           else                  next = S0; //@ LB
      S1 : if (jmp)              next = S3;
           else                  next = S2;
      S2 :                       next = S3;
      S3 : begin
             y1 = '1;
             if (!jmp)           next = S4;
             else                next = S3; //@ LB
           end
      S4 : if (jmp)              next = S3;
           else                  next = S5;
      S5 : if (jmp)              next = S3;
           else                  next = S6;
      S6 : if (jmp)              next = S3;
           else                  next = S7;
      S7 : if (jmp)              next = S3;
           else                  next = S8;
      S8 : if (jmp)              next = S3;
           else                  next = S9;
      S9 : if (jmp)              next = S3;
           else                  next = S0;
      default: begin
             y1 = 'x;
                                 next = XX;
           end
    endcase
  end
endmodule
```

Example 21 - fsm7_2x - 2-always block style with combinatorial outputs

Appendix 2.9 **FSM7 - 3 always block - Recommended - Registered Outputs**

```
module fsm7_3x (
  output logic y1,
  input  logic jmp, go, clk, rst_n);

  import fsm7_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                    //@LB next = state;
    case (state)
      S0 : if      (go &&  jmp) next = S3;
           else if (go && !jmp) next = S1;
           else                 next = S0; //@ LB
      S1 : if (jmp)             next = S3;
           else                 next = S2;
      S2 :                      next = S3;
      S3 : if (!jmp)            next = S4;
           else                 next = S3; //@ LB
      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;
      default:                  next = XX;
    endcase
  end

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) y1 <= '0;
    else begin
      y1 <= '0;
      case (next)
        S3 : y1 <= '1;
      endcase
    end
endmodule
```

Example 22 - fsm7_3x - 3-always block style with registered outputs

Appendix 2.10 **FSM7 - 4 always block RTL - Recommended - Registered Outputs**

```
module fsm7_4x (
  output logic y1,
  input  logic jmp, go, clk, rst_n);

  logic n_y1; // next combinatorial outputs

  import fsm7_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                      //@LB next = state;
    case (state)
      S0 : if      (go &&  jmp) next = S3;
           else if (go && !jmp) next = S1;
           else                 next = S0; //@ LB
      S1 : if (jmp)             next = S3;
           else                 next = S2;
      S2 :                      next = S3;
      S3 : if (!jmp)            next = S4;
           else                 next = S3; //@ LB
      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;
      default:                  next = XX;
    endcase
  end

  always_comb begin
    n_y1 = '0;
    case (state)
      S0 : if      (go &&  jmp) n_y1 = '1; // S3
           else ;                          // S0, S1
      S2 :                      n_y1 = '1; // S3
      S3 : if (!jmp) ;                     // S4
           else                 n_y1 = '1; // S3
      S1, S4, S5, S6, S7, S8, S9:
           if (jmp)             n_y1 = '1; // S3
           else ;  // S2, S5, S6, S7, S8, S9, S0
      default:                  n_y1 = 'x;
    endcase
  end
```

```
    always_ff @(posedge clk, negedge rst_n)
      if (!rst_n) y1 <= '0;
      else        y1 <= n_y1;
  endmodule
```

Example 23 - fsm7_4x - 4-always block style with registered outputs

Appendix 2.11 **FSM8 - fsm8_pkgs for abstract and binary encoded enums**

The [a]bstract enumerated types package file is named `fsm8_pkg_a.sv` but the package name is `fsm8_pkg`.

```
package fsm8_pkg;
  typedef enum {S0,
                S1,
                S2,
                S3,
                S4,
                S5,
                S6,
                S7,
                S8,
                S9,
                XX } state_e;
endpackage
```

Example 24 - File: fsm8_pkg_a.sv - fsm8_pkg enumerated typedef

The [b]inary encoded enumerated types package file is named `fsm8_pkg_b.sv` but the package name is still `fsm8_pkg`.

