08/30/2021 10:16 AM, Preliminary
01/10/2022 12:26 PM

Coding in System Verilog

Coding in a Hardware Description Language (HDL) such as SystemVerilog is quite different than using a traditional software language such as C, C++, Python, or Java. While much of the syntax is similar, the rules surrounding the use of the syntax are different, and the end result of SystemVerilog code compilation is a description of digital logic behavior.

The purpose of this document is to instruct the user on methods to construct digital logic using SystemVerilog. This document is not exhaustive, and is intended to describe enough of the SystemVerilog syntax to enable users to create viable digital logic designs.

1. The abbreviation **SV** will be used for SystemVerilog throughout this document.
2. Best Practices will be highlighted through the use of lines above and below the text with the heading of Best Practices.
3. Important syntax rules will be highlighted through the use of lines above and below the text with the heading of Syntax Rules.

1. Understanding SV Modules

Start by imagining that you are creating a high level block diagram of your design. Each block will contain inputs, outputs, and a description of logic behavior. A complete design can then be constructed by connecting the individual blocks together correctly.

To use a C programming analogy, each program must have a single top level file or module called main. main can contain the entire program, but typically additional functions are created that are then called in the main module. Since main is the highest level or ranked module in the program, we say that functions called by main are at a lower level or hierarchy (or ranking). Each function can also called additional functions, and this nesting can be as simple or complex as necessary. We use the term **hierarchy** to describe this relationship of how modules or functions call other modules of functions.

A SV module can be thought of as a single block. One SV module must be at the top of the hierarchy. The inputs and outputs (or ports) of this top level block will be the inputs and outputs of your completed project. For our purposes (compiling to a target Field Programmable Gate Array or FPGA device), the top level module ports will connect to input and/or outputs pins of the target FPGA. Other lower level modules will of course have inputs and outputs, but these will be signals internal to the design.

A SV design of non-trivial complexity typically consists of a top level module, and additional modules that exist at various levels of hierarchy. When you place a module inside a higher level module, we term that as **instantiation**. Instantiation is an obscure term but is widely used in HDL design, and simply means that we are creating an instance of that specific module. Thinking in terms of hardware, each time a module is instantiated, the logic synthesized from that module becomes part of the overall logic design. If a module is instantiated multiple times, then multiple copies of that logic are created.

Best Practices

1. One file (with a .sv extension) should contain only one module.
2. The file name and the module name should be the same.

2. Module Declarations

A SV module normally begins with the keyword **module** and ends with the keyword **endmodule**. The module statement must include the names of the ports for that module. SV provides several approaches to declaring the ports, which will be shown below.

a. Port list contains only signal names

The inputs and outputs are defined in parentheses following the module name, but the direction and width of the ports are defined in following statements

```
module top (in1, in_vector2, out1, out_vector2);
input inp;
input [7:0] in_vector2;
output out1;
output [7:0] out_vector2;
```

b. Port list defines everything

```
module top (input in1, input [7:0] in_vector2,
            output out1, output [7:0] out_vector2);
```

SV ignores whitespace, so the module and port declarations can be formatted as the user selects. For example, the code in part b above could look like this. Note this example includes the **endmodule** statement

```
module top (input        in1,
            input [7:0]  in_vector2,
            output       out1,
            output [7:0] out_vector2
            );

// Your code goes here.................

endmodule
```

Syntax Rules

1. SV **is** case sensitive.
2. The **module** declaration must end with a semicolon.
3. The **endmodule** keyword is not followed by a semicolon.
4. The last item in the port list should not include a comma.
5. A single line comment begins with **//**, and can begin anywhere on a line.
6. A block comment begins with **/***, and ends with ***/**.

Best Practices

1. Use meaningful module and port names, this makes your code easier to read and understand.
2. Complex port names can use an underscore (_) to separate words, or you can also use UPPERCASE or CamelCase to enhance readability. Remember that SV **is** case sensitive.
3. Adding a comment block at the top of a module describing module behavior is very useful.

---

3. Concurrent vs Sequential Statements

When a software program is run, statements are evaluated one at a time in some defined order. Assume you have 10 statements that assign some values to 10 different variables. The first assignment will be made, followed by the second assignment, and so on until the 10th assignment is completed.

If you do the same thing in SV it is entirely possible that all ten assignments will be done concurrently (at the same time). A microprocessor is limited in that it has one chain of execution, and instructions are analyzed and acted on in order. Hardware does not have this limitation.

