

SystemVerilog Cheat Sheet

Avinash Uttamchandani
avinash@nonholonomy.com

1 A Hardware Description Language

Verilog is a hardware description language (HDL). The important thing to remember with Verilog is that every line you write creates actual gates and does not execute sequentially. It is always best practice to know what digital circuit you want to create *before* you start coding.

SystemVerilog is a “newer¹” version of Verilog that supports advanced features and far simpler ways of denoting combinational and sequential logic, the rest of this document assumes you are using SystemVerilog (which is supported by any modern closed or open source tool).

Caveat emptor² - This guide started in 2011 (so might always be a bit outdated), prioritizes both open-source and proprietary implementations (so it lags on some features), and last but not most important, it’s a cheatsheet! If you want to know this well, get the books listed in the Resources section and be ready for some dense reading.

Last but not least, some important definitions - for various historical and practical reasons Verilog supports different features for either **simulateable** (never will be on hardware) and **synthesizable** (can be a real circuit (FPGA, ASIC, etc.)). If you are ever unclear about the difference, ask! But as a rule anything “structural” is synthesizable, anything “behavioural” is simulateable, and *some* but not all behavioural is synthesizable.

2 Types in SystemVerilog

There are two³ useful “types” in Verilog that map to real hardware: **wire** and **logic**.

Use the **wire** type for connections between gates, or inputs to modules. Use the **logic** type for anything that *may* be driven by a logic gate (combinational or sequential).

Note that all “nets” aka “signals” carried by **wire** and **logic** elements can be in one of four states:

1 Logic high, or VDD.

0 Logic low, or GND.

z High Impedance (aka floating). Used to implement tristates, etc.

x an unknown value.

¹I mean, as of 2012, but who’s counting.

²Fancy speak for “buyer beware.”

³I’m omitting **reg** and **bit** for good reasons (see Sutherland Appendix A 3-3).

- In combinational logic this typically means something is unconnected. Check your wiring!
- In sequential logic this can also be an uninitialized value. Check your reset logic!

2.1 Older Verilog

Note that in older Verilog you had to use `wire` for `assign` statements. This guide will forgo that in favor of the newer `alwayscomb` statement with `logic` types. You might see the `reg` keyword, but you can assume it's the same as the `logic` type for most purposes.

3 Buses

Most quantities that you'll want to work with when designing blocks are going to need more than one bit to be represented. In verilog, you can do this with buses. To create a bus, enter the bus type (`wire`, `logic`), the bus size ([most significant bit: least significant bit]), and finally the name (or names if you have multiple wires of the same size). For module ports you must prefix the definition with the signal's direction (`input`, `output`, or `inout`). Some examples:

```
wire [1:0] e,f;           // Two 2-bit wires.
wire g,h,i;             // Verilog assumes 1 bit wide wires by default.
input wire [3:0] a, b, c; // Three inputs (a,b,c) that are all 4 bits wide.
output wire [7:0] d;      // One 8 bit output.
```

WARNING: The width must go before the name! If you put it after you create a memory instead of a bus (with very different hardware implications).

You can access a single bit of the bus with standard C style array indexing. You can also grab *slices* of a bus by specifying a bit span with square brackets and colon. Finally you can concatenate wires into a bus with curly brackets:

```
wire a;
wire [1:0] b;
wire [3:0] c, d;
wire [2:0] e;
wire [6:0] f;

always_comb a = b[0] ^ b[1]; // Accessing bits of bus b individually.
always_comb d = c[1:0] & c[3:2]; // And-ing the lowest two bits of c with the
    highest two bits.
// This is the concatenation operator:
always_comb f = {a, b, c}; //f[0] = c[0], f[1] = c[1], f[2] = c[2],
    //f[3] = c[3], f[4] = b[0], f[5] = b[1],
    //f[6] = a
```

4 Constants

Every 'number' in verilog is actually a collection of bits, so it's important to set constants correctly. Constants in Verilog are written in the form of radix'constant. Radix is the base

(b for binary, d for decimal, h for hexadecimal), and the constant needs to be written in the appropriate base. Here are some examples:

```
logic [7:0] a, b, c;

// a, b, and c are all assigned to decimal 224.
always_comb begin : constants_example
    a = 8'b1110_0000; //a = 224 in decimal
                        //use underscores to make numbers more reasonable
    b = 8'hE0;
    c = 8'd224;
end

logic [3:0] d, e;
// d and e are equivalent because if you try to always_comb to a bus that's
// smaller than the value, only the least significant bits will go through.
always_comb begin : truncation_example
    d = 8'b11001010;
    e = 4'b1010;
end

// If you want all the bits set to be one, you can do it a few ways:
logic [N-1:0] f, g, h;
always_comb begin : all_ones_example
    f = -1; // Uses two's complement definition.
    g = 'b1; // Not supported on all tools.
    h = {N {1'b1}}; // Uses the *repetition* operator.
end

// And for signed/two's complement values, if you declare the net
// as signed you can put a - sign before a constant definition.
signed logic [7:0] i;
i = -8'd2; // will be 1111_1110
```

