System Verilog FAQs

Vikas Dhiman

November 30, 2022

Question 1. *Can you give us a template for all modules?*

There is no general template, but the following template will work for all **Synchronous Sequential** modules that do not call any other module.

```
// module keyword starts a module definition.
module module_named_foo(
    // Every module should have a single bit clock and single bit reset signal
    input wire [0:0] clock,
    input wire [0:0] reset,
    // All inputs to the module are declared as wires
    input wire [bits1:0] input_1,
    input wire [bits2:0] input_2,
    ...
    // All outputs from the module are declared as regs
    output reg [bits3:0] output_1,
    output reg [bits4:0] output_2
);
// Every synchronous module will need some states
// States are always declared as registers
reg [bit5:0] state_1;
reg [bit6:0] state_2;
...

// We have the choice of writing procedural code or structural code. Here we
// use procedural block. I will separate the procedural code into a
// register block and two combinational logic blocks

/////////////////////////////////////////////////////////////////////
// First block: Register block
/////////////////////////////////////////////////////////////////////
// Create some intermediate states
// These intermediate states could have been wires if we were using assign
// statement to create the combinational block. assign statement is easy to
// write only for very simple circuits like slowclock. For the rest, we use
// procedural code and reg for intermediate variables.
reg [bit5:0] next_state_1;
reg [bit6:0] next_state_2;
...
// Always block that triggers only on the posedge of clock and posedge of
// reset signal.
// always_ff is same as always, but it ensures that a flip-flop circuit is
// synthesized.
always_ff @(posedge clock or posedge reset) begin
  if (reset) begin
    // This is the initialization block. You can assign initial values to your
    // state here
    state_1 <= 0;
```

```
      state_2 <= 0;
      ...
      // Using the non-blocking assign ``<='' in register block is recommended
    end else begin
      // At the rising edge next state is copied to current state
      state_1 <= next_state_1;
      state_2 <= next_state_2;
      ...
    end
  end

  ////////////////////////////////////////////////////////////////////
  // Second block: converts from current state and input to next state
  ////////////////////////////////////////////////////////////////////
  // 1. Most of the logic of your state machine goes here
  // 2. Note that combinational logic always block does not trigger on posedge
  //    clock instead it triggers on any change in input.
  // 3. You can also use always_comb instead of always @(*) which will ensure that
  //    a combinational logic is synthesized.
  // 4. Only next_state must be on the left hand side.
  always @(*) begin
     if (/*some conditions on states and inputs */) begin
        next_state_1 = //some expression of states and inputs;
        next_state_2 = //some expression of states and inputs;
        ...
        // Using the blocking assign ``='' in combinational block is recommended
     end else if (/*more conditions on states and inputs */) begin
       next_state_1 = // some expression of states and inputs;
       next_state_2 = // some expression of states and inputs;
       ...
     end else begin
       next_state_1 = // some expression of state and inputs;
       next_state_2 = // some expression of state and inputs;
       ...
     end
  end

  ////////////////////////////////////////////////////////////////////
  // Third block: converts from current state and input to output (Mealy)
  ////////////////////////////////////////////////////////////////////
  always @(*) begin
      if (/* condition on states and inputs */) begin
         output_1 = // some expression of states and inputs
         output_2 = // some expression of states and inputs
         ...
      end else if (/* condition on states and inputs */) begin
        output_1 = // some expression of states and inputs
        output_2 = // some expression of states and inputs
        ...
      end else begin
        output_1 = // some expression of states and inputs
        output_2 = // some expression of states and inputs
        ...
      end
  end
  endmodule
```

Question 2. *Can I combine the register block and the two combinational block into a single always block?*

