Draft 3 - page 32 - Example FSM1

At the end of this document I have drawn conclusions based on my experiences experimenting with the new SystemVerilog state machine syntax. Based on those conclusions, I propose the following:

Proposal: Remove state-types, transition, endtransition, and the ->> token from SystemVerilog. I do not see a compelling argument to use or keep them in the language.

I spent two full days trying different coding styles in an effort to embrace the new FSM coding style but in the end I found myself frustrated with the new syntax. Below is the review I made and the process I went through to lead me to making the above proposal:

One question I have failed to recognize until now is the question of legacy Verilog FSM code. Many of these designs define the variable "state." This will now be a syntax error using SystemVerilog. This will impact a huge percentage of existing FSM designs (including every FSM design I have ever coded - I routinely use the identifiers state and next). When we created new keywords in Verilog-2001, we always considered how likely it was whether the new keyword existed in many designs or not.

Could I suggest using only one "input" keyword in each of the FSM examples, per the Verilog-2001 enhancement?

```
module FSM3(output found_101, input serial, clock, reset);
```

The always_comb used with this syntax will be quite confusing. The casual observer might not recognize that the reset is synchronous to the posedge clk because the reset assignment takes place inside of an always_comb block. (I guess I should not complain. It just gives me one more important bullet to highlight in a training class!)

Upon closer examination, I am confused by the always_comb statement. If I understand correctly, the always_comb is sensitive to changes on the variables: reset, state variable S and serial. If I am not mistaken, if reset is not asserted and if the current state is S1, then if the serial input goes high, found_101 will be set high (Mealy output). Now permit reset to go high, again triggering the always_comb block but the "if (reset)" statement will halt execution of the always_comb block until the next clock edge (??) to permit synchronous reset-state assignment (??). And if that is true, what happens if the reset signal goes both high and 2ns later goes low between clock edges? Will the always_block wait on the first reset-high signal and make the reset assignment (which would be wrong) or does it schedule the reset-state assignment, return to the top of the always_comb block, recognize the reset changing to low and asynchronously make a transition state assignment? (which also seems wrong) and complete the first scheduled reset assignment (also wrong)? This mixing of synchronous and asynchronous activity in the same always_comb block seems to be both confusing and dangerous. Thoughts?

In the same example, the reset assignment uses the format: ->> S.S0
while the other next state transitions drop the leading "S." characters and take the form:
->> S2
->> S1 ... ->> S0
->> S0

I believe the formats are interchangeable, but I don't think the formats had been explained prior to the FSM1 example. I suggest changing the first assignment to ->> S0 to avoid confusion at this point in the spec.

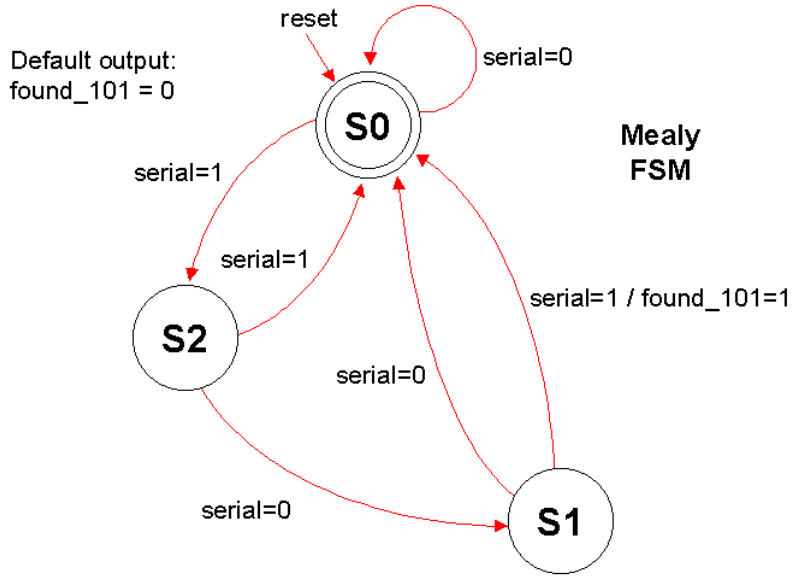
I believe there is a missing semi-colon in the third transition statement:
S1: ->> S0 ; if (serial ...
and do transition-item statements require begin-end for multiple statements (like Verilog case-item statements) or are transition-item statement blocks begin-end free (like VHDL case statements)? The BNF indicates it is a statement_or_null, meaning that the begin-end pair are required.