```
package fsm8_pkg;
  typedef enum logic [3:0] {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,
                            XX  = 'x     } state_e;
endpackage
```

Example 25 - File: fsm8_pkg_b.sv - fsm8_pkg enumerated typedef

These packages include a typedef for the `state_e` type that is used by all of the `fsm8` designs.

Using the same package name makes it possible to read the separate package files first in a simulation and first in synthesis compilation to selectively choose abstract or binary encoded enumerated types without touching the `fsm8` files.

Appendix 2.12 **FSM8 - 1 always block RTL - NOT Recommended - Registered Outputs**

```
module fsm8_1x (
  output logic y1, y2, y3,
  input  logic jmp, go, sk0, sk1, clk, rst_n);

  import fsm8_pkg::*;
  state_e state;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      y1 <= '0;
      y2 <= '0;
      y3 <= '0;
                           state <= S0;
    end
    else begin
      y1 <= '0;
      y2 <= '0;
      y3 <= '0;
                           state <= XX; //@ LB
      case (state)
        S0:  if      ( go &&  jmp) begin
               y1 <= '1;
               y2 <= '1;
                           state <= S3;
             end
             else if ( go && !jmp) begin
               y2 <= '1;
                           state <= S1;
             end
             else          state <= S0; //@ LB
        S1 : if (jmp) begin
               y1 <= '1;
               y2 <= '1;
                           state <= S3;
             end
             else          state <= S2;
        S2 : if (jmp) begin
               y1 <= '1;
               y2 <= '1;
                           state <= S3;
             end
             else begin
               y1 <= '1;
               y2 <= '1;
               y3 <= '1;
                           state <= S9;
             end
        S3:  if (!jmp)     state <= S4;
             else begin
               y1 <= '1;
               y2 <= '1;
                           state <= S3; //@ LB
             end
        S4 : if (jmp) begin
               y1 <= '1;
```

```
           y2 <= '1;
                        state <= S3;
         end
         else if (sk0 && !jmp) begin
           y1 <= '1;
           y2 <= '1;
           y3 <= '1;
                        state <= S6;
         end
         else           state <= S5;
    S5 : if (jmp) begin
           y1 <= '1;
           y2 <= '1;
                        state <= S3;
         end
         else if (!sk1 && !sk0 && !jmp) begin
           y1 <= '1;
           y2 <= '1;
           y3 <= '1;
                        state <= S6;
         end
         else if (!sk1 && sk0 && !jmp) begin
           y3 <= '1;
                        state <= S7;
         end
         else if (sk1 && !sk0 && !jmp) begin
           y2 <= '1;
           y3 <= '1;
                        state <= S8;
         end
         else begin
           y1 <= '1;
           y2 <= '1;
           y3 <= '1;
                        state <= S9;
         end
    S6 : if (jmp) begin
           y1 <= '1;
           y2 <= '1;
                        state <= S3;
         end
         else if (go && !jmp) begin
           y3 <= '1;
                        state <= S7;
         end
         else begin
           y1 <= '1;
           y2 <= '1;
           y3 <= '1;
                        state <= S6; //@ LB
         end
    S7 : if (jmp) begin
           y1 <= '1;
           y2 <= '1;
                        state <= S3;
         end
         else begin
           y2 <= '1;
```

```
               y3 <= '1;
                           state <= S8;
             end
        S8 : if (jmp) begin
               y1 <= '1;
               y2 <= '1;
                           state <= S3;
             end
             else begin
               y1 <= '1;
               y2 <= '1;
               y3 <= '1;
                           state <= S9;
             end
        S9 : if (jmp) begin
               y1 <= '1;
               y2 <= '1;
                           state <= S3;
             end
             else          state <= S0;
        default: begin
               y1 <= 'x;
               y2 <= 'x;
               y3 <= 'x;
                           state <= XX;
             end
      endcase
    end
endmodule
```

Example 26 - fsm8_1x - 1-always block style with registered outputs

Appendix 2.13 **FSM8 - 2 always block RTL - Recommended - Combinatorial Outputs**