4.1 Enums and Defines

Sometimes you want to define a constant that can be used in multiple places. There are a few good ways to do that. A common, if slightly old school, way is with a “macro”.

```
// Define a constant.
`define A_CONSTANT 8'b1010_0101

// Use it later.
always_comb some_logic = `A_CONSTANT;
```

NOTE: Macros are just text substitution, that's why we don't put a semicolon at the end of them. In general they are a very “shoot yourself in the foot” type of tool, it's easy to do things that are hard to debug. But, they can do more than the subsequent safer methods, so they're still useful from time to time.

If you can, use the `const` keyword:

```
const logic [7:0] A_CONSTANT;
```

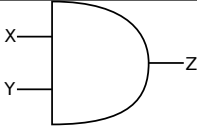
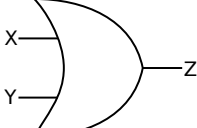
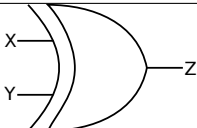
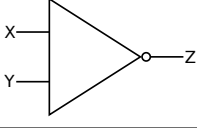
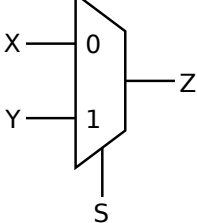
```
always_comb some_other_logic = A_CONSTANT;
```

Last, if you want to allocate a bunch of named constants to a bus (most common example being the states of an FSM), you can use the `enum` keyword. This is best used in combination with `typedef` to create a new aggregate type!

```
typedef enum logic [1:0] {IDLE, BUSY, DONE, ERROR} state_t;
state_t state; // is a logic [1:0].
always_ff @(posedge clk) begin
    case(state)
        IDLE: state <= BUSY;
        BUSY: state <= DONE;
        DONE: state <= IDLE;
        default: state <= ERROR;
    endcase
end
```

5 Structural Gates

The following is a list of gates and their structural verilog equivalents.

Gate	Schematic	Structural Verilog
and		<code>always_comb Z = X & Y;</code>
or		<code>always_comb Z = X Y;</code>
xor		<code>always_comb Z = X ^ Y;</code>
not		<code>always_comb Z = ~ X;</code>
mux		<code>always_comb Z = S ? Y : X;</code>

Remember that these gates operate bitwise, so if X and Y are buses with N bits, N one-bit gates will be created. So in the following example, outputs Y and Z are equivalent.

```
module four_one_bit_gates(A,B,Y,Z); //list all ports here
    //define all ports
    input wire [3:0] A, B;
    output logic [3:0] Y, Z;

    //shorthand method
    always_comb Y = A & B;

    //longhand (you never need to do this)
    always_comb begin
        Z[0] = A[0] & B[0];
        Z[1] = A[1] & B[1];
        Z[2] = A[2] & B[2];
        Z[3] = A[3] & B[3];
    end
endmodule
```

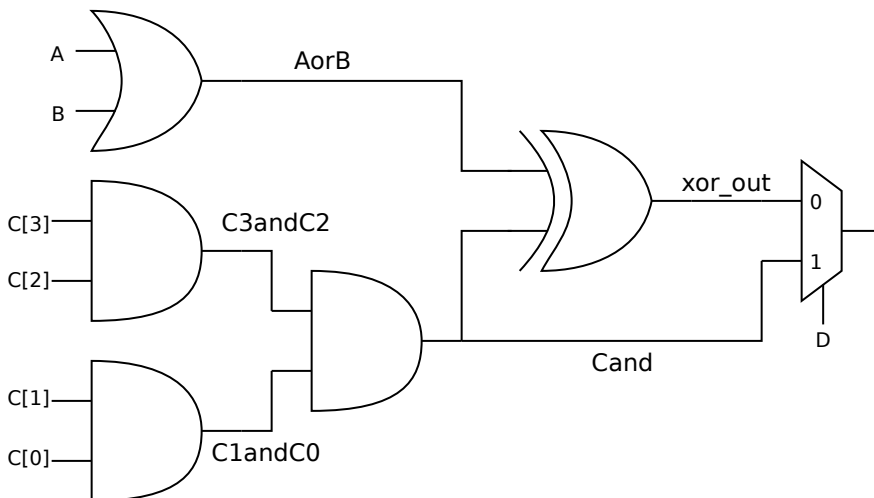
If instead you want to create one N -bit gate, you can use the bitwise operators as prefixes to a bus, like follows. Again, outputs Y and Z are equivalent:

```
module one_four_bit_gate(A,Y,Z);
    input wire [3:0] A;
    output wire Y,Z;

    //shorthand method
    always_comb Y = |A;

    //longhand (you never need to do this)
    always_comb Z = A[0] | A[1] | A[2] | A[3];
endmodule
```