For readability, I also suggest the following formatting changes for the transition statement example:

```
// display this code with a mono-spaced font
else transition (S)
  S0: if( serial) ->> S2;
  S2: if(!serial) ->> S1;
      else        ->> S0;
  S1: begin
                  ->> S0;
        if( serial) found_101 = 1;
      end
endtransition
```

In the above re-formatted example, I suggest using the more Verilog-like Boolean tests as opposed to the more VHDL-like equal comparisons, and to line up the state transitions in the same column for readability. Lining up the next state assignments makes the code more readable, even for Verilog designs. It makes it easier to scan down the code to see the transitions in a nice orderly column as opposed to following the contour of the code to find the transition assignments.

Figure 9-1 should also point out that the input defined for all transition arcs is a signal named "serial." At present, one must guess that information from the code.



The output port declaration for FSM1 is also wrong, unless Peter added my request to automatically infer 1-bit reg variables (please, please, please!!!)

Similar formatting changes for readability should be made to the FSM2 and FSM3 examples.

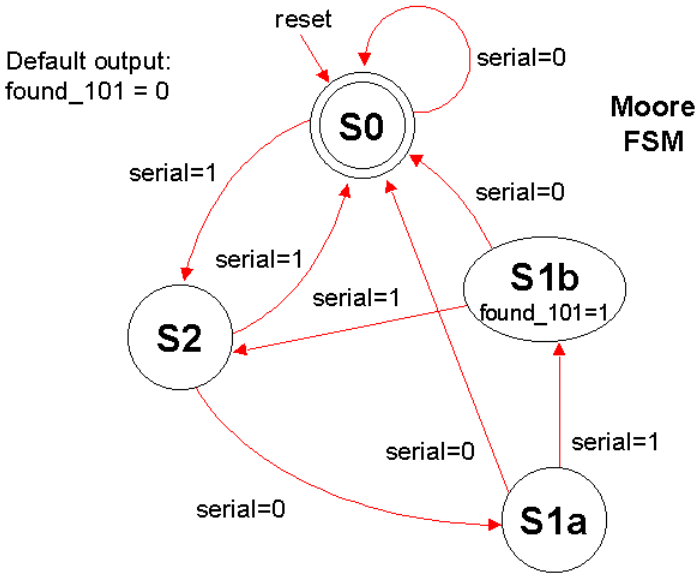
```
module FSM3 (output logic found_101, input serial, clock, reset);
  state {S0, S1, S2} S #1ns; // transition is nonblocking with delay

  always @(posedge clock or posedge reset) // asynchronous reset
    if (reset) ->> S0;
    else transition (S)
           S0: if ( serial) ->> S2;
           S2: if (!serial) ->> S1;
               else         ->> S0;
           S1:              ->> S0;
    endtransition

  always_comb
    if (S1 && serial) found_101 = 1;
    else              found_101 = 0;
endmodule
```

One of the most popular FSM coding styles in Verilog is to use two always blocks, one for the sequential flip-flops and one for all of the combinational logic (Steve Golson and I, who have written almost all of the technical papers on synthesizable FSM coding styles, both prefer and recommend the two always block method). This appears to be problematic with the new FSM syntax.

```
module FSM3a (output logic found_101, input serial, clock, reset);

  // the BNF prohibits listing multiple identifiers for a
  // state_type declaration, as shown below. Adding #1 to
  // this type of declaration would also raise questions
  // about which variable(s) include the delay
  state {S0, S1, S2} S, N; // transition is nonblocking

  always @(posedge clock or posedge reset) // asynchronous reset
    if (reset) ->> S.S0;
    else       ->> S.N; // the current value of the next state?

  always_comb begin // @(S, serial)
    found_101 = 0;
    transition (S)
      S0: if ( serial) ->> N.S2;
      S2: if (!serial) ->> N.S1;
          else         ->> N.S0;
      S1: begin
                       ->> N.S0;
            if (serial) found_101 = 1;
          end
    endtransition
  end
endmodule
```

The above example, if eventually permitted, should include an explanation of the S.S0 and N.S syntax before it is introduced. The above example cleanly separates the combinational logic from the sequential logic.

