Modeling FSMs in SV

Modeling FSMs

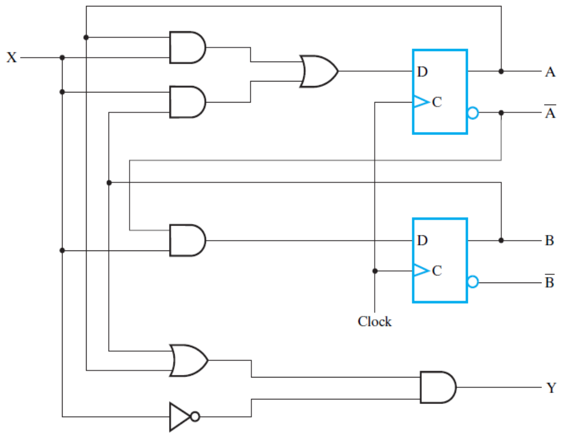
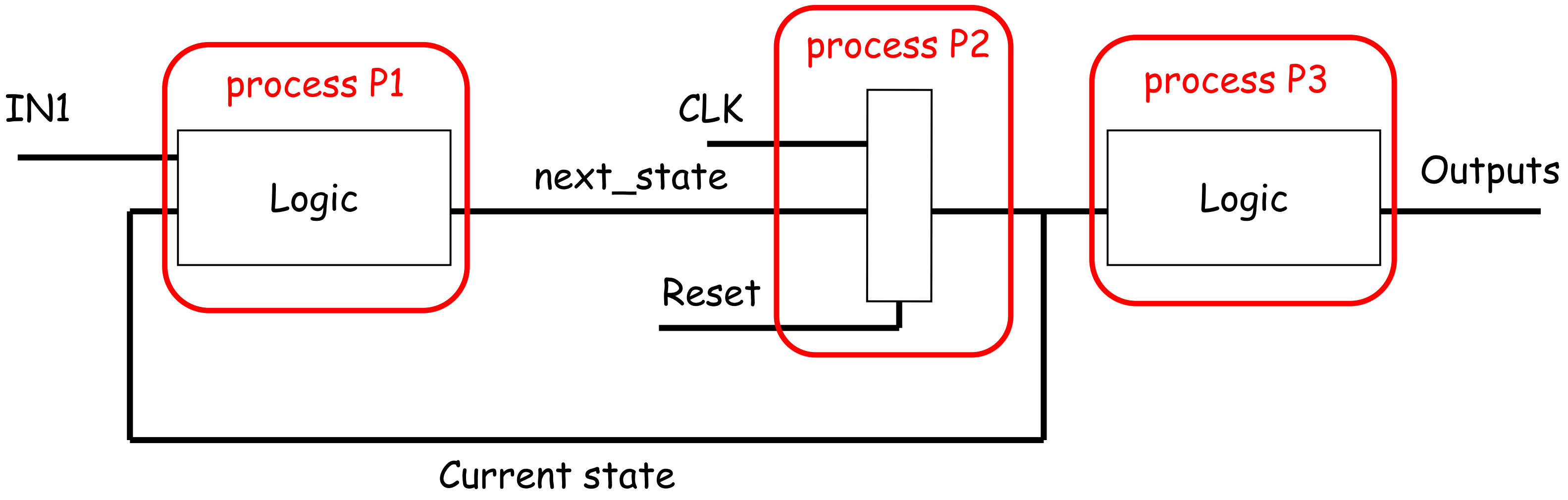


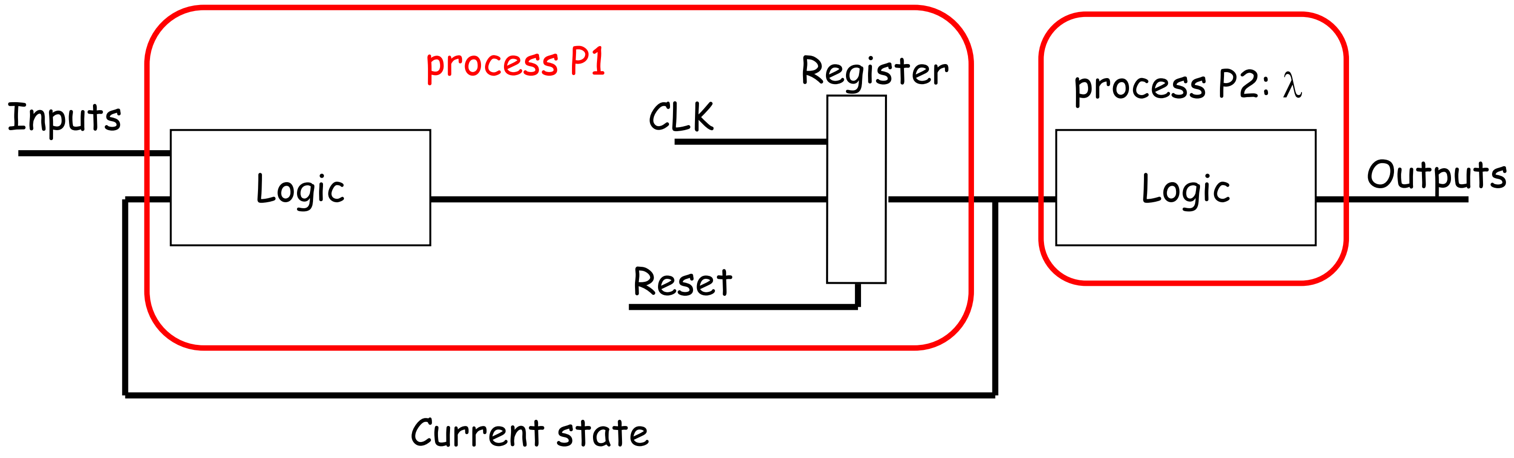
TABLE 5-1
State Table for Circuit of Figure 5-15

Present State		Input	Next State		Output
A	**B**	**X**	**A**	**B**	**Y**
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

- We need to describe the following objects.
  - States: enumerated types
  - Next state function (State transition table): case statement
  - Output function: case statement
  - Transfer from the Current State to Next State: always statement

Modeling Decisions: Moore FSM





Modeling States

- The state of the system must be held in an internal register.
- In SystemVerilog, the state can be represented by an *enumerated type*.
- The possible values of this type are the state names and
  - the name of the variable is given after the list of values,
  - e.g. **enum** {s0, s1, ...} state;

Moore FSM

```
module divideby3FSM(input logic clk,
                    input logic reset,
                    output logic y);
typedef enum logic [1:0] {S0, S1, S2} statetype;
statetype [1:0] state, nextstate;
// state register
always_ff @(posedge clk, posedge reset)
if (reset) state <= S0;
    else state <= nextstate;
// next state logic
always_comb
   case (state)
   S0: nextstate = S1;
   S1: nextstate = S2;
   S2: nextstate = S0;
   default: nextstate = S0;
endcase
// output logic
assign y = (state = = S0);
endmodule
```

- typedef defines statetype to be a two-bit logic value with S0, S1, or S2.
- state and nextstate are statetype signals.
- The enumerated encodings default to numerical order: S0 = 00, S1 = 01, and S2 = 10.
- The encodings can be explicitly set by the user; however, the synthesis tool views them as suggestions, not requirements.
- For example, 3-bit one-hot encoding: typedef enum logic [2:0] {S0 = 3'b001, S1 = 3'b010, S2 = 3'b100} statetype;

Moore FSM

```
module patternMoore(input logic clk,
                    input logic reset,
                    input logic a,
                    output logic y);
typedef enum logic [1:0] {S0, S1, S2}
statetype;
statetype state, nextstate;
// state register
always_ff @(posedge clk, posedge reset)
  if (reset) state <= S0;
  else state <= nextstate;
// next state logic
always_comb
   case (state)
     S0: if (a) nextstate = S0;
         else nextstate = S1;
     S1: if (a) nextstate = S2;
         else nextstate = S1;
     S2: if (a) nextstate = S0;
         else nextstate = S1;
     default: nextstate = S0;
  endcase
// output logic
assign y = (state = = S2);
endmodule
```

- Nonblocking assignments (<=) are used in the state register to describe sequential logic
- Blocking assignments (=) are used in the next state logic to describe combinational logic.

