Reference Docs

SystemVerilog Tutorial

Updated: March 2, 2020. Version: SystemVerilog 2017

Introduction

Verilog is a hardware description language (HDL), which means that it is used to define or "build" hardware within a programmable device, such as a field-programmable gate array (FPGA). This is important because it means that Verilog code is NOT executed sequentially, line-by-line, like typical programming languages. Statements in Verilog define *hardware*, and events in hardware happen simultaneously/concurrently. (Think of a circuit with multiple IC chips and wires. The electrons flowing down the wires all move at the same time, not one after the other.) SystemVerilog (SV) is the most recent version of Verilog. It modifies the language slightly and adds the ability to do verification. For more, see About SystemVerilog.

Verilog can be used to do:

- Behavior modeling
- Gate-level modeling
- Transistor-level modeling
- Simulation for all the modeling
- Verification of designs (SystemVerilog only)

References

1. Verilog Tutorial, chipverify.com (Verilog 2001)
2. Quick Reference Guide (Verilog 2001)
3. SystemVerilog Tutorial, chipverify.com
4. SystemVerilog Tutorial, verificationguide.com
5. FPGA Prototyping By Verilog Examples, Chu (Verilog 2001)
6. FPGA Prototyping by SystemVerilog Examples, Chu

Unfortunately, most SystemVerilog (SV) references you will find on the internet (including the ones above) assume you already know Verilog, and of course, none of the Verilog references include any information about SV. This document will provide explanations of improvements/changes that SV makes to the Verilog 2001 standard.

Part 1 - The Basics

Modules

See Verilog Module or Module Definition (page 12).

Modules are the basic building blocks of Verilog. They represent a unit with inputs and outputs (or "ports"). Modules that do not have IOs are used for testing only. You should always have only **one module per Verilog file** (.v or .sv).

```
module module_name
#(parameter_declaration,...) // optional (explained in Part 2)
(port_declaration port_name, port_name, ...,       // Inputs and
 port_declaration port_name, port_name, ... );     // outputs

// --- Design code ---
// code for what is inside the module goes here...

endmodule
```

For example, the following is a module called *halfadder*, with two input ports named *a*, *b*, and two output ports *s*, *c*.

```
module halfadder (input a, input b, output s, output c); // <-- always
                                                         // ends with a ";"
// ...
endmodule // halfadder
```

You can group ports that are the same type together.

```
module halfadder (input a, b, output s, c); // still ends with a ";"
// ...
endmodule // halfadder
```

A long list all in a row can get hard to read, so separate lines is better. The tools will still read the file exactly the same, but this makes it easier for us humans.

```
module more_ports (
   input a, b,
   input q, r, s,
   output s, c,
   output t, u, v
); // still ends with a ";"
// ...
endmodule // more_ports
```

Data Types

See Data Types or Data Type Declarations (page 15).

The two that are most important for now are:

`wire`:

- Just like a physical wire, connects an output to an input

- Continuously driven by the output to which it is connected
- Used in IO ports by default

`reg`:

- Used in test benches for inputs (because they are defined inside `initial` blocks)
- Will be discussed more in Part 2

For signals (`wire`'s and `reg`'s) inside a module, you should always *declare* them before using them. Declare means to state that they exist (and define the number of bits, see the section on Vectors). If you do not, the tools will assume that you want a 1-bit wire. A problematic example of this is given in the section on Vectors.

Wrong way:

```
module data_types(input a, b,
    output q);

   // Assume w exists - BAD!

   assign w = a & b;
   assign q = w;

endmodule // data_types
```

Right way:

```
module data_types(input a, b,
    output q);

   // Declare w - GOOD!
   wire w;

   assign w = a & b;
   assign q = w;

endmodule // data_types
```

Logic Values

See Data Types: Values or Logic Values (page 10).

Verilog uses a 4-value logic system for modeling. 'X' often appears during testing if you forget to connect something or don't specify its value properly.