Consider a Moore example of the same FSM (requires 4 states - 2 states to replace S1 and the Mealy output).




```
module FSM3b (output logic found_101, input serial, clock, reset);
  state {S0, S1a, S1b, S2} S; // transition is nonblocking with delay

  always @(posedge clock or posedge reset) // asynchronous reset
    if (reset) begin
      ->> S0;
      found_101 = 0;
    end
    else begin
      found_101 = 0;
      transition (S)
        S0 : if ( serial) ->> S2;
        S2 : if (!serial) ->> S1a;
             else         ->> S0;
        S1a: if ( serial) begin
                           ->> S1b;
               found_101 = 1;
             end
             else         ->> S0;
        S1b: if ( serial) ->> S2;
             else         ->> S0;
      endtransition
    end
endmodule
```

This is a simple example that shows that for a Moore FSM design using one sequential always block, the output assignment is not made in the same state where the output is assigned, but instead is made in all states that can transition to the state where the output changes. If you have a state machine with ten states, and a Moore output is only assigned in state S10, but the other nine states S1-S9 can all transition to state S10, the above coding style will require nine assignments to the output, as opposed to one assignment to the output using a two always block technique. This is more verbose and quite error prone.

When coding with the one always block sequential style, you have to ask yourself the question, "what will the output be when I transition to the next state" and make assignments accordingly. Translation: you are always making *next-state* output assignments, which can be quite error prone, especially with a real FSM design that includes a significant number of states and outputs. Even though the style works, I now only include it in my classes as reference material, not as a technique that I actively teach.

Next = X's

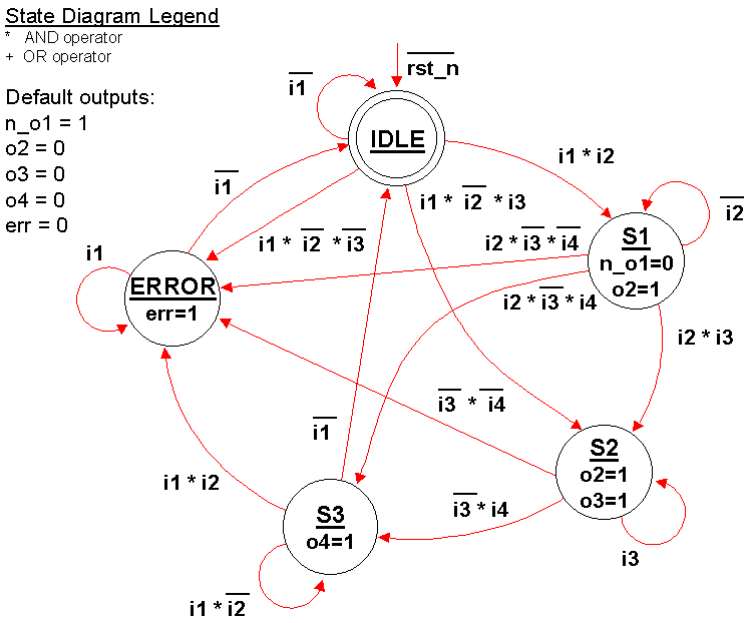
Another interesting technique that is frequently used by Verilog engineers is to make an assignment of all X's to the next state variable at the top of the combinational always block, immediately after the combinational sensitivity list. This trick was shown to me a number of years ago by Mike McNamara and Steve Golson.