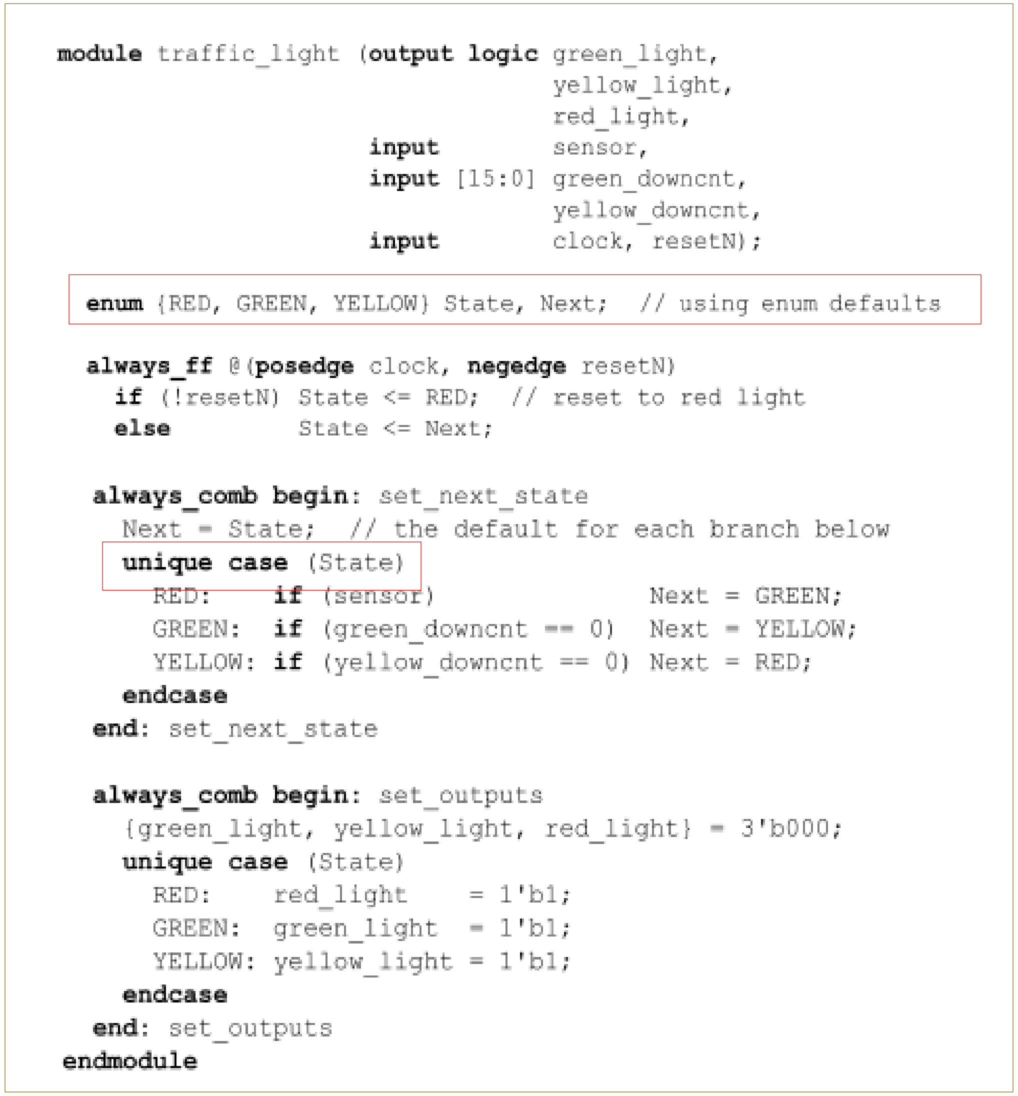
```
module patternMoore(input logic clk,
                    input logic reset,
                    input logic a,
                    output logic y);
typedef enum logic [1:0] {S0, S1, S2} statetype;
statetype state, nextstate;
// state register
always_ff @(posedge clk, posedge reset)
  if (reset) state <= S0;
  else state <= nextstate;
// next state logic
always_comb
   case (state)
   S0: if (a) nextstate = S0;
          else nextstate = S1;
   S1: if (a) nextstate = S2;
          else nextstate = S1;
   S2: if (a) nextstate = S0;
          else nextstate = S1;
   default: nextstate = S0;
  endcase
// output logic
assign y = (state = = S2);
endmodule
```

Moore

```
module patternMealy(input logic clk,
                    input logic reset,
                    input logic a,
                    output logic y);
typedef enum logic {S0, S1} statetype;
statetype state, nextstate;
// state register
always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else state <= nextstate;
// next state logic
always_comb
   case (state)
   S0: if (a) nextstate = S0;
          else nextstate = S1;
   S1: if (a) nextstate = S0;
          else nextstate = S1;
   default: nextstate = S0;
endcase
// output logic
assign y = (a & state = = S1);
endmodule
```

Mealy

Enumerated Types + Unique Case

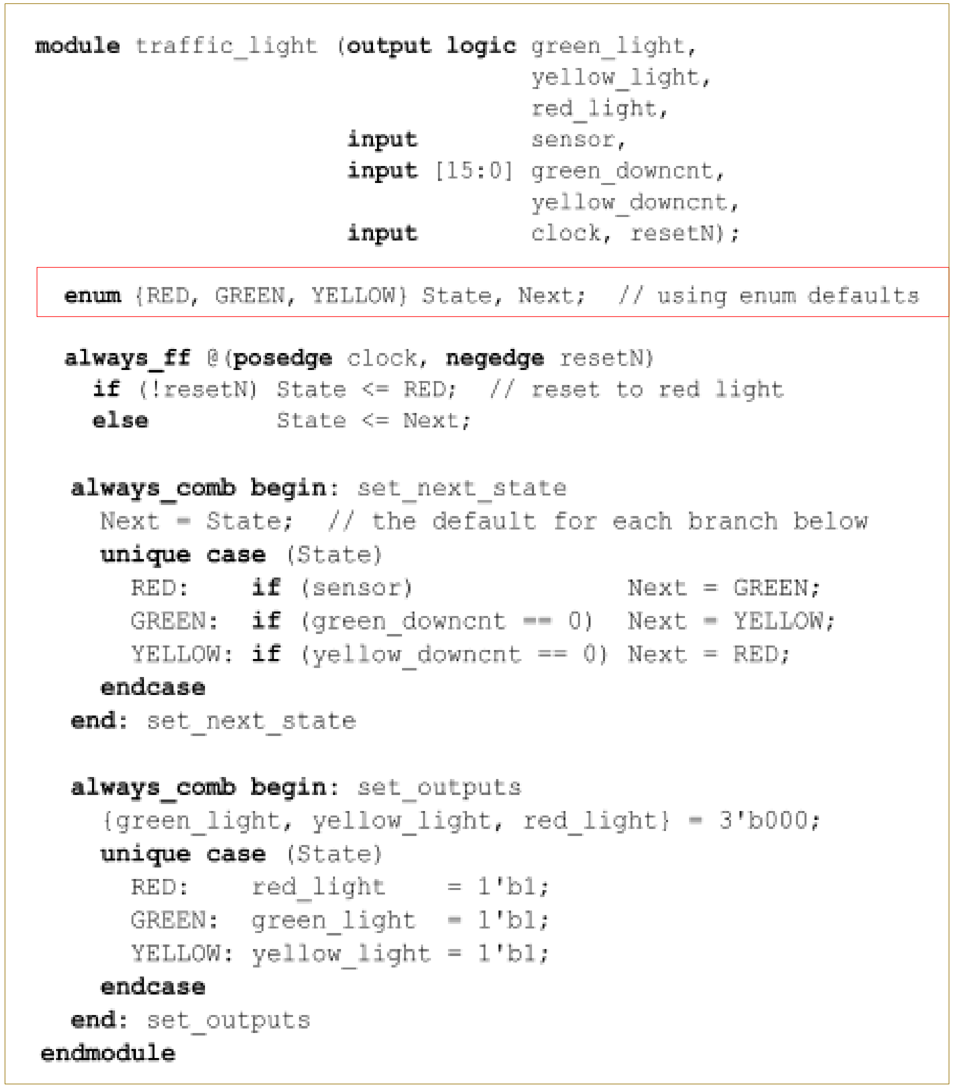


```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {RED, GREEN, YELLOW} State, Next;  // using enum defaults

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)              Next = GREEN;
      GREEN:  if (green_downcnt == 0)  Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state

  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

- By using enumerated types, the only possible values of the State and Next variables are the ones listed in their enumerated type lists.
- The unique modifier to the case statements in the state machine logic helps confirm that the case statements cover all possible values of the State and Next variables

Default (int) -> logic

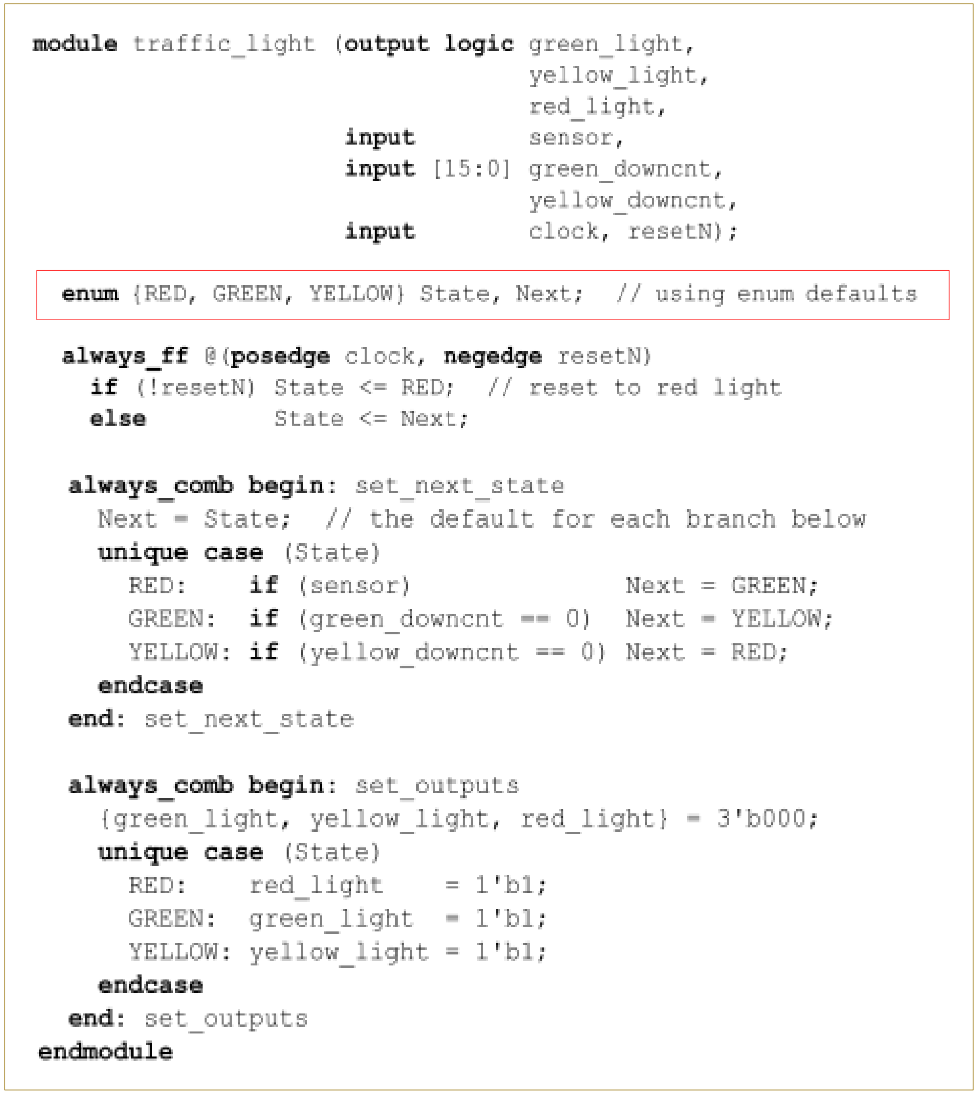
- It uses the **default** enum base type of **int**, and the default values for each value label (0, 1 and 2, respectively).
- These defaults might not accurately reflect hardware behavior in simulation.
  - The **int** type is a 32-bit 2-state type.
  - The real hardware only needs a 2- or 3-bit vector, depending on state coding.
- The gate-level model of the implementation will have 4-state semantics.
- **Solution: Use logic as Based Type**