Value	Description
0	zero, low, or false
1	one, high, or true
z or Z	high impedance (tri-state or floating)
x or X	unknown, uninitialized, or don't care

Literal Integer Numbers

See Number Format or Literal Integer Numbers (page 11).

When you type in a number, we say that it is "hard coded" into your project or that it is a numeric "literal." Literals use the following format:

`<size>'<base><value>`

where

- `size` = number of bits for the binary equivalent
- `base` = one of the 4 options below
- `value` = the value of the number in the specified `base`

Bases:

Radix	Symbol	Legal Values
Binary	`'b`	0, 1, x, X, z, Z, ?, _
Octal	`'o`	0-7, x, X, z, Z, ?, _
Decimal	`'d`	0-9, _
Hexdecimal	`'h`	0-9, a-f, A-F, x, X, z, Z, ?, _

Examples:

Examples	Size	Radix	Binary Equivalent
`10`	unsized	decimal	`0...00001010` (32-bit)
`'o7`	unsized	octal	`0...00000111` (32-bit)
`1'b1`	1 bit	binary	`1`
`8'hAB`	8 bits	hexadecimal	`10101011`
`6'hF0`	6 bits	hexadecimal	`110000` (truncated)
`6'hA`	6 bits	hexadecimal	`001010` (zero filled)
`6'bz`	6 bits	binary	`zzzzzz` (z filled)

If you do not specify a size (i.e. "unsized"), it will default to 32 bits or larger. Thus, a number by itself will be interpreted as a 32-bit (or larger) decimal integer. A problematic example of this is given in the next section on Vectors.

Vectors and Bit Selection

See Scalar and Vector or Vector Bit Selects (page 20).

Multi-bit signals are referred to as "vectors" and are defined using square brackets. Note several things:

- In order for signals to have different bit widths, they have to be separate `inputs`, `outputs`, `wires`, etc.
- Define the width (number of bits) using square brackets between the data type/port declaration (`input`, `wire`, etc.) and the signal name.
- Choose bits from already-defined vectors using square brackets *after* the name. (This is very similar to Matlab, but brackets instead of parentheses.)
- Vectors can be defined in any bit order (e.g. `wire[6:4] a` or `wire[2:9] b`), but most often we are going to put the MSB on the left and count down to 0 (e.g. `wire[3:0] a`).

```
module vectors (
   input[3:0] a, b,  // Both a and b will be 4 bits.
   output c,         // In order for ports to have different bit widths,
   output[7:0] d     // they have to have a seperate 'output' keyword.
   );

wire[3:0] e, f, g;   // Like inputs and outputs, wires and regs have to be
wire[1:0] h;         // separated to be different widths.

assign c = a[2];     // Select a single bit from vector a

assign h = a[1:0];   // Select multiple bits from vector a (bits 1 and 0)

assign e = b;        // Assign all bits of b to e (don't have to use brackets
                     // if using all bits)

assign d = {a,b};    // Concatenate (join) a and b together to form d
```

```
assign f = 8'hBA;     // This is a valid but problematic statement.
                      // Only the 4 LSBs ("A") will be assigned to f,
                      // because f is only 4 bits.

assign j = 4'hA;      // This is a valid but problematic statement.
                      // Only one bit (LSB) will be assigned to j,
                      // because it was not declared and thus
                      // defaults to a 1-bit wire.

endmodule // vectors
```

Operators

See Operators or Operators (page 33).

We will cover more about operators in the future, but here are the basic ones:

```
module halfadder (input a, b,
                  output s, c);

// This statement is called a "continuous assignment"
// s must be a "wire" type
assign s = a ^ b;     // XOR
assign c = a & b;     // AND

endmodule // halfadder
```

The other basic operators are ~ = NOT and | = OR.

This style of coding is called "behavioral" modeling because you are not actually specifying the gates to be used but rather the behavior that you want to accomplish.

Primitives

See Gate-level Modeling or Primitive Instances (page 19).

Another way to define operations is to specify the exact gate, which is called "gate-level" modeling.

```
module halfadder (input a, b,
                  output s, c);

// When using primitive instances, output is always the first port
xor(s, a, b);
and(c, a, b);

endmodule // halfadder
```

The other primitives are `not` and `or`.