```
module fsm8_2x (
  output logic y1, y2, y3,
  input  logic jmp, go, sk0, sk1, clk, rst_n);

  import fsm8_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                           //@LB next = state;
    y1 = '0;
    y2 = '0;
    y3 = '0;
    case (state)
      S0:  if      ( go &&  jmp)         next = S3;
           else if ( go && !jmp)         next = S1;
           else                          next = S0; //@ LB
      S1 : begin
             y2 = '1;
             if (jmp)                    next = S3;
             else                        next = S2;
           end
      S2 : if (jmp)                      next = S3;
           else                          next = S9;
      S3 : begin
             y1 = '1;
             y2 = '1;
             if (!jmp)                   next = S4;
             else                        next = S3; //@ LB
           end
      S4 : if      (jmp)                 next = S3;
           else if (sk0 && !jmp)         next = S6;
           else                          next = S5;
      S5 : if      (jmp)                 next = S3;
           else if (!sk1 && !sk0 && !jmp) next = S6;
           else if (!sk1 &&  sk0 && !jmp) next = S7;
           else if ( sk1 && !sk0 && !jmp) next = S8;
           else                          next = S9;
      S6 : begin
             y1 = '1;
             y2 = '1;
             y3 = '1;
             if      (jmp)               next = S3;
             else if (go && !jmp)        next = S7;
             else                        next = S6; //@ LB
           end
      S7 : begin
             y3 = '1;
             if (jmp)                    next = S3;
             else                        next = S8;
           end
      S8 : begin
```

```
                y2 = '1;
                y3 = '1;
                if (jmp)                        next = S3;
                else                            next = S9;
              end
          S9 : begin
                y1 = '1;
                y2 = '1;
                y3 = '1;
                if (jmp)                        next = S3;
                else                            next = S0;
              end
          default: begin
                y1  = 'x;
                y2  = 'x;
                y3  = 'x;
                                                next = XX;
              end
        endcase
      end
    endmodule
```

Example 27 - fsm8_2x - 2-always block style with combinatorial outputs

Appendix 2.14 **FSM8 - 3 always block - Recommended - Registered Outputs**

```
module fsm8_3x (
  output logic y1, y2, y3,
  input  logic jmp, go, sk0, sk1, clk, rst_n);

  import fsm8_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                                //@LB next = state;
    case (state)
      S0:  if      ( go &&  jmp)              next = S3;
           else if ( go && !jmp)              next = S1;
           else                               next = S0; //@ LB
      S1 : if ( jmp)                          next = S3;
           else                               next = S2;
      S2 : if ( jmp)                          next = S3;
           else                               next = S9;
      S3 : if (!jmp)                          next = S4;
           else                               next = S3; //@ LB
      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; //@ LB
      S7 : if ( jmp)                          next = S3;
           else                               next = S8;
      S8 : if ( jmp)                          next = S3;
           else                               next = S9;
      S9 : if ( jmp)                          next = S3;
           else                               next = S0;
      default:                                next = XX;
    endcase
  end

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      y1 <= '0;
      y2 <= '0;
      y3 <= '0;
    end
    else begin
      y1 <= '0;
      y2 <= '0;
      y3 <= '0;
      case (next)
```

```
          S7     :   y3 <= '1;
          S1     :   y2 <= '1;
          S3     : begin
                     y1 <= '1;
                     y2 <= '1;
                   end
          S8     : begin
                     y2 <= '1;
                     y3 <= '1;
                   end
          S6, S9 : begin
                     y1 <= '1;
                     y2 <= '1;
                     y3 <= '1;
                   end
      endcase
    end
  endmodule
```

Example 28 - fsm8_3x - 3-always block style with registered outputs

Appendix 2.15 **FSM8 - 4 always block RTL - Recommended - Registered Outputs**

