Reference Docs

SystemVerilog Tutorial

Updated: February 5, 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`:

- Interconnecting wire, connect output to input

- Its value is driven by the output it is connected to.
- Used in IO ports by default
- Used outside procedural blocks (more later)
- Any variable that is not declared is assumed to be 1-bit wire.

`reg`:

- The driver (behavior) of any 'reg' variable must be defined in the module where the variable is declared.
- Used inside procedural blocks
- But must be declared outside procedural blocks
- *Note*: reg is *not* a register. Used to be, not anymore.
- When used in IO ports, only outputs can be declared as 'reg'

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.

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 - Procedural

To come...