Module Instances

See Module Instantiations or Module Instances (page 21).

Basic syntax:

```
module_name instance_name (.port_name(signal), .port_name(signal), ... );
```

Example:

```
module fulladder (input a, b, cin,
                  output sum, cout);

//Internal wires that connect between the half adder modules
wire s1, c1, c2;

halfadder HA1(.a(a), .b(b), .s(s1), .c(c1));
// The a, b, s, c outside the parentheses are the IO ports of halfadder
// module. Inside parentheses are IO ports or internal wires of this
// module (fulladder).

halfadder HA2(.a(s1), .b(cin), .s(sum), .c(c2));

assign cout = c1 | c2;  // or(cout, c1, c2);

endmodule // fulladder
```

Testing

See Combinational Logic Examples or Simulation Basics.

When you build a module, you should test it to verify that it works as you intended. You can do this by programming hardware, but it's more convenient to do it with simulation first. We call this a "test bench" for the module we are testing.

```
module halfadder_test();
   // Note the lack of IO ports. None needed for a test bench.

   // Instead, we need some internal signals:
   reg a_in, b_in;     // Since we need to change the inputs, they need to
                       // be declared as reg.
   wire c_out, s_out;  // Since the outputs will be driven by the halfadder
                       // instance, they need to be declared as wire.

   // Instance of the module we want to simulate
   halfadder HA(.a(a_in), .b(b_in), .c(c_out), .s(s_out));

   initial
      begin
         // Test case #1
         a_in = 0;
         b_in = 0;
         #10;      // Tells simlulator to wait for 10 ns. We need this so
                   // that we can observe the output for this test case.

         // Test case #2
         a_in = 0;
         b_in = 1;
         #10;

         // Alternatively, we can specify a_in and b_in together:
         {a_in, b_in} = 2'b10;
         #10;
```

```
        // To shorten the code, we can put it all on one line, but don't
        // forget the semicolon after each statement!
        {a_in, b_in} = 2'b11;  #10;

        $finish;  // If you know exactly how long the test needs to run,
                  // stop the simulation when it is done.
   end
endmodule
```

Part 2 - Combinational Logic

Conditional Operator

See Operators (page 33).

Verilog provides a compact statement for a simple true/false condition using the "`?`" conditional operator. Generically, we could write it as

```
(output) = (condition) ? (if_true) : (if_false);
```

This is equivalent to an *if/else* statement. Here is an example that assigns `out = a` if `sel == 1` (true), otherwise `out = b`.

```
assign out = sel? a : b;
```

Another example is provided below.

Data Types Again

See Data Types or Data Type Declarations (page 15).

`reg` - more information:

- Used inside procedural blocks (`always` or `initial`)
- Only updated periodically (not continuously like a `wire`)
- Must be declared *outside* of procedural blocks.
- The variable needs to hold (store) a value, so if your Verilog code does not define a register, the tools will add one. That is, a `reg` infers data storage if needed.
- When used in IO ports, only outputs can be declared as `reg`.

Blocking vs. Non-blocking Assignment

See Procedural Assignment Statements (page 29). They provide a good explanation of the difference, so it won't be repeated here. If you want to go deeper, with examples, see Blocking/Non-blocking.

Summary: `reg` variables can be assigned with either `=` (blocking) or `<=` (non-blocking). We will use `=` for combinational logic and `<=` for sequential logic. More on this in Part 3.

Procedural Blocks

See Always Block or Procedural Blocks (page 27).

There are two types of *procedural* blocks: `always` and `initial`. They are called "procedural" because unlike Verilog code outside of these blocks, the code within them is executed in order (sequentially, like steps in a process). `initial` executes its code one time. `always` runs its code over and over in an infinite loop.

Sensitivity Lists

The timing of when an `always` block is executed can be controlled by a list of signals, called the "sensitivity list." That is, the block is *sensitive* to those signals and executes when they change. This list will become very important when we get to sequential logic.