Every statement in SV is a concurrent statement, unless you explicitly instruct the SV compiler to analyze the code in a sequential manner. The two keywords that force this sequential code analysis are **always** and **initial**. Certain syntax requires the use of **always** or **initial** due to the inherent sequential nature of the syntax. An **if / else if / else** construct must be analyzed from the beginning and follow the correct order, otherwise the syntax yields meaningless results. It is this reason why certain keywords (or syntax) can only occur inside a sequential block.

always and **always** blocks

The always keyword is used to describe behavior that is triggered by specific events, and that behavior will repeat as long as the resulting digital logic is powered. Any clocked or synchronous logic will run as long as the clock is oscillating.

If more than one statement must be analyzed inside an **always** block, the use of **begin** and **end** is required, and we refer to this as an **always** block.

SV supports 4 unique **always** keywords.

a. **always_ff** should be used for all clocked or synchronous logic. This form of **always** must contain a sensitivity list (a list of signals that define events that cause the logic to activate, such as a clock). The **sensitivity list** should first contain the clock signal along with the active clock edge, and may contain addition signals that generally are asynchronous control signals. The sensitivity list is preceded by the **@** character, which is used to signify that an event in the sensitivity list will trigger the desired behavior.

While there can be multiple assignments to a signal inside an **always** block, it is not possible to make assignments to that same signal either outside the **always** block or inside a different always block. While this restriction makes little sense if viewed from a software perspective, remember that SV creates hardware. Each **always** block or concurrent assignment creates a unique electrical driver for that signal. Multiple assignments results in multiple drivers for a signal, and resulting electrical conflicts.

```
always_ff @ (posedge clock, negedge reset_n)
    if (reset_n == 0)
        something <= 0;
    else
        something <= something_else;
```

Syntax Rules

1. posedge is used to indicate a rising clock edge is to be used, negedge is used to indicate a falling clock edge is to be used.
2. The active level of a control signal is indicated by posedge (active high) or negedge (active low). The active level of the control signal must match the value used in the test of that signal. Using negedge reset_n in the sensitivity list but testing for a logic high (if reset_n == 1) will result in an error.
3. Use the non-blocking **<=** operator when **always_ff** is used.

---

b. **always_comb**

always_comb (combinatorial or non-clocked) should be used combinatorial logic will result (not latches, see below). This style of **always** should **not** have a sensitivity list. Use the blocking **=** assignment operator.

c. **always_latch**

Used when explicitly creating gated latches. Use of latches in FPGA designs must be analyzed carefully, and avoided if possible. This style of **always** should **not** have a sensitivity list. Use the blocking **=** assignment operator.

d. **always**

This form of **always** can be used in place of the three keywords above, but recommended practice is to use one of the three forms discussed above.

Syntax Rules

1. Assignment statements must begin with the **assign** keyword, unless the assignments are made inside an **always** or **initial** block. These assignments should use the blocking **=** assignment operator.
2. Use the appropriate blocking or non-blocking assignment operators as discussed above.
3. When an **always** or **initial** block contains multiple statements, these statements must be bracketed with **begin** and **end**. **case** and **if / else if / else** statements are considered to be single statements.
4. Using **begin** and **end** to bracket a single statement is acceptable, but unnecessary.

Best Practices

1. Do not use the **initial** keyword in synthesizable code. Only testbenches should use **initial**.
2. Avoid use of **always**, use the form that matches the type of logic your code will create.
3. Generally avoid use of latches.

---

4. Values, Numbers, Signals and Variables

1. A signal in SV can assume one of four values:
   1. 0 (logic low)
   2. 1 (logic high)
   3. z or Z (high impedance)
   4. x or X (unknown)
2. An integer value with no explicit radix or size is interpretted as a 32 bit unsigned decimal value. The specific number of bits can be controlled, and numbers can be entered as binary or hexadecimal values (in addition to decimal). Unassigned bits are set to 0. The underscore character can be used to to improve readability.

```
13'b1_0011_0101_0000    // 13 bit binary value
16'hABCD                // 16 bit hexadecimal value
10'd12                  // 10 bit decimal value

// 8'b1111_0000, 8'hf0, and 8'd240 are equivalent.
```

3. SV provides a data type **logic** that can be used for all types of assignments. Use logic for all internal signals that were either type **wire** or type **reg** in old-fashioned Verilog. Use of **logic** will eliminate 99% of confusion about which data type to use.

The 1% remaining has to do with outputs in modules. If your port list declares a port as an output, but the output is assigned in an **always** block, an error will occur. The error can be easily resolved by changing **output** to **output logic** in the port declaration.

If is possible to declare all outputs as **output logic** without considering how the output is assigned. This syntax should remove all confusion.

4. Scalar vs. Vector

SV allows creation of single bit signals (scalars) or multi-bit signals (vectors). A scalar can be thought of as a single wire, and a vector as a bundle of wires or a cable. Use of vector notation often reduces the amount of code required.