```
module fsm8_4x (
  output logic y1, y2, y3,
  input  logic jmp, go, sk0, sk1, clk, rst_n);

  logic a1, a2, a3; // next combinatorial outputs

  import fsm8_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                               //@LB next = state;
    case (state)
      S0:  if       ( go &&  jmp)            next = S3;
           else if  ( go && !jmp)            next = S1;
           else                              next = S0; //@ LB
      S1 : if ( jmp)                         next = S3;
           else                              next = S2;
      S2 : if ( jmp)                         next = S3;
           else                              next = S9;
      S3 : if (!jmp)                         next = S4;
           else                              next = S3; //@ LB
      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; //@ LB
      S7 : if ( jmp)                         next = S3;
           else                              next = S8;
      S8 : if ( jmp)                         next = S3;
           else                              next = S9;
      S9 : if ( jmp)                         next = S3;
           else                              next = S0;
      default:                               next = XX;
    endcase
  end

  always_comb begin
    {a3,a2,a1} = '0;
    case (state)
      S0 : if       (go &&  jmp)             {   a2,a1} = '1; // S3
           else if  (go && !jmp)             {   a2   } = '1; // S1
           else ;                                             // S0

      S1 : if (jmp)                          {   a2,a1} = '1; // S3
           else ;                                             // S2
      S2 : if (jmp)                         {   a2,a1} = '1; // S3
           else                             {a3,a2,a1} = '1; // S9

      S3 : if (!jmp) ;                                       // S4
           else                             {   a2,a1} = '1; // S3

      S4 : if ( jmp)                        {   a2,a1} = '1; // S3
           else if (sk0 && !jmp)            {a3,a2,a1} = '1; // S6
           else ;                                            // S5

      S5 : if ( jmp)                        {   a2,a1} = '1; // S3
           else if (!sk1 && !sk0 && !jmp) {a3,a2,a1} = '1; // S6
           else if (!sk1 &&  sk0 && !jmp) {a3      } = '1; // S7
           else if ( sk1 && !sk0 && !jmp) {a3,a2   } = '1; // S8
           else                             {a3,a2,a1} = '1; // S9

      S6 : if      (jmp)                    {   a2,a1} = '1; // S3
           else if (go && !jmp)             {a3      } = '1; // S7
           else                             {a3,a2,a1} = '1; // S6

      S7 : if ( jmp)                        {   a2,a1} = '1; // S3
           else                             {a3,a2   } = '1; // S8

      S8 : if ( jmp)                        {   a2,a1} = '1; // S3
           else                             {a3,a2,a1} = '1; // S9

      S9 : if ( jmp)                        {   a2,a1} = '1; // S3
           else ;                                            // S0
      default:                              {a3,a2,a1} = 'x;
    endcase
  end

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
      y1 <= '0;
      y2 <= '0;
      y3 <= '0;
    end
    else begin
      y1 <= a1;
      y2 <= a2;
      y3 <= a3;
    end
endmodule
```

Example 29 - fsm8_4x - 4-always block style with registered outputs

Appendix 2.16 PREP4 - prep4_pkgs for abstract and binary encoded enums

The [a]bstract enumerated types package file is named `prep4_pkg_a.sv` but the package name is `prep4_pkg`.

```
package prep4_pkg;
  typedef enum {S0,
                S1,
                S2,
                S3,
                S4,
                S5,
                S6,
                S7,
                S8,
                S9,
                S10,
                S11,
                S12,
                S13,
                S14,
                S15,
                XX  } state_e;
endpackage
```

Example 30 - File: prep4_pkg_a.sv - prep4_pkg enumerated typedef

The [b]inary encoded enumerated types package file is named `prep4_pkg_b.sv` but the package name is still `prep4_pkg`.

```
package prep4_pkg;
  typedef enum logic [3:0] {S0  = 4'b0000,
                            S1  = 4'b0100,
                            S2  = 4'b0101,
                            S3  = 4'b0001,
                            S4  = 4'b1011,
                            S5  = 4'b1001,
                            S6  = 4'b0010,
                            S7  = 4'b0011,
                            S8  = 4'b0110,
                            S9  = 4'b1111,
                            S10 = 4'b1101,
                            S11 = 4'b0111,
                            S12 = 4'b1010,
                            S13 = 4'b1000,
                            S14 = 4'b1110,
                            S15 = 4'b1100,
                            XX  = 'x     } state_e;
endpackage
```

Example 31 - File: prep4_pkg_b.sv - prep4_pkg enumerated typedef

These packages include a typedef for the `state_e` type that is used by all of the `prep4` designs.

Using the same package name makes it possible to read the separate package files first in a simulation and first in synthesis compilation to selectively choose abstract or binary encoded enumerated types without touching the `prep4` files.

Appendix 2.17 **PREP4 - 1 always block RTL - NOT Recommended - Registered Outputs**