For combinational logic, we want all of the signals used in the block to be listed because those statements need to be executed whenever *any* of the signals change. There is a nice short-hand notation for this which is `@*` ("at" followed by "star/asterisk"). This method is preferred because with the other method you are likely to forget a signal in the sensitivity list.

Explicit listing (not preferred):

```
always @(a, b)
   begin
      s = a ^ b;
      c = a | b;
   end
```

Include all signals (preferred for combinational):

```
always @*   // all signals
   begin
      s = a ^ b;
      c = a | b;
   end
```

Programming Statements

See Control Flow and Case Statement or Procedural Programming Statements (page 30).

The primary two statements we will use are `if` and `case` statements. An example of these, along with a conditional assignment statement, are given in the following code. All of these implement a 1-bit, 2-input multiplexer (MUX).

Conditional operator:

```
module mux(
   input in0, in1, sel,
   output out
   );

assign out = sel?
             in0 : in1;

endmodule // mux
```

`if` statement:

```
module mux(
   input in0, in1, sel,
   output out
   );

always @*
   begin
      if (sel == 1)
         out = in1;
      else
         out = in0;
   end

endmodule // mux
```

`case` statement:

```
module mux(
   input in0, in1, sel,
   output out
   );

always @*
   begin
      case (sel)
         1'b0: out = in0;
         1'b1: out = in1;
      endcase
   end

endmodule // mux
```

Begin and End?

You will notice in the examples above that some places have a `begin` and `end` keyword and some do not. You need `begin-end` whenever you have more than one statement inside of another (just like curly braces in C/C++). Here are several examples:

```
case (q)
   2'b00:
      begin // required
         r = 1'b0;
         s = 1'b0;
      end
   2'b01:
      begin // not required, but allowed
         {r,s} = 2'b01;
         // this is a way to assign bits to multiple individual signals
      end
   2'b10:
      // not required, so not included
      {r,s} = 2'b10;
   2'b11:
      // technically each "if" is one statement, so begin-end not required,
      // but you must use good indentation!
      if (a)
         if (b)
            {r,s} = 2'b11;
         else
            {r,s} = 2'b10;
      else
         {r,s} = 2'b01;

      // It could also be written like this, but it's harder to read
      if (a)
         if (b) {r,s} = 2'b11; else {r,s} = 2'b10;
      else
         {r,s} = 2'b01;

   default: // it's a good idea to always have a default case,
            // and a good practice to use "z" for an undesired value
      {r,s} = 2'bzz;

endcase // (q)
```

All the Same?
While the example code above accomplishes the same task, there are subtle but sometimes important differences in the hardware that is generated from your code. For example, nested `if` statements create a series of 2-input MUXs that cascade from one to the next, creating a "priority routing network" where later inputs have a higher priority than earlier inputs. In contrast, a `case` statement creates single, multi-input MUX so that all inputs have the same priority, as shown in Figure .1. Differences like these become more important the farther into digital design you go.

Errors to Avoid
From section 3.7 of FPGA Prototyping By Verilog Examples, Chu.

Avoid doing the following:

- Assigning a value to a `reg` signal in more than one `always` block – This is equivalent to connecting a wire to more than one output. Outputs don't like to be driven by other outputs.
- Leaving a signal out of the sensitivity list – This applies to both combinational and sequential logic. For combinational logic, use `@*`. Sequential logic is covered in the next section.