Signals are assumed to be scalars unless additional vector notation is added. A vector must define the range of indices. The indices are contained within square brackets, and separated by a colon. When declaring a vector (either a module port or an internal signal), the indices must be place in front of the vector name, but when referencing the vector in code, the indices must follow the vector name.

```
module top (input [7:0] in1, output logic [15:0] out1);

assign out1[15:0] = {in1[7:0], in1[7:0]};
```

The indices can be omitted if the full width of the vector is to be used. However, the width of the value on each side of the assignment operator must match.

It is not necessary that the leftmost index be larger than the rightmost index. As long as the use of index direction is consistent, no problems will occur.

5. Useful SV Syntax and SyntaxRules

1. Boolean Operators
   ~ (not), & (and), | (or), ^ (xor)

2. Arithmetic Operators

a + b, a - b, a * b, a / b, a % b

Best Practices

1. The divide and modulus operators are not synthesizable, so do not use these operators.
2. Use of the multiply operator create a lot of logic

---

3. Relational Operators (used for testing, returns 0 or 1)
   a == b // test for equality
   a == !b // test for inequality
   a > b // a greater than b
   a < b // a less than b
   a <= b // a less than or equal b
   a >= b // a greater than or equal b

4. Other useful Operators

Logical Shift {A << B}, {A >> B}
Concatenate {A,B}
Replicate {B{A}}

```
assign a[7:0] = 8'b1111_0000;
assign x[7:0] = a[7:0] << 3;      // x = 8'b1000_0000;
assign y[7:0] = a[7:0] >> 2;      // y = 8'b0011_1100;

assign a[3:0] = 4'b1010;
assign b[3:0] = 4'b1100;
assign x[7:0] = {a,b};           // x = 8'b1010_1100;

assign a[1:0] = 2'b01;
assign x[7:0] = {4{a}};          // x = 8'b0101_0101;
```

4. Conditional Assignment

x gets assigned the value of a if z is true (logic 1), otherwise x is assigned the value of b. An expression can be used for z, but the result of evaluating the expression must be a Boolean 1 or 0 value.

```
assign x = z ? a : b;
```

5. if / else if / else

The **if** and **else** clauses are required, one or more **else if** clauses can be included. **if / else if / else** clauses can be nested. Because this syntax must be evaluated by the SV compiler sequentially, an **always** block must be used.

```
if (something) begin
....... end
else if (something else) begin
......  end
else if (something else again) begin
......  end
else begin
......  end
```

Best Practices

1. If no **else** clause is included, latches will be inferred. Therefore, always include an **else** clause.
2. Deeply nested **if / else if / else** constructs are not recommended. Remember, hardware is being created, and it is very difficult to understand what is being synthesized from deeply nested constructs.

6. case
   A **case** statement can often be used in place of overly complex **if / else if / else** syntax, and are much easier to read and understand. Because this syntax must be evaluated by the SV compiler sequentially, an **always** block must be used.

```
case (a[2:0])
0:  this = that;
1:  begin  this1 = that1;  this2 = that2;  end
3:  state = go_go_go;
others:  state = idle;
endcase
```

Best Practices

1. Always include an **others** clause. If not all possible input combinations are used, latches can be inferred. The **others** clause will keep latch inference from occurring.
2. Assignments in the **case** statement should use the blocking **=** assignment operator.

---

7. loops

Best Practices

1. Do not use loops in synthesizable SV. The resulting hardware can explode in size.

---

6. Creating Registers and Counters

Registers and counters are similar in that they are constructed using flip flops, and have a width > 1.

To create a register that can be used to hold a value:

```
logic [15:0] my_register;

always @ (posedge clock, negedge reset_n)
    if (reset_n == 0)
        my_register <= 0;
    else
        my_register <= something;
```

A counter is simply a register that is assigned a value based on the current value.

```
logic [15:0] my_counter;

always @ (posedge clock, negedge reset_n)
    if (reset_n == 0)
```

```
        my_counter <= 0;
    else if (direction == up)
        my_counter <= my_counter + 1;
    else
        my_counter <= my_counter - 1;
```

Best Practices

1. If a register does not include a reset signal, simulations will report the output value as unknown until an assignment is made.
2. If a counter does not include a reset signal, simulations will report the output as unknown forever. Adding or subtracting some value from an unknown value yields an unknown value.
3. Including a reset in the sensitivity list as shown above results in an asynchronous reset. If reset is omitted from the sensitivity list, a synchronous reset will result.

7. Use of Parameters

To be added.

8. Testbenches and Simulation

In order to simulate a SV design (a single module, or a complete system), a method must be used to provide simulated stimulus to the design under test, and a method must be provided to read the resulting outputs. The approach used is to create a simulation only file called a testbench. The testbench will instantiate the design to be tested, provide the stimulus, and display or capture the resulting behavior.