```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {RED, GREEN, YELLOW} State, Next;  // using enum defaults

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)              Next = GREEN;
      GREEN:  if (green_downcnt == 0)  Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state

  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Explicit Enumeration Label Values

- Since the values for the enumerated labels were not explicitly specified, synthesis compilers might optimize the gate-level implementation to different values for each state.
- This makes it more difficult to compare the pre- and post-synthesis model functionality, or to specify assertions that work with both the pre- and post-synthesis models.
- **Solution: Explicit Enumeration Label Values**



```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {RED, GREEN, YELLOW} State, Next;  // using enum defaults

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)             Next = GREEN;
      GREEN:  if (green_downcnt == 0) Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state

  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Explicit Encoding => Consistency

- The enumerated label values that represent the state sequencing are explicitly specified in the RTL model.
- Synthesis compilers will retain these values in the gate-level implementation.
- This helps in comparing pre- and post-synthesis model functionality.
- It also makes it easier to specify verification assertions that work with both the pre- and post-synthesis models.

```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum logic [2:0] {RED    = 3'b001, // explicit enum definition
                    GREEN  = 3'b010,
                    YELLOW = 3'b100} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)                Next = GREEN;
      GREEN:  if (green_downcnt   == 0) Next = YELLOW;
      YELLOW: if (yellow_downcnt  == 0) Next = RED;
    endcase
  end: set_next_state
  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Optimization in Synthesis

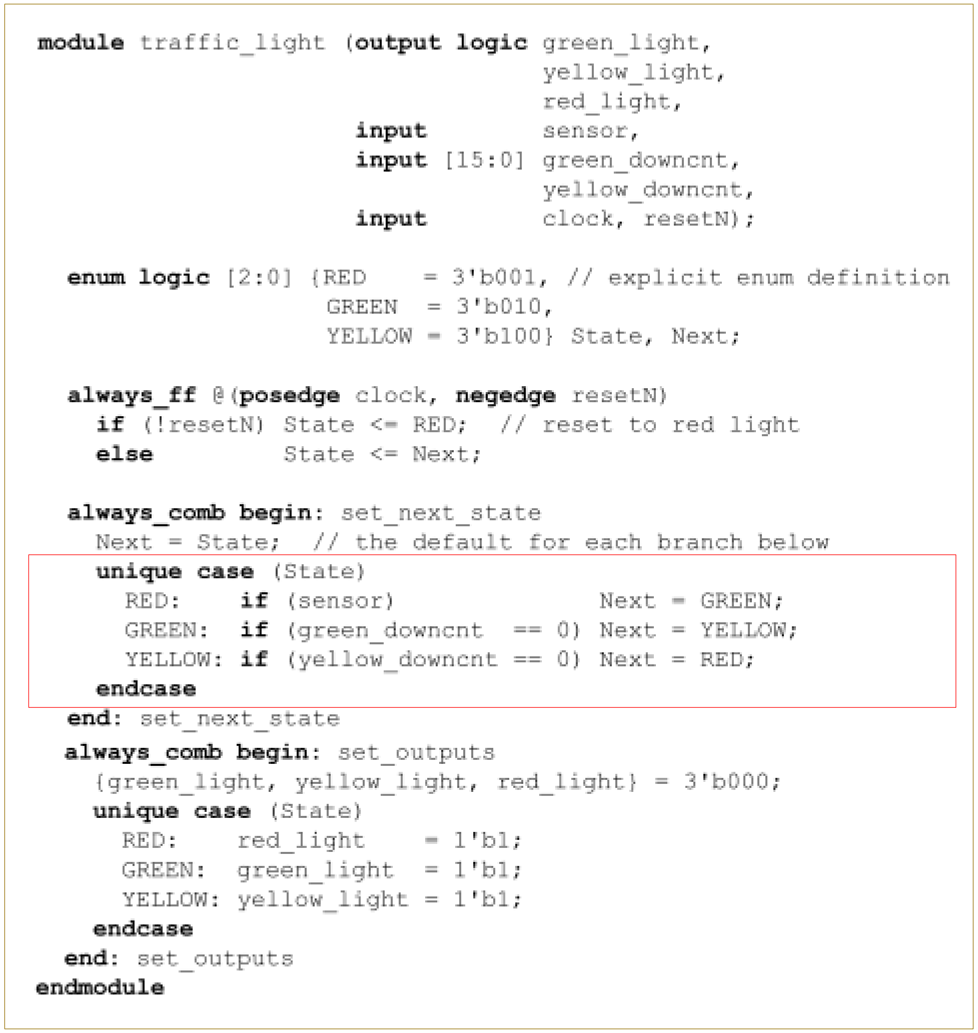
- Synthesis compiler may provide a way to override the explicit enumeration label values, in order to optimize the gate-level implementation;
- This type of optimization cancels many of the benefits of specifying explicit enumeration values.

Unique Case with One-Hot

- The use of the **unique** modifier to the case statement in the preceding example is important.
- Since a one-hot state machine only has one bit of the state register set at a time,
  - only one of the case selection items will match the literal value of 1 in the case expression.

Regular Case Statements with Enumerated Types

- The typical use of a case statement is to
  - specify a variable as the case expression, and then
  - list explicit values to be matched as the list of case selection items.



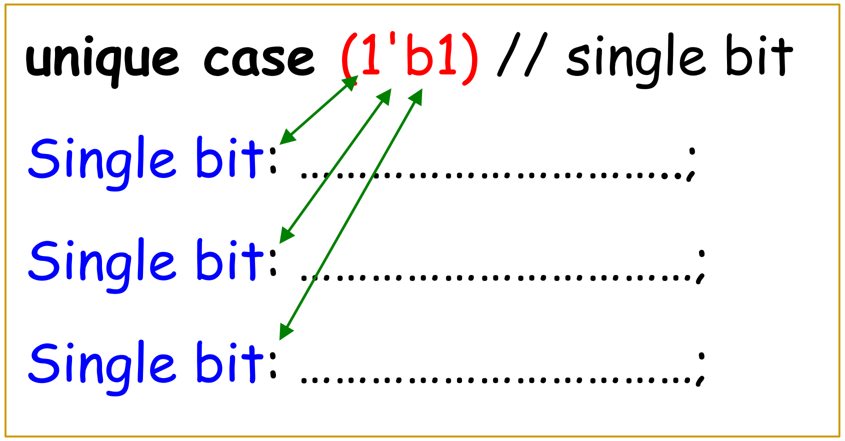
```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum logic [2:0] {RED    = 3'b001, // explicit enum definition
                    GREEN  = 3'b010,
                    YELLOW = 3'b100} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)              Next = GREEN;
      GREEN:  if (green_downcnt  == 0) Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state
  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Regular Case: Vector Comparison

```
unique case (string) // vector
String : ..............................;
String : ..............................;
String : ..............................;
```

```
unique case (opcode)
    3'b000: y = a + b;
    3'b001: y = a - b;
    3'b010: y = a * b;
    3'b100: y = a / b;
endcase
```

Reversed Case: Single Bit Comparison