Making X-assignments offers the following advantages:

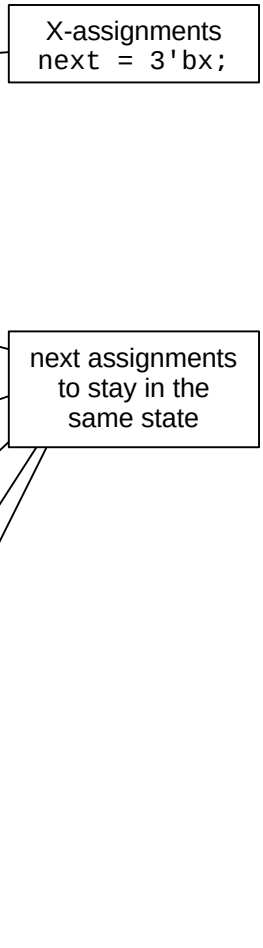
(1) It forces the engineer to code all next state assignments explicitly in the body of the combinational case statement. If an engineer forgets to code one of the next state assignments in the case statement, it will become very obvious when the code is simulated, because the next state will go unknown ("bleed red" in the waveform window) at the point where the assignment is missing. This helps to quickly identify design defects. Leaving the next state unchanged until assigned in the case statement can often obfuscate errors in the design, since the design will continue to go to valid states, even if a next state assignment is missing. The latter is more difficult to debug since the symptoms of the problem might not be manifest until multiple clock cycles after the missed transition. All of the examples in the SystemVerilog spec use the latter more error-prone coding style (which also is slightly less verbose, since no state transition is listed if the FSM stays in the same state).

(2) The X-assignment is treated as a "don't care" by synthesis tools, to help optimize the design. The X-assignment at the top of the always block removes state-latches from the design better than a full_case statement and also has basically the same effect as a full_case statement. It is also better than using a case default statement because the X-assignment happens every time the always block is entered while a case default is only executed if the case expression does not match one of the case items. If case default is used and if the case expression matches a case item that is missing a next state assignment, the previous next state assignment will not change but this might not be very apparent during simulation.

Consider the state diagram shown below. The Verilog code for this design is shown on the next page. The Verilog code uses the X-assignment tricks described above.



```
module fsm_cc0 (output reg err, n_o1, o2, o3, o4,
                input  i1, i2, i3, i4, clk, rst_n );

  parameter [2:0] IDLE = 3'b000,
                    S1 = 3'b001,
                    S2 = 3'b010,
                    S3 = 3'b011,
                 ERROR = 3'b111;

  reg [2:0]  state, next;

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

  // combinational always block
  // with combinational output assignments
  always @* begin
    next = 3'bx;
    err  = 0;
    n_o1 = 1;
    o2   = 0;
    o3   = 0;
    o4   = 0;
    case (state)
      IDLE:  begin
               if      (!i1) next = IDLE;
               else if ( i2) next = S1;
               else if ( i3) next = S2;
               else          next = ERROR;
             end
      S1:    begin
               n_o1 = 0;
               o2   = 1;
               if      (!i2) next = S1;
               else if ( i3) next = S2;
               else if ( i4) next = S3;
               else          next = ERROR;
             end
      S2:    begin
               o2 = 1;
               o3 = 1;
               if      ( i3) next = S2;
               else if ( i4) next = S3;
               else          next = ERROR;
             end
      S3:    begin
               o4 = 1;
               if      (!i1) next = IDLE;
               else if ( i2) next = ERROR;
               else          next = S3;
             end
      ERROR: begin
               err = 1;
               if (i1)       next = ERROR;
               else          next = IDLE;
             end
    endcase
  end
endmodule
```