It is only necessary to instantiate the top level module to be tested in a testbench. A testbench contains no input or output ports, so no logic is synthesized.

In a simple testbench, input stimulus is applied at defined times, and outputs become valid at some later time. A register transfer level or RTL simulation, actual logic element and routing delays are not considered, but the simulator does keep the correct order of logic changes in order to create an accurate simulation. Gate level or timing simulations model actual logic and routing delays in the simulation.

A testbench is a SV module that contains non-synthesizable constructs. Since simulation time if advanced in simulation time units, the basic time unit must be defined using a **`timescale** keyword. Signals of type **logic** must be defined that correspond to the design under test inputs and outputs. A clock signal (if the design is synchronous) must be generated, and

```
`timescale 1 ns / 100 ps

module tb();

/* Create logic types for all inputs and outputs of design under
   test (DUT).  For example.....
*/
```

```
logic clock;
logic in1, in2, in3;
logic [99:0] big_input;
logic out1, out2, out3;
logic [199:0] really_big_output;

/* Instantiate DUT.
   The .* syntax can be used if the input and output signals above
   have identical names to the ports of DUT.
*/

my_module DUT (.*);

// Create a clock (if necessary)

parameter half_clock_period = <time in ns>;
always #half_clock_period clock = ~clock;

// Define values for all inputs at t = 0;

initial begin
    clock = 0;
    in1 = 0;  in2 = 1; in3 = 0;
    big_input = 99'b0;

/* Move time forward using #delay, and modify input values.
   For example ......
*/
    #100 in1 = 1;
    #500 in2 = 0;
    #150 in3 = 1;
    #20000 big_input = 99'd1234;

// Run simulation as needed, then suspend simulation
    #10000 $stop;

end

endmodule
```

9. Designing with FPGAs

1. Combinatorial decoding can cause glitches on signals

Logic design uses Boolean gates (AND, OR, NOT) to combine and decode signals. FPGAs map the Boolean functions into lookup table structures that can be programmed as needed. Anytime more than one input to a lookup table changes simulataneously it is possible the output of the lookup table with glitch. Complex decoding may require multiple levels of lookup tables. Combining outputs of multiple lookup tables can also cause glitching, and the routing delays from multiple lookup tables will vary depending on placement.

A designer needs to determine if a possible glitch on a signal is critical or not. If a decoded signal is to be used as a clock to other parts of the logic, glitches can result in incorrect and unpredictable behavior. In order to remove possible glitches, outputs should first be registered.

2. Synchronous reset removal (or deassertion) is critical

When an FPGA is first configured (programmed), an internal state machine eventually removes a globally generated reset signal in order to allow the FPGA circuitry to begin operation. The removal of this reset is asynchronous to any clocks in the system. A side effect of this asynchonous reset removal is that some synchronous elements may see reset removed prior to or after other synchronous elements.

Every design should contain a well controlled reset that is removed synchronously from all flip flops. This sounds easier than it actually is. Systems often contain multiple clocks, phase lock loops, and large FPGAs may have routing delays that do not allow a single signal to successfully propogate to all logic elements in a single clock cycle.

At a minimum, a user reset signal should be created, and the reset signal should be removed in a synchronous manner.

3. Understand FPGA resources

```
assign a = b / c;
```

See how easy it is to construct a divider in SV? But, what actually gets created?

FPGAs often contain arithmetic blocks to allow creation of fast adders, subtractors, and often these blocks include hard multipliers. However, no FPGA contains a divider block. A synthesis tool may be able to create a divider, but the resulting circuit will be large and slow.

Use of a division operator (among others) can be helpful when constructing non-synthesizable code such as testbenches, but should not be considered good design practice for synthesis.

FPGA documentation provides coding templates for the corresponding compilers that will guide the compilers to use available resources in the best possible manner.

FPGAs contain memory blocks. Correct coding will result in proper use of those blocks, incorrect coding may result in the use of thousands of flip flops instead.

5. Do not gate clocks

Gating clocks is a easy way to generate glitches on clock signals, which can result in incorrect clocking of logic resources. If clocks must be turned off, either take advantage of built in clock enables (if available) or design circuitry that can cleaning turn clocks on or off without glitching.

6. Avoid asynchronous design

Aynchronous design relies on delays of logic elements to properly sequence the logic. However, FPGAs add an additional design element in that the delays between these elements can vary based on design placement and how the routing is done. These variables mean that a design can change timing every time the FPGA design is recompiled. Using time constraints to manage this is difficult, and attempting to lock down placment and routing is often self-defeating.