```
unique case (1'b1) // single bit
Single bit: ..............................;
Single bit: ..............................;
Single bit: ..............................;
```

Reversed Case Statements

```
enum {R_BIT = 0, // index of RED state in State register
      G_BIT = 1, // index of GREEN state in State register
      Y_BIT = 2} state_bit;
enum logic [2:0] { RED = 3'b001<<R_BIT,
                   GREEN = 3'b001<<G_BIT,
                   YELLOW = 3'b001<<Y_BIT} State, Next;
```

```
RED     =001
GREEN   =010
YELLOW  =100
```

```
R_BIT = 0,
G_BIT = 1,
Y_BIT = 2
```

Reversed Case Statements

State is a 3-bit vector

State=(State[2], State[1], State[0])

```
RED    =001
GREEN  =010
YELLOW =100
```

```
R_BIT = 0,
G_BIT = 1,
Y_BIT = 2
```

```
1) State=001
unique case (1'b1)
State[0] = 1 : ………;
State[1]  = 0 : ………;
State[2]  = 0 : ………;
```

```
2) State=010
unique case (1'b1)
State[0] = 0 : ………;
State[1]  = 1 : ………;
State[2]  = 0 : ………;
```

```
3) State=100
unique case (1'b1)
State[0] = 0 : ………;
State[1]  = 0 : ………;
State[2]  = 1 : ………;
```

Reversed Case Statements

- It has an effective hardware implementation.
- It can be used only for One-hot coding modeling.

```
unique case (1'b1) // reversed case statement
State[R_BIT]: if (sensor) Next = GREEN;
State[G_BIT]: if (green_downcnt == 0) Next = YELLOW;
State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
```

Reversed Case Statements with Enumerated Types

```
module traffic_light ( output logic green_light,
                               yellow_light,
                               red_light,
                               input sensor,
                               input [15:0] green_downcnt,
                               yellow_downcnt,
                               input clock, resetN);
enum {R_BIT = 0, // index of RED state in State register
      G_BIT = 1, // index of GREEN state in State register
      Y_BIT = 2} state_bit;
// shift a 1 to the bit that represents each state
enum logic [2:0] { RED = 3'b001<<R_BIT,
                   GREEN = 3'b001<<G_BIT,
                   YELLOW = 3'b001<<Y_BIT} State, Next;
always_ff @(posedge clock, negedge resetN)
if (!resetN) State <= RED; // reset to red light
     else State <= Next;
always_comb begin: set_next_state
Next = State; // the default for each branch below
```

```
unique case (1'b1) // reversed case statement
State[R_BIT]: if (sensor) Next = GREEN;
State[G_BIT]: if (green_downcnt == 0) Next = YELLOW;
State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
endcase
end: set_next_state

always_comb begin: set_outputs
{red_light, green_light, yellow_light} = 3'b000;
unique case (1'b1) // reversed case statement
      State[R_BIT]: red_light = 1'b1;
      State[G_BIT]: green_light = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
endcase
end: set_outputs
endmodule
```

Reversed Case Statements

```
enum {R_BIT = 0, // index of RED state in State register
        G_BIT = 1, // index of GREEN state in State register
        Y_BIT = 2} state_bit;
enum logic [2:0] { RED = 3'b001<<R_BIT,
                    GREEN = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;
```

Logic Label	Label Internal Values
RED	=001
GREEN	=010
YELLOW	=100

<< shift operation

- a = 5'b10100;
- b = a <<< 2; //b == 5'b10000
- c = a >>> 2; //c == 5'b11101, 'cause sign bit was `1`
- d = a << 2; //d == 5'b10000
- e = a >> 2; //e == 5'b00101

Reversed Case Statements with Enumerated Types

- An effective style for modeling one-hot state machines:
  - the case expression and the case selection items are reversed.
  - The case expression is specified as the literal value to be matched, which is a 1-bit value of 1.
  - The case selection items are each bit of the state variable.
- In some synthesis compilers, using the reversed case style for one-hot state machines might yield more optimized synthesis results than the standard style of case statements.

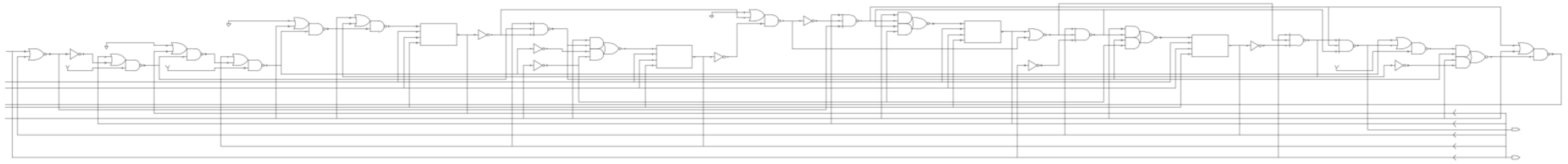
```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {R_BIT = 0,  // index of RED state in State register
        G_BIT = 1,  // index of GREEN state in State register
        Y_BIT = 2} state_bit;

  // shift a 1 to the bit that represents each state
  enum logic [2:0] {RED    = 3'b001<<R_BIT,
                    GREEN  = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: if (sensor)              Next = GREEN;
      State[G_BIT]: if (green_downcnt  == 0) Next = YELLOW;
      State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state


  always_comb begin: set_outputs
    {red_light, green_light, yellow_light} = 3'b000;
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: red_light    = 1'b1;
      State[G_BIT]: green_light  = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

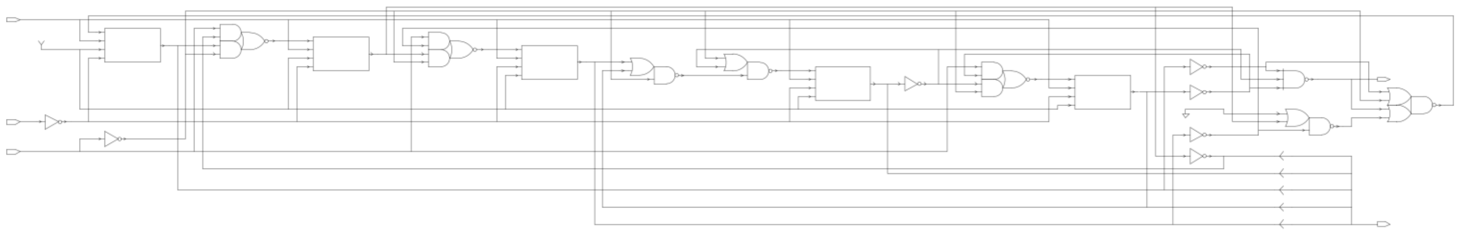
Regular and Reversed Case Statements

```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum logic [2:0] {RED    = 3'b001, // explicit enum definition
                    GREEN  = 3'b010,
                    YELLOW = 3'b100} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (State)
      RED:    if (sensor)              Next = GREEN;
      GREEN:  if (green_downcnt  == 0) Next = YELLOW;
      YELLOW: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state
  always_comb begin: set_outputs
    {green_light, yellow_light, red_light} = 3'b000;
    unique case (State)
      RED:    red_light    = 1'b1;
      GREEN:  green_light  = 1'b1;
      YELLOW: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {R_BIT = 0,  // index of RED state in State register
        G_BIT = 1,  // index of GREEN state in State register
        Y_BIT = 2} state_bit;

  // shift a 1 to the bit that represents each state
  enum logic [2:0] {RED    = 3'b001<<R_BIT,
                    GREEN  = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: if (sensor)              Next = GREEN;
      State[G_BIT]: if (green_downcnt  == 0) Next = YELLOW;
      State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state


  always_comb begin: set_outputs
    {red_light, green_light, yellow_light} = 3'b000;
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: red_light    = 1'b1;
      State[G_BIT]: green_light  = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

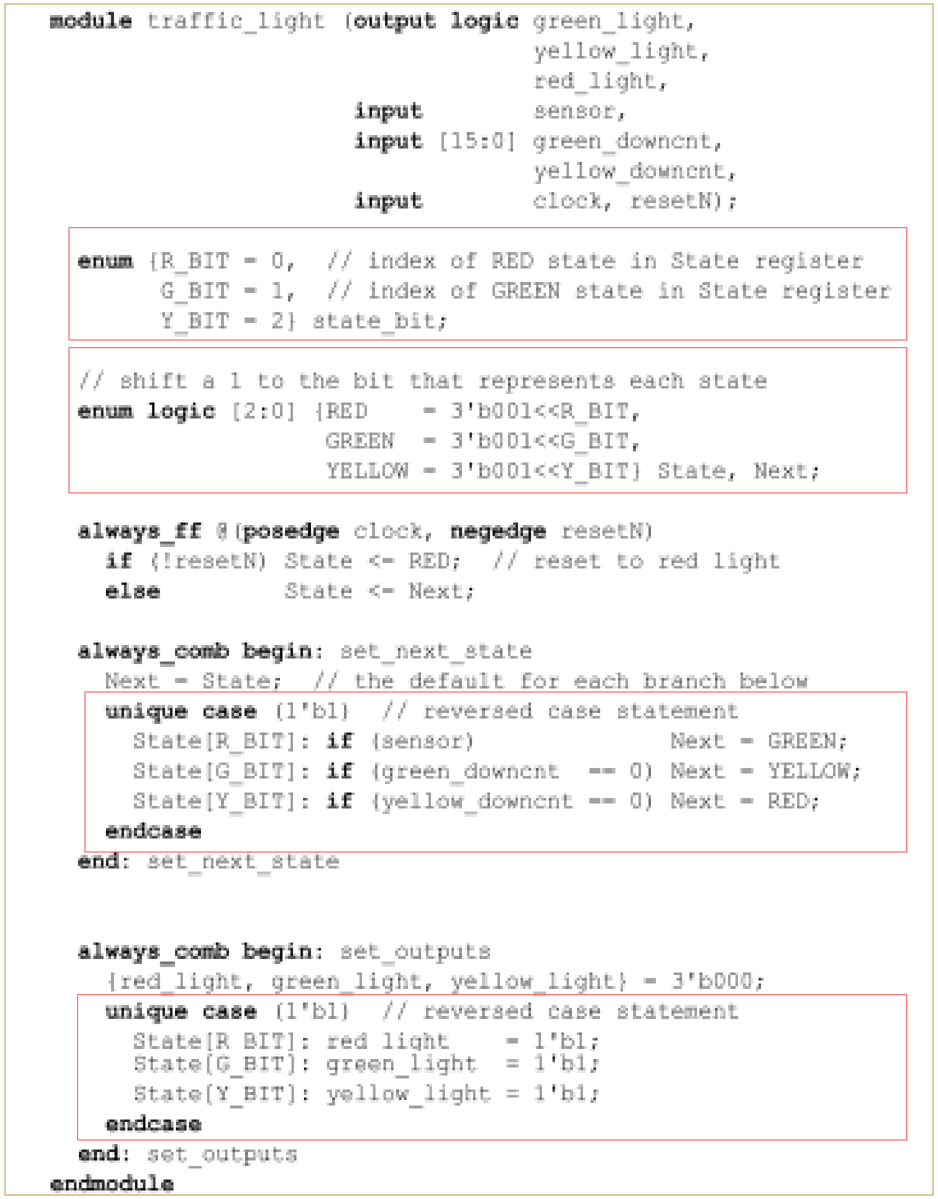
Regular and Reversed Case Statements





Reversed Case Statements

- A second enumerated type variable is declared that represents the index number for each bit of the one-hot State register.
- The name R_BIT has a value of 0, which corresponds to bit 0 of the State variable (the bit that represents the RED state).