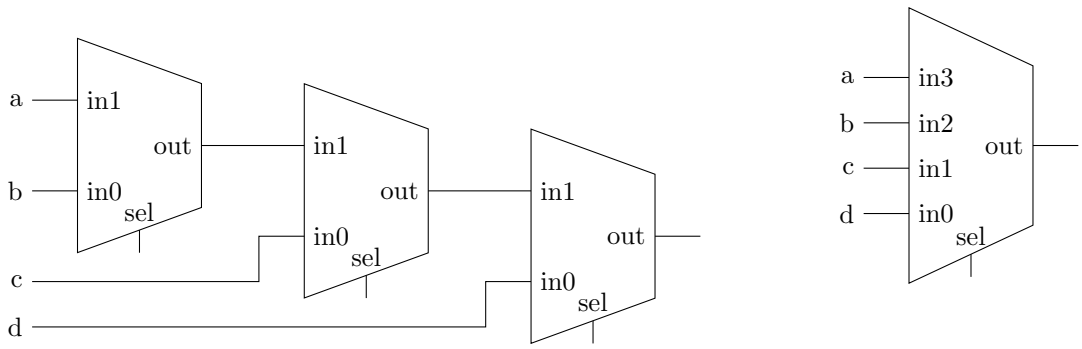

Figure .1: Priority network vs. "parallel" multiplexer

- Not assigning a value to a `reg` signal – By definition in Verilog, if you don't assign a value to a `reg`, that signal *holds* its previous value. This means your code will *infer* (i.e. create even though you don't want it to) memory elements for any signals that are not assigned a value. So make sure you always:
  - Include all branches (possible values) of an `if` or `case` statement
  - Assign a value to every output signal in every branch

Here is an example demonstrating the last rule:

Incorrect!

```
if (a)
   r = 1'b1;

// what happens when
// a == 0 ? What will
// be the value of r?
```

Correct

```
if (a)
   r = 1'b1;
else
   r = 1'b0;
```

Correct

```
// assign default value
r = 1'b0;

if (a)
   r = 1'b1;
```

Here is similar example with `case` statements:

Incorrect!

```
case (c)
   2'b00: s = 2'b11;
   2'b11: s = 2'b00;
endcase

// what happens when
// c == 01 or 10?
```

Correct

```
case (c)
   2'b00: s = 2'b11;
   2'b01: s = 2'b10;
   2'b10: s = 2'b01;
   2'b11: s = 2'b00;
endcase
```

Correct

```
// assign default value
s = 2'bzz;
case (c)
   2'b00: s = 2'b11;
   2'b11: s = 2'b00;
endcase

// or   (same result)
case (c)
   2'b00: s = 2'b11;
   2'b11: s = 2'b00;
   default: s = 2'bzz;
endcase
```

Part 3 - Other Items

Parameters

See Verilog Parameters or Parameter (page 19).

As a constant (`localparam`)

Parameters are extremely useful constructs. In the simplest form, they allow you to give a "name" to a value that you want to use repeatedly (called a "constant" in other languages). Think of this as a simple replacement: the tool (Vivado) replaces the parameter (name/text) with its value.

Note that the value of a `localparam` is constant; it cannot be changed anywhere else in the code. Using numbers directly, as on the left below, is often referred to as "hard coding" values into your designs. We want to avoid this when possible to make code more flexible and reusable.

Without a parameter:

```
module localparam_ex(
   input [7:0] a, b,
   output [7:0] q
   );




reg [7:0] w;

always @* begin
   if (a > 8'd5)
      w = b + 4'b0011;
   else
      w = b;

   // multiply by 2^2
   w = w << 2;
end

assign q = w;

endmodule // localparam_ex
```

Using a parameter:

```
module localparam_ex(
   input [7:0] a, b,
   output [7:0] q
   );

localparam W_BITS = 8;
localparam A_COMPARE = 8'd5;
localparam B_ADD = 4'b0011;
localparam SHIFT_N = 2;

reg [W_BITS-1:0] w;

always @* begin
   if (a > A_COMPARE)
      w = b + B_ADD;
   else
      w = b;

   // multiply by 2^N
   w = w << SHIFT_N;
end

assign q = w;

endmodule // localparam_ex
```

Part of module definition (`parameter` and overriding)

Another way that parameters make designs more flexible is the ability to set their value separately for each module, known as "redefining" or "overriding" the value of the parameter. We could modify the example above to be:

Parameter defined as part of module:

```
module param_ex
   #(parameter BITS=4)
   (
   input [BITS-1:0] a, b,
   output [BITS-1:0] q
   );

localparam A_COMPARE = 8'd5;
localparam B_ADD = 4'b0011;
localparam SHIFT_N = 2;

reg [BITS-1:0] w;

// ...same as above...

endmodule // param_ex
```