```
module prep4_1x (
  output logic [7:0] out,
  input  logic [7:0] in,
  input  logic       clk, rst_n);

  import prep4_pkg::*;
  state_e state;

  always_ff @(posedge clk, negedge rst_n)
    if (!rst_n) begin
                                               state <= S0;
      out <= 8'h00;
    end
    else begin
      state <= XX;                                           //@ LB
      out   <= 'x;
      case (state)
        S0 : if (in) begin
               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
             end
             else begin
               out <= 8'h00;
                                               state <= S0;  //@ LB
             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;  //@ LB
        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;  //@ LB
        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;  //@ LB
        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;  //@ LB
```

```
     end
S9 : if (in[0]) begin
       out <= 8'hFF;
                                       state <= S11;
     end
     else begin
       out <= 8'hFF;
                                       state <= S9;  //@ LB
     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; //@ LB
     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) begin
       if (in < 64) begin
         out <= 8'hFD;
                                       state <= S12;
       end
       else begin
         out <= 8'hFF;
                                       state <= S10;
       end
     end
     else begin
       out <= 8'hDF;
                                       state <= S14; //@ LB
     end
S15: if (in[7]) begin
       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
             end
             else begin
               out <= 8'h7F;
                                               state <= S15; //@ LB
             end
         default: begin
               out   <= 'x;
                                               state <= XX;
             end
      endcase
    end
endmodule
```

Example 32 - prep4_1x - 1-always block style with registered outputs

Appendix 2.18 **PREP4 - 2 always block RTL - Recommended - Combinatorial Outputs**

```
module prep4_2x (
  output logic [7:0] out,
  input  logic [7:0] in,
  input  logic       clk, rst_n);

  import prep4_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                                      //@LB next = state;
    out  = 'x;
    case (state)
      S0 : begin
             out = 8'h00;
             if (in) begin
               if      (in <  4)                    next = S1;
               else if (in < 32)                    next = S2;
               else if (in < 64)                    next = S3;
               else                                 next = S4;
             end
           else                                     next = S0;   //@ LB
           end
      S1 : begin
             out = 8'h06;
             if      (in[0] && in[1])               next = S0;
             else                                   next = S3;
           end
      S2 : begin
             out = 8'h18;
                                                    next = S3;
           end
      S3 : begin
             out = 8'h60;
                                                    next = S5;
           end
      S4 : begin
             out = 8'h80;
             if      (in[0] || in[2] || in[4])      next = S5;
             else                                   next = S6;
           end
      S5 : begin
             out = 8'hF0;
             if      (in[0])                        next = S7;
             else                                   next = S5;   //@ LB
           end
      S6:  begin
             out = 8'h1F;
             if      ( in[6] &  in[7])              next = S1;
             else if (!in[6] &  in[7])              next = S9;
             else if ( in[6] & !in[7])              next = S8;
             else                                   next = S6;   //@ LB
```