```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {R_BIT = 0,  // index of RED state in State register
        G_BIT = 1,  // index of GREEN state in State register
        Y_BIT = 2} state_bit;

  // shift a 1 to the bit that represents each state
  enum logic [2:0] {RED    = 3'b001<<R_BIT,
                    GREEN  = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: if (sensor)              Next = GREEN;
      State[G_BIT]: if (green_downcnt  == 0) Next = YELLOW;
      State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state


  always_comb begin: set_outputs
    {red_light, green_light, yellow_light} = 3'b000;
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: red_light    = 1'b1;
      State[G_BIT]: green_light  = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

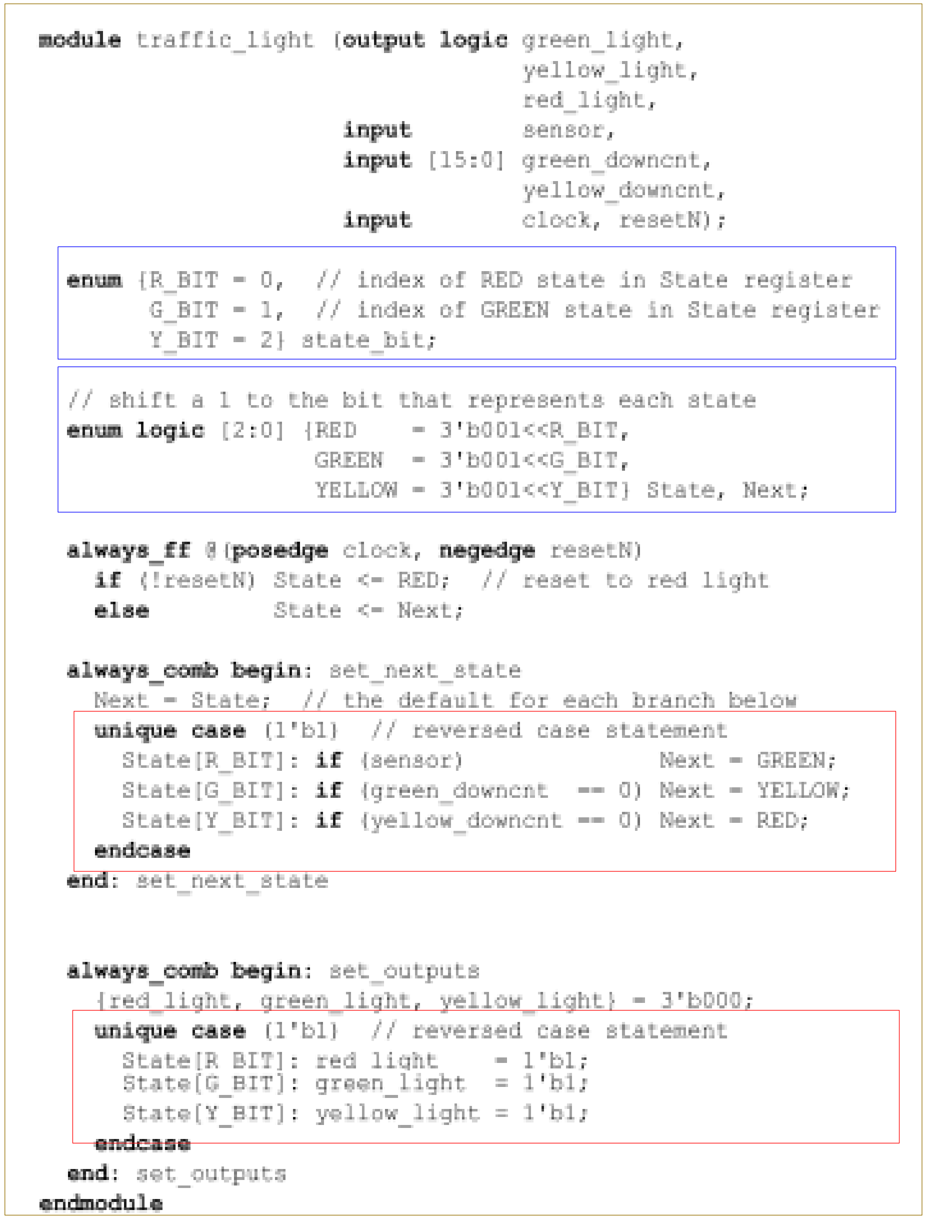
Reversed Case Statements with Enumerated Types

```
module traffic_light ( output logic green_light,
                        yellow_light,
                        red_light,
                        input sensor,
                        input [15:0] green_downcnt,
                        yellow_downcnt,
                        input clock, resetN);
enum {R_BIT = 0, // index of RED state in State register
      G_BIT = 1, // index of GREEN state in State register
      Y_BIT = 2} state_bit;
// shift a 1 to the bit that represents each state
enum logic [2:0] { RED = 3'b001<<R_BIT,
                   GREEN = 3'b001<<G_BIT,
                   YELLOW = 3'b001<<Y_BIT} State, Next;
always_ff @(posedge clock, negedge resetN)
if (!resetN) State <= RED; // reset to red light
   else State <= Next;
always_comb begin: set_next_state
Next = State; // the default for each branch below
```

- The enumerated variable state_bit specifies which bit of the state sequencer represents each state (the 1-hot bit).
- The value for each state label is calculated by shifting a 3-bit value of 001 (binary) to the bit position that is "hot" for that state.
- A value of 001 shifted 0 times (the value of R_BIT) is 001 (binary).
- A 001 shifted 1 time (the value of G_BIT) is 010 (binary), and shifted 2 times (the value of Y_BIT) is 100 (binary).
- The same enumerated state_bit labels, R_BIT, G_BIT and Y_BIT, are used in the functional code to test which bit of State is "hot".