The equivalent SystemVerilog code is shown on the next page. Notice that the new SystemVerilog coding style with X-assignments is still as verbose as normal Verilog code.

```
// Proposed System Verilog version of the fsm_cc0 design
module fsm_cc0sv1 (output reg err, n_o1, o2, o3, o4,
                   input   i1, i2, i3, i4, clk, rst_n );

  state [2:0] { IDLE = 3'b000,
                S1 = 3'b001,
                S2 = 3'b010,
                S3 = 3'b011,
              ERROR = 3'b111,
                  X = 3'bx   } S, N; // all X's permitted?

  // sequential always block
  always @(posedge clk or negedge rst_n)
    if (!rst_n) ->> S.IDLE;
    else        ->> S.N;

  // combinational always block
  // with combinational output assignments
  always_comb;
    ->> N.X; // simulation & synthesis trick
    err  = 0;
    n_o1 = 1;
    o2   = 0;
    o3   = 0;
    o4   = 0;
    transition (S)
      IDLE:  begin
               if      (!i1) ->> N.IDLE;
               else if ( i2) ->> N.S1;
               else if ( i3) ->> N.S2;
               else          ->> N.ERROR;
             end
      S1:    begin
               n_o1 = 0;
               o2   = 1;
               if      (!i2) ->> N.S1;
               else if ( i3) ->> N.S2;
               else if ( i4) ->> N.S3;
               else          ->> N.ERROR;
             end
      S2:    begin
               o2 = 1;
               o3 = 1;
               if      ( i3) ->> N.S2;
               else if ( i4) ->> N.S3;
               else          ->> N.ERROR;
             end
      S3:    begin
               o4 = 1;
               if      (!i1) ->> N.IDLE;
               else if ( i2) ->> N.ERROR;
               else          ->> N.S3;
             end
      ERROR: begin
               err = 1;
               if (i1)       ->> N.ERROR;
               else          ->> N.IDLE;
             end
    endcase
  end
endmodule
```

The only reasons that the new SystemVerilog coding styles in the Draft 3 document appear to be more code-efficient is because (1) the coding styles use the one-sequential-always block coding style (a more error prone coding style) and (2) the examples do not take advantage of the X-assignment technique. If a two always block coding style with X-next-assignment is coded, the code looks remarkably similar in both verbosity and content.

I coded the above model using enumerated types and it again appears to be rather equivalent. The code is on the next page:

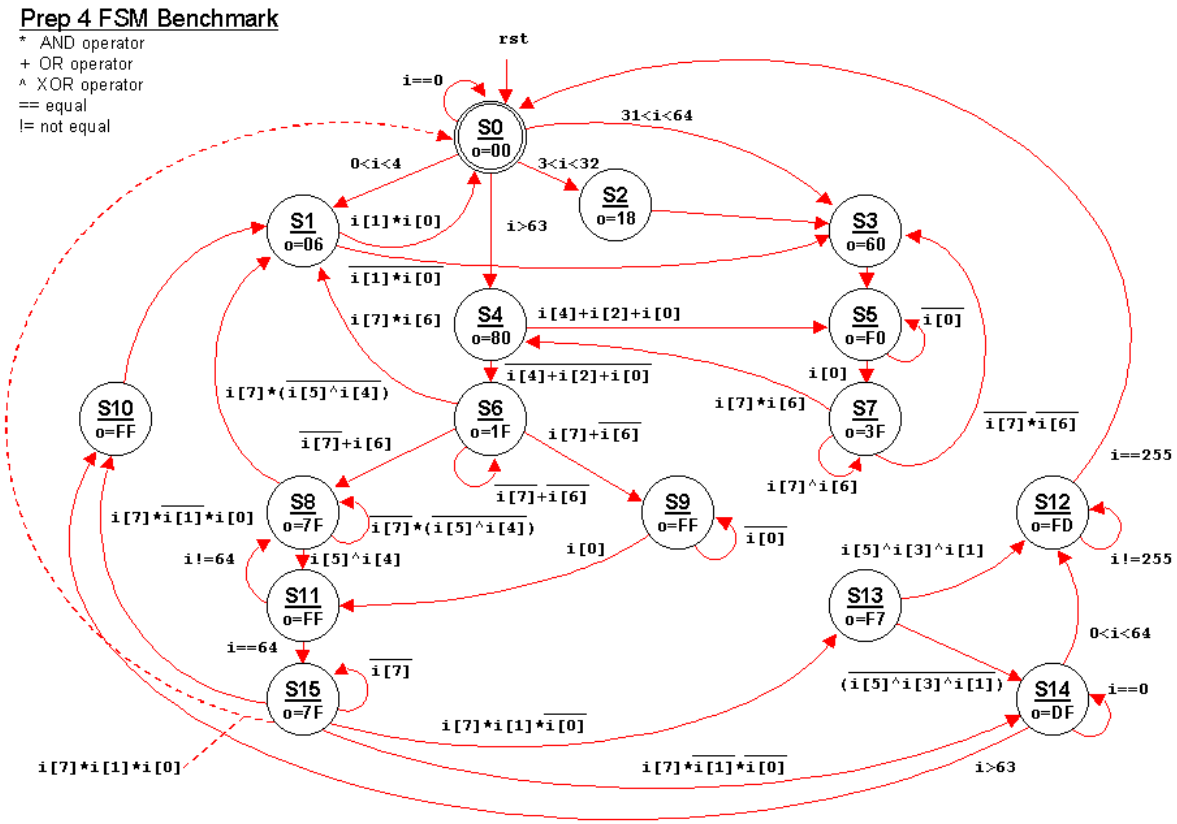
```
// Proposed SystemVerilog version of fsm_cc0 using enum
module fsm_cc0sv2 (output reg err, n_o1, o2, o3, o4,
                   input   i1, i2, i3, i4, clk, rst_n );

  enum [2:0] { IDLE = 3'b000,
                  S1 = 3'b001,
                  S2 = 3'b010,
                  S3 = 3'b011,
               ERROR = 3'b111,
                  XX = 3'bx   } state, next; // all X's permitted

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

  // combinational always block
  // with combinational output assignments
  always_comb;
    next = XX; // simulation & synthesis trick
    err  = 0;
    n_o1 = 1;
    o2   = 0;
    o3   = 0;
    o4   = 0;
    case (state)
      IDLE:  begin
               if      (!i1) next = IDLE;
               else if ( i2) next = S1;
               else if ( i3) next = S2;
               else          next = ERROR;
             end
      S1:    begin
               n_o1 = 0;
               o2   = 1;
               if      (!i2) next = S1;
               else if ( i3) next = S2;
               else if ( i4) next = S3;
               else          next = ERROR;
             end
      S2:    begin
               o2 = 1;
               o3 = 1;
               if      ( i3) next = S2;
               else if ( i4) next = S3;
               else          next = ERROR;
             end
      S3:    begin
               o4 = 1;
               if      (!i1) next = IDLE;
               else if ( i2) next = ERROR;
               else          next = S3;
             end
      ERROR: begin
               err = 1;
               if (i1)       next = ERROR;
               else          next = IDLE;
             end
    endcase
  end
endmodule
```