```
     end
S7 : begin
       out = 8'h3F;
       if      ( in[6] &  in[7])         next = S4;
       else if (!in[6] & !in[7])         next = S3;
       else                              next = S7;  //@ LB
     end
S8 : begin
       out = 8'h7F;
       if (in[4] ^ in[5])                next = S11;
       else if (in[7])                   next = S1;
       else                              next = S8;  //@ LB
     end
S9 : begin
       out = 8'hFF;
       if (in[0])                        next = S11;
       else                              next = S9;  //@ LB
     end
S10: begin
       out = 8'hFF;
                                         next = S1;
     end
S11: begin
       out = 8'hFF;
       if      (in == 64)                next = S15;
       else                              next = S8;
     end
S12: begin
       out = 8'hFD;
       if      (in == 255)               next = S0;
       else                              next = S12;  //@ LB
     end
S13: begin
       out = 8'hF7;
       if      (in[5] ^ in[3] ^ in[1])   next = S12;
       else                              next = S14;
       end
S14: begin
       out = 8'hDF;
       if (in) begin
         if (in < 64)                    next = S12;
         else                            next = S10;
       end
       else                              next = S14; //@ LB
     end
S15: begin
       out = 8'h7F;
       if (in[7]) begin
         case (in[1:0])
           2'b00:                        next = S14;
           2'b01:                        next = S10;
           2'b10:                        next = S13;
           2'b11:                        next = S0;
         endcase
       end
       else                              next = S15; //@ LB
     end
default : begin
```

```
            out = 'x;                                  next = XX;
          end
      endcase
    end
  endmodule
```

Example 33 - prep4_2x - 2-always block style with combinatorial outputs

Appendix 2.19 **PREP4 - 3 always block - Recommended - Registered Outputs**

```
module prep4_3x (
  output logic [7:0] out,
  input  logic [7:0] in,
  input  logic       clk, rst_n);

  import prep4_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                              //@LB next = state;
    case (state)
      S0 : if (in) begin
             if      (in <  4)              next = S1;
             else if (in < 32)              next = S2;
             else if (in < 64)              next = S3;
             else                           next = S4;
           end
           else                             next = S0;  //@ LB
      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 = S7;
           else                             next = S5;  //@ LB
      S6:  if      ( in[6] &  in[7])        next = S1;
           else if (!in[6] &  in[7])        next = S9;
           else if ( in[6] & !in[7])        next = S8;
           else                             next = S6;  //@ LB
      S7 : if      ( in[6] &  in[7])        next = S4;
           else if (!in[6] & !in[7])        next = S3;
           else                             next = S7;  //@ LB
      S8 : if (in[4] ^ in[5])               next = S11;
           else if (in[7])                  next = S1;
           else                             next = S8;  //@ LB
      S9 : if (in[0])                       next = S11;
           else                             next = S9;  //@ LB
      S10:                                  next = S1;
      S11: if      (in == 64)               next = S15;
           else                             next = S8;
      S12: if      (in == 255)              next = S0;
           else                             next = S12;  //@ LB
      S13: if      (in[5] ^ in[3] ^ in[1])  next = S12;
           else                             next = S14;
      S14: if (in) begin
             if (in < 64)                   next = S12;
             else                           next = S10;
           end
           else                             next = S14; //@ LB
      S15: if (in[7]) begin
```

```
              case (in[1:0])
                2'b00:                          next = S14;
                2'b01:                          next = S10;
                2'b10:                          next = S13;
                2'b11:                          next = S0;
              endcase
            end
            else                                next = S15; //@ LB
        default :                               next = XX;
      endcase
    end

    always_ff @(posedge clk, negedge rst_n)
      if (!rst_n) out <= 8'h00;
      else begin
        out <= 'x;
        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;
          default:  out <= 'x;
        endcase
      end
  endmodule
```

Example 34 - prep4_3x - 3-always block style with registered outputs

Appendix 2.20 **PREP4 - 4 always block RTL - Recommended - Registered Outputs**

```
module prep4_4x (
  output logic [7:0] out,
  input  logic [7:0] in,
  input  logic       clk, rst_n);

  logic [7:0] n_out; // next combinatorial outputs

  import prep4_pkg::*;
  state_e state, next;