Reversed Case Statements

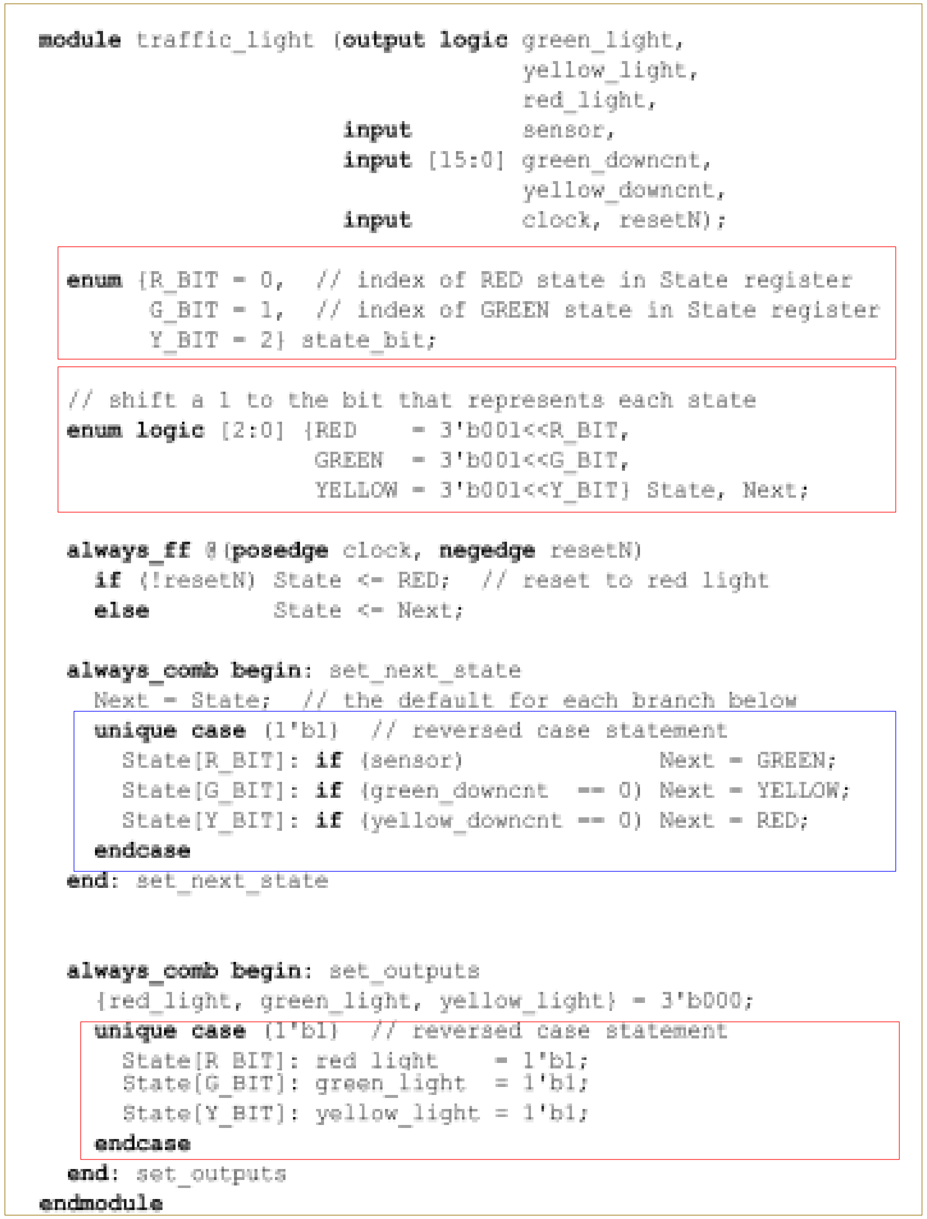
- The enumerated variable state_bit specifies which bit of the state sequencer represents each state (the 1-hot bit).
- The value for each state label is calculated by shifting a 3-bit value of 001 (binary) to the bit position that is "hot" for that state.
  - A value of 001 shifted 0 times (the value of R_BIT) is 001 (binary).
  - A 001 shifted 1 time (the value of G_BIT) is 010 (binary), and
  - shifted 2 times (the value of Y_BIT) is 100 (binary).



```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {R_BIT = 0,  // index of RED state in State register
        G_BIT = 1,  // index of GREEN state in State register
        Y_BIT = 2} state_bit;

  // shift a 1 to the bit that represents each state
  enum logic [2:0] {RED    = 3'b001<<R_BIT,
                    GREEN  = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: if (sensor)              Next = GREEN;
      State[G_BIT]: if (green_downcnt  == 0) Next = YELLOW;
      State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state


  always_comb begin: set_outputs
    {red_light, green_light, yellow_light} = 3'b000;
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: red_light    = 1'b1;
      State[G_BIT]: green_light  = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Reversed Case Statements

- The same enumerated state_bit labels, R_BIT, G_BIT and Y_BIT, are used in the functional code to test which bit of State is "hot".
- The definitions of the enumerated labels for State and the bit-selects of the State variable are linked together by the definition of state_bit.
- Using this type of case statement ensures inference of efficient comparison logic that only does 1-bit comparisons against the onehot bits of the state and next vectors.



```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {R_BIT = 0,  // index of RED state in State register
        G_BIT = 1,  // index of GREEN state in State register
        Y_BIT = 2} state_bit;

  // shift a 1 to the bit that represents each state
  enum logic [2:0] {RED    = 3'b001<<R_BIT,
                    GREEN  = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: if (sensor)              Next = GREEN;
      State[G_BIT]: if (green_downcnt  == 0) Next = YELLOW;
      State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state


  always_comb begin: set_outputs
    {red_light, green_light, yellow_light} = 3'b000;
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: red_light    = 1'b1;
      State[G_BIT]: green_light  = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Reverse Case Statement

```
enum {
 IDLE = 0,
 READ = 1,
 DLY  = 2,
 DONE = 3
} state, next;

// Sequential state transition
always_ff @(posedge clk or negedge rst_n)
 if (!rst_n) begin
   state      <= '0; // default assignment
   state[IDLE] <= 1'b1;
 end
 else
   state      <= next;
```

- The state enumerated type values represent indices into the *state* and *next* vectors.
- Synthesis tools efficiently generate output assignment and next state logic that does only 1-bit comparison against the state vectors.

```
// Combinational next state logic
always_comb begin
 next = '0;
 unique case (1'b1)
  state[IDLE] : begin
   if (go)
    next[READ] = 1'b1;
   else
    next[IDLE] = 1'b1;
  end
  state[READ] : next[ DLY] = 1'b1;
  state[ DLY] : begin
   if (!ws)
    next[DONE] = 1'b1;
   else
    next[READ] = 1'b1;
  end
  state[DONE] : next[IDLE] = 1'b1;
 endcase
end

// Make output assignments
always_ff @(posedge clk or negedge rst_n)
…
```

Reversed Case Statements with Enumerated Types

```
enum {R_BIT = 0, // index of RED state in State register
      G_BIT = 1, // index of GREEN state in State register
      Y_BIT = 2} state_bit;
// shift a 1 to the bit that represents each state
enum logic [2:0] { RED = 3'b001<<R_BIT,
                   GREEN = 3'b001<<G_BIT,
                   YELLOW = 3'b001<<Y_BIT} State, Next;
.....
always_comb begin: set_next_state
Next = State; // the default for each branch below
unique case (1'b1) // reversed case statement
State[R_BIT]: if (sensor) Next = GREEN;
State[G_BIT]: if (green_downcnt == 0) Next = YELLOW;
State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
endcase
end: set_next_state
.........
```

- This scheme serves two important purposes:
  - There is no possibility of a coding error that defines different 1-hot bit positions in the two enumerated type definitions.
  - Should the design specification change the 1-hot definitions, only the enumerated type specifying the bit positions has to change.
- The enumerated type defining the state names will automatically reflect the change.

Reversed Case Statements with Verilog

```
module fsm_cc1_3oh
 (output reg rd, ds,
  input      go, ws, clk, rst_n);
 parameter IDLE = 0,
           READ = 1,
           DLY  = 2,
           DONE = 3;
 reg [3:0] state, next;
 always @(posedge clk or negedge rst_n)
  if (!rst_n) begin
          state      <= 4'b0;
          state[IDLE] <= 1'b1;
  end
  else       state      <= next;
 always @(state or go or ws) begin
  next = 4'b0;
  case (1'b1) // synopsys parallel_case
   state[IDLE] : if (go)  next[READ] = 1'b1;
                 else     next[IDLE] = 1'b1;
   state[READ] :          next[ DLY] = 1'b1;
   state[ DLY] : if (!ws) next[DONE] = 1'b1;
                 else     next[READ] = 1'b1;
   state[DONE] :          next[IDLE] = 1'b1;
  endcase
 end
 always @(posedge clk or negedge rst_n)
  if (!rst_n) begin
   rd <= 1'b0;
   ds <= 1'b0;
  end
  else begin
   rd <= 1'b0;
   ds <= 1'b0;
   case (1'b1) // synopsys parallel_case
    next[READ] : rd <= 1'b1;
    next[ DLY] : rd <= 1'b1;
    next[DONE] : ds <= 1'b1;
   endcase
  end
endmodule
```

Miscellaneous on Modeling FSMs in SV

Unused States

- As an enumerated type, the State variable has a restricted set of values.
- The State variable is a multi-bit vector, which, at the gate-level, can reflect logic values not defined in the enumerated list.
- A FSM with three states requires a 3-bit state register for one-hot encoding.
  - This 3-bit register can contain 8 possible values.
- The hardware registers represented can hold all possible values, not just the values listed in the enumerated list.

Unused States: Enumerated Types with **logic**

```
// case statement with enumerated X default
enum logic [2:0] {   RED = 3'b001,
                     GREEN = 3'b010,
                     YELLOW = 3'b100,
                     BAD_STATE = 3'bxxx,
                     } State, Next;
case (State)
    RED: Next = GREEN;
    GREEN: Next = YELLOW;
    YELLOW: Next = RED;
    default: Next = BAD_STATE;
endcase
```

- In SV, an enumerated type can only be assigned values from its enumerated list.
- If an X assignment is desired, the base type of the enumerated type must be defined a 4-state type, such as **logic**, and an enumerated label must be defined with an explicit value of X.