Instantiation (overriding parameter):

```
// ...in another module...

// instantiate and set value of
// BITS = 8
param_ex #( .BITS(8) ) name (
   .a(wire_a), .b(wire_b),
   .q(wire_q)
   );

// ...more code...
```

Here is another, more practical example using a multiplexer. Note that the default value for the bit-width of the MUX is set in the module as 1. But when instantiating, you can set this to any value you want, allowing you to reuse the same module in lots of situations.

Module definition:

```
module mux2
   #(parameter N=1) // default
   (
   input [N-1:0] in0, in1,
   input sel,
   output [N-1:0] out
   );

assign out = sel?
             in0 : in1;

endmodule // mux2
```

Instantiate with different parameter values:

```
module top_level(
   input [3:0] a, b, c, d,
   input sel_4bit, sel_8bit,
   output [7:0] output
   );

wire [3:0] wire_ab, wire_cd;

// two 4-bit mux2's
// (parameter N = 4)
mux2  #( .N(4) )  mux_ab  (
   .in0(a), .in1(b),
   .sel(sel_4bit),
   .out(wire_ab)
   );
mux2  #( .N(4) )  mux_cd  (
   .in0(c), .in1(d),
   .sel(sel_4bit),
   .out(wire_cd)
   );

// one 8-bit mux2
// (parameter N = 8)
mux2  #( .N(8) )  mux_8bit (
   .in0(wire_ab),
   .in1(wire_cd),
   .sel(sel_8bit),
   .out(output)
   );

endmodule // top_level
```

Array of Instances

See Module Instances (page 21).

This is not a necessary topic for you to know, but it could be useful. If you want to define lots of instances of the same module, you can create an "array of instances" similar to a vector signal.

Here is an example. The register on the left has an 8-bit input and output and two 1-bit inputs. The module on the right instantiates 4 copies of the 8-bit register and then connects the 32-bit input `in` to the 4 x 8 = 32 bits of the `D` inputs and the 32-bit output `out` to the 4 x 8 = 32 bits of the `Q` outputs. The single bit (scalar) signals `clock_in` and `enable_in` are connected to all of the single-bit inputs on the 4 modules.

Register module:

```
module register_8bit (
   input [7:0] D,
   input clk, enable,
   output [7:0] Q,
   );

   // register code...

endmodule // register_8bit
```

Instantiated in another module:

```
module some_other_module (
   input [31:0] in,
   input clock_in, enable_in,
   output [31:0] out
   );

   register_8bit r[3:0] (
      .D(in),
      .clk(clock_in),
      .enable(enable_in),
      .Q(out)
      );

   // rest of design...

endmodule // some_other_module
```

As another example, we can use the *fulladder* defined earlier to create a ripple adder. Better yet, we could use a parameter to make this module whatever size we want. (Note this is a very efficient way to write code, which is the point here, but a ripple adder is not the best design for an adder.)

Without parameter:

```
module ripple_adder
   (
   input [7:0] in_a, in_b,
   input cin,
   output [7:0] sum,
   output cout
   );


   wire [7:0] carry;

   assign carry[0] = cin;

   fulladder adders [7:0] (
      .a(in_a), .b(in_b),
      .cin(carry),
      .sum(sum),
      .cout({cout, carry[7:1]})
      );

endmodule // ripple_adder
```

With parameter:

```
module ripple_adder
   #(parameter N=1)
   (
   input [N-1:0] in_a, in_b,
   input cin,
   output [N-1:0] sum,
   output cout
   );

   wire [N-1:0] carry;

   assign carry[0] = cin;

   fulladder adders [N-1:0] (
      .a(in_a), .b(in_b),
      .cin(carry),
      .sum(sum),
      .cout({cout, carry[N-1:1]})
      );

endmodule // ripple_adder
```

Part 4 - Sequential Logic

To come...