On this page is an efficient Verilog-2001 onehot coding style for the same FSM. On the next page is my attempt to code a SystemVerilog version of the same FSM using enumerated types. I had no clue on how to code the same FSM using state-types, transitions, etc.

```
module fsm_cc1 (output reg err, n_o1, o2, o3, o4,
                input   i1, i2, i3, i4, clk, rst_n );

  parameter  IDLE = 0,
               S1 = 1,
               S2 = 2,
               S3 = 3,
            ERROR = 4;

  reg [4:0]  state, next;

  // sequential always block
  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                  state       <= 5'b0;
                  state[IDLE] <= 1'b1;
                end
    else          state <= next;

  // combinational always block
  // with combinational output assignments
  always @(state or i1 or i2 or i3 or i4) begin
    next = 5'b0;
    err  = 0;
    n_o1 = 1;
    o2   = 0;
    o3   = 0;
    o4   = 0;
    (* synthesis, full_case, parallel_case *) case (1'b1)
      state[IDLE]:  begin
                      if      (!i1) next[IDLE]  = 1'b1;
                      else if ( i2) next[S1]    = 1'b1;
                      else if ( i3) next[S2]    = 1'b1;
                      else          next[ERROR] = 1'b1;
                    end
      state[S1]:    begin
                      n_o1 = 0;
                      o2   = 1;
                      if      (!i2) next[S1]    = 1'b1;
                      else if ( i3) next[S2]    = 1'b1;
                      else if ( i4) next[S3]    = 1'b1;
                      else          next[ERROR] = 1'b1;
                    end
      state[S2]:    begin
                      o2 = 1;
                      o3 = 1;
                      if      ( i3) next[S2]    = 1'b1;
                      else if ( i4) next[S3]    = 1'b1;
                      else          next[ERROR] = 1'b1;
                    end
      state[S3]:    begin
                      o4 = 1;
                      if      (!i1) next[IDLE]  = 1'b1;
                      else if ( i2) next[ERROR] = 1'b1;
                      else          next[S3]    = 1'b1;
                    end
      state[ERROR]: begin
                      err = 1;
                      if (i1)       next[ERROR] = 1'b1;
                      else          next[IDLE]  = 1'b1;
                    end
    endcase
  end
endmodule
```

```
// SystemVerilog version of the fsm_cc1 design
module fsm_cc1sv (output reg err, n_o1, o2, o3, o4,
                  input   i1, i2, i3, i4, clk, rst_n );

  enum [4:0] { IDLE = 0,
               S1 = 1,
               S2 = 2,
               S3 = 3,
            ERROR = 4 } index;
  // ?? Can the above enumerated types be used to index
  // into the following state and next variables??
  // If so, the following coding style works but does
  // not show state names in a waveform viewer
  reg  [4:0] state, next;

  // sequential always block
  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                  state       <= 5'b0;
                  state[IDLE] <= 1'b1;
                end
    else          state       <= next;

  // combinational always block
  // with combinational output assignments
  always_comb
    next = 5'b0;
    err  = 0;
    n_o1 = 1;
    o2   = 0;
    o3   = 0;
    o4   = 0;
    // Use "case" instead of "transition" to compare individual bits
    (* synthesis, full_case, parallel_case *) case (1'b1)
      state[IDLE]:  begin
                      if      (!i1) next[IDLE]  = 1'b1;
                      else if ( i2) next[S1]    = 1'b1;
                      else if ( i3) next[S2]    = 1'b1;
                      else          next[ERROR] = 1'b1;
                    end
      state[S1]:    begin
                      n_o1 = 0;
                      o2   = 1;
                      if      (!i2) ->> next[S1];
                      else if ( i3) ->> next[S2];
                      else if ( i4) ->> next[S3];
                      else          ->> next[ERROR];
                    end
      state[S2]:    begin
                      o2 = 1;
                      o3 = 1;
                      if      ( i3) ->> next[S2];
                      else if ( i4) ->> next[S3];
                      else          ->> next[ERROR];
                    end
      state[S3]:    begin
                      o4 = 1;
                      if      (!i1) ->> next[IDLE];
                      else if ( i2) ->> next[ERROR];
                      else          ->> next[S3];
                    end
      state[ERROR]: begin
                      err = 1;
                      if (i1)       ->> next[ERROR];
                      else          ->> next[IDLE];
                    end
    endcase
  end
endmodule
```

Below is amore interesting FSM design. For this design, I want to use encoded outputs, where the output bits will be equal to some of the state-encoding bits.



state	x1	x0	o(outputs)
S0	0	0	00
S1	0	0	06
S2	0	0	18
S3	0	0	60
S4	0	0	80
S5	0	0	f0
S6	0	0	1f
S7	0	0	3f
S8	0	0	7f
S9	0	0	ff
S10	0	1	ff
S11	1	0	ff
S12	0	0	fd
S13	0	0	f7
S14	0	0	df
S15	0	1	7f

Two extra state bits are required to make unique state assignments

The S8 outputs are used in states S8 and S15

The S9 outputs are used in states S9, S10 and S11

All other output assignments are unique

Basically, the technique is implemented by examining the output assignments for each state and then adding extra state bits (x1 and x0 in this example) to insure a unique encoding for each state while insuring the output assignments compose part of the state-encoding pattern.

This techniques reduces the number of flip-flops required to implement the design while registering the outputs as direct connections to the state bits. This is technique yields an efficient, fast FSM design with glitch-free outputs.