Unused States: A Better SV Solution

```
// case statement with enumerated X default
enum logic [2:0] {   RED = 3'b001,
                     GREEN = 3'b010,
                     YELLOW = 3'b100,
                     BAD_STATE = 3'bxxx,
                     } State, Next;
case (State)
      RED: Next = GREEN;
      GREEN: Next = YELLOW;
      YELLOW: Next = RED;
      default: Next = BAD_STATE;
endcase
```

```
// unique case statement
enum logic [2:0] {   RED = 3'b001,
                     GREEN = 3'b010,
                     YELLOW = 3'b100,
                     } State, Next;
unique case (State)
      RED: Next = GREEN;
      GREEN: Next = YELLOW;
      YELLOW: Next = RED;
endcase
```

Assigning State Values to Enumerated Variables

- Assignments with Enumerated Type Variables:
  - Enumerated types are more strongly typed than other Verilog and SystemVerilog variables.
  - Enumerated types can only be assigned a value that is a member of the type list of that enumerated type.
  - An enumerated type can be assigned the value of another enumerated type, but only if both enumerated types are from the same definition.

Enumerated Assignments: Illegal

```
enum {R_BIT = 0, // index of RED state in State register
G_BIT = 1, // index of GREEN state in State register
Y_BIT = 2} state_bit;
// shift a 1 to the bit that represents each state
enum logic [2:0] {RED = 3'b001<<R_BIT,
                  GREEN = 3'b001<<G_BIT,
                  YELLOW = 3'b001<<Y_BIT} State, Next;
...
always_comb begin: set_next_state
Next = 3'b000; // clear Next - ERROR: ILLEGAL
ASSIGNMENT
unique case (1'b1) // reversed case statement
// WARNING: FOLLOWING ASSIGNMENTS ARE
POTENTIAL DESIGN ERRORS
State[R_BIT]: if (sensor == 1) Next[G_BIT] = 1'b1;
State[G_BIT]: if (green_downcnt==0) Next[Y_BIT] = 1'b1;
State[Y_BIT]: if (yellow_downcnt==0) Next[R_BIT] = 1'b1;
endcase
end: set_next_state
...
```

- A common Verilog style when using one-hot state sequences is to first clear the next state variable, and then set just the one bit of next state variable that indicates what the next state will be.
- This style will not work with enumerated types.
- Two problems:
  - First, a default assignment of all zeros is made to the Next variable.
  - This is an illegal assignment.
  - An enumerated type must be assigned labels from its enumerated list, not literal values.

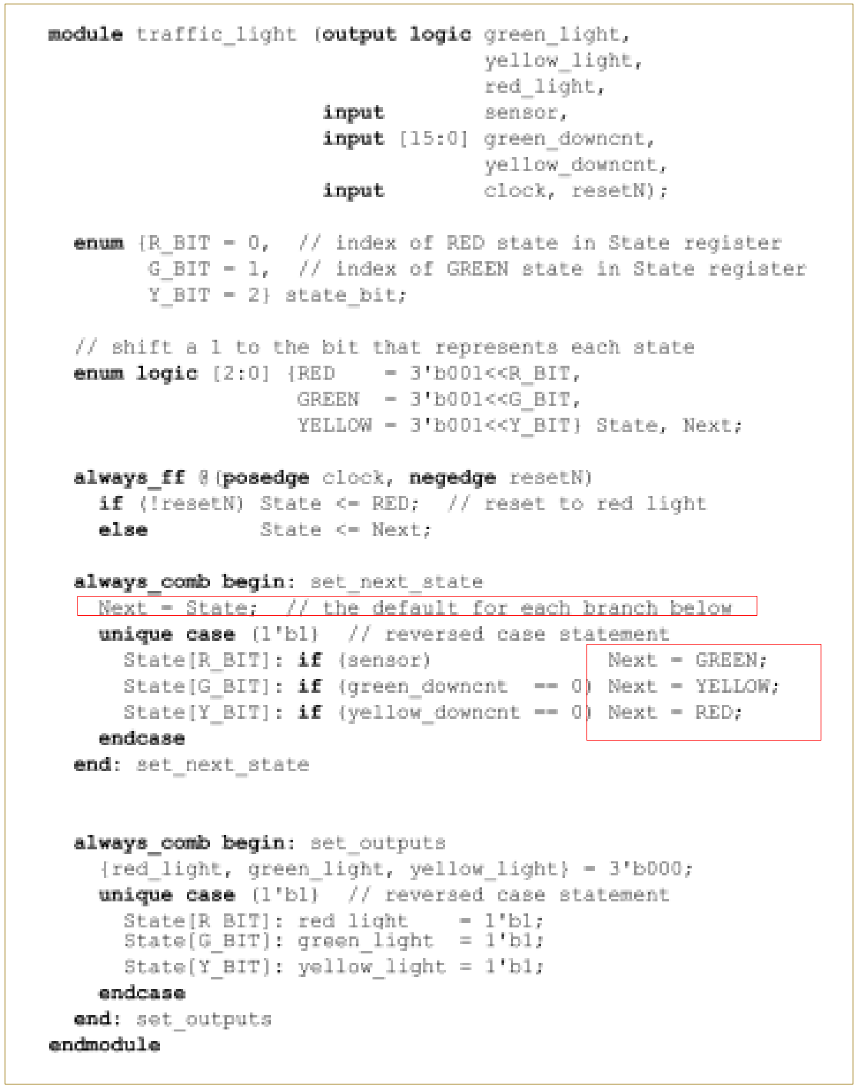
Enumerated Assignments: Legal but not Recommended

```
enum {R_BIT = 0, // index of RED state in State register
G_BIT = 1, // index of GREEN state in State register
Y_BIT = 2} state_bit;
// shift a 1 to the bit that represents each state
enum logic [2:0] {RED = 3'b001<<R_BIT,
                  GREEN = 3'b001<<G_BIT,
                  YELLOW = 3'b001<<Y_BIT} State, Next;
...
always_comb begin: set_next_state
Next = 3'b000; // clear Next - ERROR: ILLEGAL
ASSIGNMENT
unique case (1'b1) // reversed case statement
// WARNING: FOLLOWING ASSIGNMENTS ARE
POTENTIAL DESIGN ERRORS
State[R_BIT]: if (sensor == 1) Next[G_BIT] = 1'b1;
State[G_BIT]: if (green_downcnt==0) Next[Y_BIT] = 1'b1;
State[Y_BIT]: if (yellow_downcnt==0) Next[R_BIT] = 1'b1;
endcase
end: set_next_state
...
```

- Second, within the case statements, assignments are made to individual bits of the Next variable.
- Assigning to a discrete bit of an enumerated type **MAY be allowed by compilers**, but it is not a good style when using enumerated types.
- By assigning to a bit of an enumerated type variable, an illegal value could be created that is not in the enumerated type list.
- This would result in design errors that could **be difficult to debug**.

The Correct One: Use its Labels

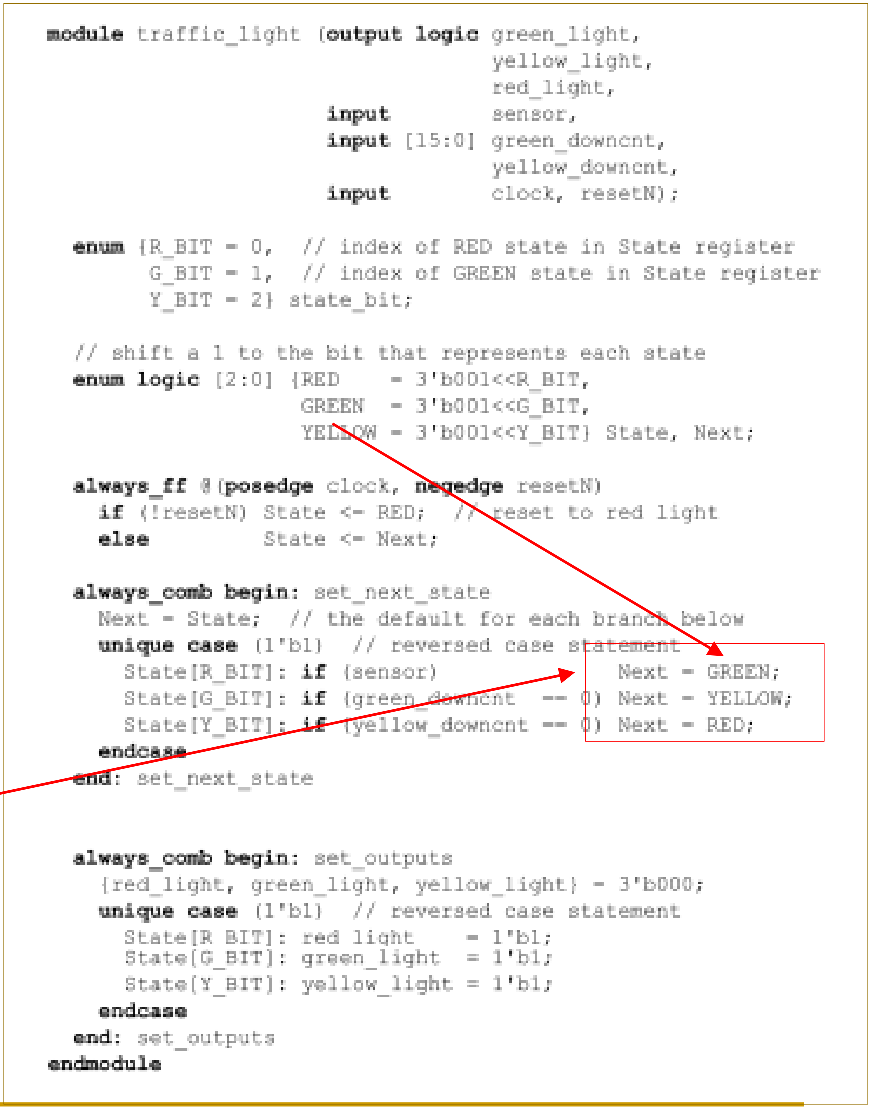
- Assignments to enumerated type variables should be from the list of labels for that type.
- Assigning to bit-selects or part-selects of an enumerated type should be avoided.
- When assignments to bits of a variable are required, the variable should be declared as standard type, such as bit or logic, instead of an enumerated type.
- *Assign an enumerated type variable a label from its enumerated list, instead of a value.*



```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {R_BIT = 0,  // index of RED state in State register
        G_BIT = 1,  // index of GREEN state in State register
        Y_BIT = 2} state_bit;

  // shift a 1 to the bit that represents each state
  enum logic [2:0] {RED    = 3'b001<<R_BIT,
                    GREEN  = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: if (sensor)                Next = GREEN;
      State[G_BIT]: if (green_downcnt  == 0)   Next = YELLOW;
      State[Y_BIT]: if (yellow_downcnt == 0)   Next = RED;
    endcase
  end: set_next_state


  always_comb begin: set_outputs
    {red_light, green_light, yellow_light} = 3'b000;
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: red_light    = 1'b1;
      State[G_BIT]: green_light  = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

The Correct One: Use its Labels