Yes, you can. Not recommended, but it works. Most students are doing everything in a single always block. It does not mean that you should. Remember, you want to generate a circuit from this HDL code. It is helpful for your understanding to write HDL code that corresponds to circuit blocks. You should periodically check the RTL diagram in the Netlist viewer.

```
// module keyword starts a module definition.
module module_named_foo(
    // Every module should have a single bit clock and single bit reset signal
    input wire [0:0] clock,
    input wire [0:0] reset,
    // All inputs to the module are declared as wires
    input wire [bits1:0] input_1,
    input wire [bits2:0] input_2,
    ...
    // All outputs from the module are declared as regs
    output reg [bits3:0] output_1,
    output reg [bits4:0] output_2,
);
// Every synchronous module will need some states
// States are always declared as registers
reg [bit5:0] state_1;
reg [bit6:0] state_2;
...

// Always block that triggers only on the posedge of clock and posedge of
// reset signal.
// ``always_ff'' is same as ``always'', but it ensures that a flip-flop circuit is
// synthesized.
always_ff @(posedge clock or posedge reset) begin
  if (reset) begin
    // This is the initialization block. You can assign initial values to your
    // state here
    state_1 <= 0;
    state_2 <= 0;
    ...
    // Using the non-blocking assign ``<='' in register block is recommended
  end else if (/*some condition on states and inputs */)begin
    // At the rising edge next state is copied to current state
    state_1 <= /* some expression of states and inputs */;
    state_2 <= /* some expression of states and inputs */;
    ...
    output_1 <= /* some expression of states and inputs */;
    output_2 <= /* some expression of states and inputs */;
    ...
  end
end
```

Question 3. *How to connect multiple modules in the top level module?*

Please refer to Lab 7 for details of instantiating modules. There is confusion about whether reg can connect to wires or not. reg CAN connect to wires and vice versa.

```
module module_top(input wire CLOCK_50,
                  input wire [2:0] BUTTON,
                  ...);

   // You can use wire and assign for simple combinational circuits.
   // One a wire is assigned it cannot be assigned anything else.
   wire reset;
```

```
    assign reset = BUTTON[1];

    // You can use wire to take the connect the output of one module to another.
    wire CLOCK_10;
    slowclock instance1_of_slowclock(CLOCK_50,
    reset,
    CLOCK_10);

    // Here wire CLOCK_10 connects the output of slowclock to the input of
    // foo
    module_named_foo instance1_of_foo( CLOCK_10,
                                       reset,
                                       ...
                                       ...);

endmodule
```

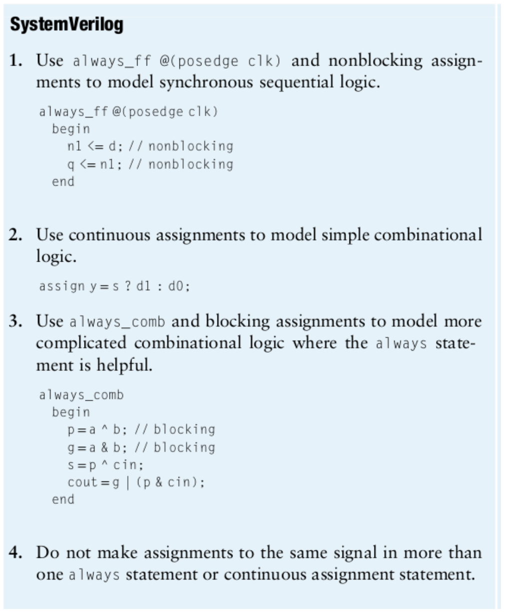
Question 4. *When to use register `reg` vs wire `wire`?*

Please refer back to Lab 6, when we learned about Verilog Procedural Operators. This is a quote from Lab 6 manual: "Another important aspect of the procedural always blocks is you would use registers on the left hand side of equations inside an always block. You would not use wires on the left hand side." In general, the following rules can help:

1. Inputs of a module inside the module are `wire`. They are declared such even when the keyword `wire` is ommitted.
2. Outputs of a module inside the module are `reg`. They are declared such even when `reg` is ommitted.
3. Different modules are typically connected through a `wire`.
4. Only use `assign` with a `wire` on the left hand side. You CANNOT `assign` a `wire` more than one time.
5. When a symbol is on the left hand side of a equation inside the always block, it must be a `reg`.
6. `reg` are more general than `wire`. When in doubt use a `reg`.

Question 5. *When to use continuous assign `assign` vs non-blocking assign "<=" vs blocking assign "="?*

The textbook has a very nice explanation of this usage in Section 4.5.4. I have reproduced the summary block here. In general, Chapter 4 is a useful read if you are still struggling with System Verilog programming.



SystemVerilog

1. Use `always_ff @(posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Use continuous assignments to model simple combinational logic.

```
assign y = s ? d1 : d0;
```

3. Use `always_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```
always_comb
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

Question 6. *What's the deal with `initial` block?*

You should only use `initial` block for simulation. It is a non-synthesizable block, so it will not be converted into a circuit. Instead, use a reset signal and an `if (reset)` block to initialize your states.