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

  always_comb begin
    next = XX;                               //@LB next = state;
    case (state)
      S0 : if (in) begin
             if      (in <  4)               next = S1;
             else if (in < 32)               next = S2;
             else if (in < 64)               next = S3;
             else                            next = S4;
           end
           else                              next = S0;  //@ LB
      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 = S7;
           else                              next = S5;  //@ LB
      S6:  if      ( in[6] &  in[7])         next = S1;
           else if (!in[6] &  in[7])         next = S9;
           else if ( in[6] & !in[7])         next = S8;
           else                              next = S6;  //@ LB
      S7 : if      ( in[6] &  in[7])         next = S4;
           else if (!in[6] & !in[7])         next = S3;
           else                              next = S7;  //@ LB
      S8 : if (in[4] ^ in[5])                next = S11;
           else if (in[7])                   next = S1;
           else                              next = S8;  //@ LB
      S9 : if (in[0])                        next = S11;
           else                              next = S9;  //@ LB
      S10:                                   next = S1;
      S11: if      (in == 64)                next = S15;
           else                              next = S8;
      S12: if      (in == 255)               next = S0;
           else                              next = S12;  //@ LB
      S13: if      (in[5] ^ in[3] ^ in[1])   next = S12;
           else                              next = S14;
      S14: if (in) begin
             if (in < 64)                    next = S12;
             else                            next = S10;
           end
```

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

     always_comb begin
       n_out = 8'h00;
       case (state)
         S0 : if (in) begin
                if      (in <  4)                 n_out = 8'h06; // S1
                else if (in < 32)                 n_out = 8'h18; // S2
                else if (in < 64)                 n_out = 8'h60; // S3
                else                              n_out = 8'h80; // S4
              end
              else                                n_out = 8'h00; //@ LB
         S1 : if      (in[0] && in[1])            n_out = 8'h00; // S0
              else                                n_out = 8'h60; // S3
         S2 :                                     n_out = 8'h60; // S3
         S3 :                                     n_out = 8'hF0; // S5
         S4 : if      (in[0] || in[2] || in[4])   n_out = 8'hF0; // S5
              else                                n_out = 8'h1F; // S6
         S5 : if      (in[0])                     n_out = 8'h3F; // S7
              else                                n_out = 8'hF0; // S5
         S6:  if      ( in[6] &  in[7])           n_out = 8'h06; // S1
              else if (!in[6] &  in[7])           n_out = 8'hFF; // S9
              else if ( in[6] & !in[7])           n_out = 8'h7F; // S8
              else                                n_out = 8'h1F; // S6
         S7 : if      ( in[6] &  in[7])           n_out = 8'h80; // S4
              else if (!in[6] & !in[7])           n_out = 8'h60; // S3
              else                                n_out = 8'h3F; // S7
         S8 : if (in[4] ^ in[5])                  n_out = 8'hFF; // S11
              else if (in[7])                     n_out = 8'h06; // S1
              else                                n_out = 8'h7F; // S8
         S9 : if (in[0])                          n_out = 8'hFF; // S11
              else                                n_out = 8'hFF; // S9
         S10:                                     n_out = 8'h06; // S1
         S11: if      (in == 64)                  n_out = 8'h7F; // S15
              else                                n_out = 8'h7F; // S8
         S12: if      (in == 255)                 n_out = 8'h00; // S0
              else                                n_out = 8'hFD; // S12
         S13: if      (in[5] ^ in[3] ^ in[1])     n_out = 8'hFD; // S12
              else                                n_out = 8'hDF; // S14
         S14: if (in) begin
                if (in < 64)                      n_out = 8'hFD; // S12
                else                              n_out = 8'hFF; // S10
              end
              else                                n_out = 8'hDF; // S14
         S15: if (in[7]) begin
                case (in[1:0])
```

```
                 2'b00:                         n_out = 8'hDF; // S14
                 2'b01:                         n_out = 8'hFF; // S10
                 2'b10:                         n_out = 8'hF7; // S13
                 2'b11:                         n_out = 8'h00; // S0
               endcase
             end
             else                               n_out = 8'h7F; // S15
        default :                               n_out = 'x;
      endcase
    end

    always_ff @(posedge clk, negedge rst_n)
      if (!rst_n) out <= 8'h00;
      else        out <= n_out;
  endmodule
```

Example 35 - prep4_4x - 4-always block style with registered outputs