```
enum {R_BIT = 0, // index of RED state in State register
G_BIT = 1, // index of GREEN state in State register
Y_BIT = 2} state_bit;
// shift a 1 to the bit that represents each state
enum logic [2:0] {RED = 3'b001<<R_BIT,
                  GREEN = 3'b001<<G_BIT,
                  YELLOW = 3'b001<<Y_BIT} State, Next;
...
always_comb begin: set_next_state
Next = 3'b000; // clear Next - ERROR: ILLEGAL
ASSIGNMENT
unique case (1'b1) // reversed case statement
// WARNING: FOLLOWING ASSIGNMENTS ARE
POTENTIAL DESIGN ERRORS
State[R_BIT]: if (sensor == 1) Next[G_BIT] = 1'b1;
State[G_BIT]: if (green_downcnt==0) Next[Y_BIT] = 1'b1;
State[Y_BIT]: if (yellow_downcnt==0) Next[R_BIT] = 1'b1;
endcase
end: set_next_state
...
```



```
module traffic_light (output logic green_light,
                                   yellow_light,
                                   red_light,
                      input        sensor,
                      input [15:0] green_downcnt,
                                   yellow_downcnt,
                      input        clock, resetN);

  enum {R_BIT = 0,  // index of RED state in State register
        G_BIT = 1,  // index of GREEN state in State register
        Y_BIT = 2} state_bit;

  // shift a 1 to the bit that represents each state
  enum logic [2:0] {RED    = 3'b001<<R_BIT,
                    GREEN  = 3'b001<<G_BIT,
                    YELLOW = 3'b001<<Y_BIT} State, Next;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= RED;  // reset to red light
    else         State <= Next;

  always_comb begin: set_next_state
    Next = State;  // the default for each branch below
    unique case (1'b1)  // reversed case statement
      State[R_BIT]: if (sensor)               Next = GREEN;
      State[G_BIT]: if (green_downcnt  == 0) Next = YELLOW;
      State[Y_BIT]: if (yellow_downcnt == 0) Next = RED;
    endcase
  end: set_next_state


  always_comb begin: set_outputs
    {red_light, green_light, yellow_light} = 3'b000;
    unique case (1'b1)  // reversed case statement
      State[R BIT]: red light    = 1'b1;
      State[G_BIT]: green_light  = 1'b1;
      State[Y_BIT]: yellow_light = 1'b1;
    endcase
  end: set_outputs
endmodule
```

Lock-up with Default Values

```
enum {WAITE, LOAD, STORE} State, Next;
always @(posedge clock, negedge resetN)
if (!resetN) State <= WAITE;
        else State <= Next;
always @(State)
    case (State)
        WAITE: Next = LOAD;
        LOAD: Next = STORE;
        STORE: Next = WAITE;
    endcase
```

- Both State and Next variables begin simulation with a value of 0,
  - = WAITE =0 .
- At every positive edge of clock,
  - State <= Next (0)
  - the value it already has
  - no transition occurs.
- Since there is no transition,
  - always @(State) is not triggered
  - Next is not changed from its initial value of 0.

Solution: logic type

```
enum {WAITE, LOAD, STORE} State, Next;
always @(posedge clock, negedge resetN)
if (!resetN) State <= WAITE;
            else State <= Next;
always @(State)
      case (State)
            WAITE: Next = LOAD;
            LOAD: Next = STORE;
            STORE: Next = WAITE;
      endcase
```

```
enum logic[1:0] {WAITE, LOAD, STORE} State,
Next;
always @(posedge clock, negedge resetN)
if (!resetN) State <= WAITE;
            else State <= Next;
always @(State)
      case (State)
            WAITE: Next = LOAD;
            LOAD: Next = STORE;
            STORE: Next = WAITE;
      endcase
```

- Simulation will then begin with State and Next having an un-initialized value of X.
- In RTL simulation, when reset is applied, the State variable will transition from X to its reset value of WAITE (=00).
  - This transition will trigger the logic that decodes Next, setting Next to its appropriate value of LOAD.

Solution: always_comb

```
enum {WAITE, LOAD, STORE} State, Next;
always @(posedge clock, negedge resetN)
if (!resetN) State <= WAITE;
          else State <= Next;
always @(State)
     case (State)
          WAITE: Next = LOAD;
          LOAD: Next = STORE;
          STORE: Next = WAITE;
     endcase
```

```
enum {WAITE, LOAD, STORE} State, Next;
always @(posedge clock, negedge resetN)
if (!resetN) State <= WAITE;
          else State <= Next;
always_comb
    case (State)
          WAITE: Next = LOAD;
          LOAD: Next = STORE;
          STORE: Next = WAITE;
     endcase
```

- An always_comb block automatically executes its statements once at simulation time zero, even if there were no transitions on its inferred sensitivity list.
  - The initial value of State will be decoded, and the Next variable set accordingly.
- This fixes the start of simulation lock-up problem.

A SV Robust Solution

```
enum logic[1:0] {WAITE, LOAD, STORE}
State, Next;
always @(posedge clock, negedge resetN)
if (!resetN) State <= WAITE;
            else State <= Next;
always_comb
     case (State)
            WAITE: Next = LOAD;
            LOAD: Next = STORE;
            STORE: Next = WAITE;
      endcase
```

```
enum logic[1:0] {WAITE, LOAD, STORE} State, Next;
always_ff @(posedge clock, negedge resetN)
if (!resetN) State <= WAITE;
            else State <= Next;
always_comb
     unique case (State)
            WAITE: Next = LOAD;
            LOAD: Next = STORE;
            STORE: Next = WAITE;
      endcase
```

- If State had not been reset, it would be a logic X, which will not match any of the case items to which State is compared.
- The unique case (State) statement will issue a run-time warning whenever no case items match the case expression.
- A warning would also be issued if the case expression matches more than one case item.

A SV Robust Solution

```
enum logic[1:0] {WAITE, LOAD, STORE} State, Next;
always_ff @(posedge clock, negedge resetN)
if (!resetN) State <= WAITE;
            else State <= Next;
always_comb
     unique case (State)
            WAITE: Next = LOAD;
            LOAD: Next = STORE;
            STORE: Next = WAITE;
      endcase
```

- A SV Robust Solution:
  - Combining Unique Case with a logic based Enumerated Types
- This combination of SV constructs not only simplifies writing RTL code, it can trap design problems that could have been difficult to detect and debug in Verilog.

References

- SystemVerilog for Design, by Stuart Sutherland et al.
- "Synthesizable Finite State Machine Design Techniques" by Clifford E. Cummings, Sunburst Design, Inc. 2003.
- "Synthesizing SystemVerilog" by S. Sutherland, D. Mills. 2013.
- Sutherland, "SystemVerilog Saves the Day—the Evil Twins are Defeated! unique and priority" are the new Heroes", Synopsys Users Group (SNUG) conference, San Jose, California, March 2005.
- Mills and Sutherland, "SystemVerilog Assertions Are For Design Engineers Too!", Synopsys Users Group (SNUG) conference, San Jose, California, March 2006.
- Mills, "Being assertive with your X (SystemVerilog assertions for dummies)", Synopsys Users Group Conference (SNUG) San Jose, California, 2004.
- Mills, "Yet Another Latch and Gotchas Paper", Synopsys Users Group (SNUG) conference, San Jose, California, March 2012.
- Sutherland, "I'm Still in Love with My X!", Design and Verification Conference (DVCon), San Jose, California, February 2013.
- Greene, Salz and Booth, "X-Optimism Elimination during RTL Verification", Synopsys Users Group Conference (SNUG) San Jose, 2012.