As can be seen from this example, in order to efficiently code this FSM design, the engineer needs to have control of and direct access to the state bits for each state.

```
// PREP4 contains a large state machine

module prep4_ffo (output [7:0] o, input [7:0] i, input clk, rst);
  // The S8 outputs are used in states S8 and S15
  // The S9 outputs are used in states S9, S10 and S11
  // Two extra state bits are required to make unique
  //   state assignments
  enum {S0  = 10'h0_00, // unique outputs
        S1  = 10'h0_06, // unique outputs
        S2  = 10'h0_18, // unique outputs
        S3  = 10'h0_60, // unique outputs
        S4  = 10'h0_80, // unique outputs
        S5  = 10'h0_f0, // unique outputs
        S6  = 10'h0_1f, // unique outputs
        S7  = 10'h0_3f, // unique outputs
        S8  = 10'h0_7f, // S8     outputs *
        S9  = 10'h0_ff, // S9     outputs +
        S10 = 10'h1_ff, // S9     outputs ++
        S11 = 10'h2_ff, // S9     outputs +++
        S12 = 10'h0_fd, // unique outputs
        S13 = 10'h0_f7, // unique outputs
        S14 = 10'h0_df, // unique outputs
        S15 = 10'h1_7f} // S8     outputs **
        state, next;

  always @ (posedge clk or posedge rst)
    if (rst) state <= S0;
    else     state <= next;

  always @* begin
    next = 10'bx;
    case (state)
      S0 : begin
             if      (   i == 0   )  next = S0;
             else if ( 0<i && i<4 )  next = S1;
             else if ( 3<i && i<32)  next = S2;
             else if (31<i && i<64)  next = S3;
             else                    next = S4;
           end

      S1 :   if      ( i[1]&i[0]  )  next = S0;
             else                    next = S3;

      S2 :                           next = S3;

      S3 :                           next = S5;

      S4 :   if (i[4] | i[2] | i[0]) next = S5;
             else                    next = S6;

      S5 :   if (!i[0])              next = S5;
             else                    next = S7;

      S6 :   case (i[7:6])
               2'b00:                next = S6;
               2'b01:                next = S8;
               2'b10:                next = S9;
               2'b11:                next = S1;
```

```
            endcase

        S7 :   case (i[7:6])
                 2'b00:                next = S3;
                 2'b11:                next = S4;
                 default:              next = S7;
               endcase

        S8 :   if (i[5] ^ i[4])         next = S11;
               else if (i[7])           next = S1;
               else                     next = S8;

        S9 :   if (i[0])                next = S11;
               else                     next = S9;

        S10:                            next = S1;

        S11:   if (i == 64)             next = S15;
               else                     next = S8;

        S12:   if (i == 255)            next = S0;
               else                     next = S12;

        S13:   if (i[5] ^ i[3] ^ i[1]) next = S12;
               else                     next = S14;

        S14:   if      (   i == 0   )   next = S14;
               else if ( 0<i && i<64)   next = S12;
               else                     next = S10;

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

    // The output assignments are simply
    // the 8 LSBs of the state variable
    assign o = state[7:0];
endmodule
```

Attempting to implement this FSM using the "state" data type with one sequential always block would be difficult and error prone. Keeping track of all of the "next output" assignments would be tricky. Notice how easy it is to code the registered outputs using the above technique.

Conclusions

In summary, I have pretty much talked myself out of using the new state-type / transition / ->> syntax. In the examples I have examined, I see no coding benefit over just using simple enumerated types with Verilog-2001. In fact, since I have to use the Verilog coding style to design an efficient onehot FSM or to design the above output-encoded style of FSM, I believe engineers are better off just learning the Verilog style with the enumerated-type capability. Based on my current understanding of the new syntax, if I were to teach a SystemVerilog class today, I would show the new syntax (for completeness) and then give the guideline to discard it in favor of the generally better suited Verilog-2001 coding style with enumerated types. I still do not have a good answer for enumerated-onehot FSM coding styles (it works but does not display correctly in a waveform viewer) and am open to proposals.

By adding a state-type to the SystemVerilog language, we are going to break 1,000's of existing FSM designs. At the very least, we should come up with a different keyword to implement this functionality.

Proposal: Remove state-types, transition, endtransition, and the ->> token from SystemVerilog. I do not see a compelling argument to use or keep them in the language.

Regards - Cliff Cummings