Here's one last example summarizing how to create gates and hook them up in Verilog. The following circuit and Verilog module are identical.



```

module gates_example(A,B,C,D,Z1,Z2);           //declare a new module

//declare ports
input wire A, B;
input wire [3:0] C;
input wire D;
output logic Z1, Z2;

//create a logic for every gate output
logic AorB, C3andC2, C1andC0, Cand, xor_out;

//or gate
always_comb AorB = A | B; //creates an or gate with A and B as inputs, and
    AorB as the output
//and gates
always_comb C3andC2 = C[3] & C[2];
always_comb C1andC0 = C[1] & C[0];
always_comb Cand = C3andC2 & C1andC0;

//xor gate
always_comb xor_out = AorB ^ Cand;

//mux
always_comb Z1 = D ? Cand : xor_out;

//and once you get the hang of it, you can do all of that in a single line:
always_comb Z2 = D ? &C : (A|B)^(&C);
//just be careful about order of operations (use parentheses!)

endmodule //end the module - note that there is no semicolon here

```

6 Behavioural Combinational Logic

Once you get the hang of it, you can start using some behavioural statements (if, else, switch) in `alwayscomb` blocks. Here's an example that you can use to implement arbitrary truth tables!

```

// Make an XOR gate, the truth table way.
wire [1:0] xor_in;
logic xor_out;
always_comb begin : truth_table_for_xor;
    case(xor_in)
        2'b00 : out = 0;
        2'b01 : out = 1;
        2'b10 : out = 1;
        2'b11 : out = 0;
    endcase
end

// Make a 3:8 Decoder
wire [2:0] decoder_in;
logic [7:0] decoder_out;
always_comb begin : mux4
    case(switch)

```

```

3'd0 : decoder_out = 8'b00000001;
3'd1 : decoder_out = 8'b00000010;
3'd2 : decoder_out = 8'b00000100;
3'd3 : decoder_out = 8'b00001000;
3'd4 : decoder_out = 8'b00010000;
3'd5 : decoder_out = 8'b00100000;
3'd6 : decoder_out = 8'b01000000;
3'd7 : decoder_out = 8'b10000000;
endcase
end

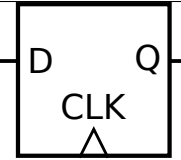
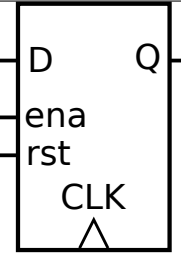
```

7 State Elements

This guide will focus on synthesizable circuits, which on FPGAs defaults to synchronous flip flops.

7.1 Synchronous Flip Flops

You can create flip-flop elements in Verilog with the `always_ff` block:

Schematic	Verilog
	<pre>always_ff @(posedge clk) q <= d;</pre>
	<pre>always_ff @(posedge clk) begin if(rst) q <= 0; else if (ena) q <= d; end</pre>

The `<=` operator isn't a "less than or equal to", it should be read as "becomes", as in "*q* will *become d* at the next positive edge of *clk*".

Both 'ena' and 'rst' are synchronous in the above examples. This guide avoids async inputs because (a) they are way harder to debug and (b) many FPGAs have scarcer asynchronous primitives and (c) modern FPGA clock speeds are high enough that I've never really needed an async input.

You can store multiple bits at a time by making *d* and *q* busses to create a *register* i.e.:

```

parameter N = 8;
wire [N-1:0] d;
logic [N-1:0] q;

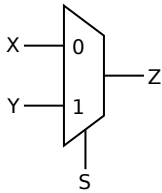
```

```
always_ff @(posedge clk) q <= d;
```

8 Modules

The basic building block in Verilog is the module - a bundle of hardware that you can instantiate multiple times. As an example, let's make a 2:1 mux module, then use it to build a 4:1 mux.

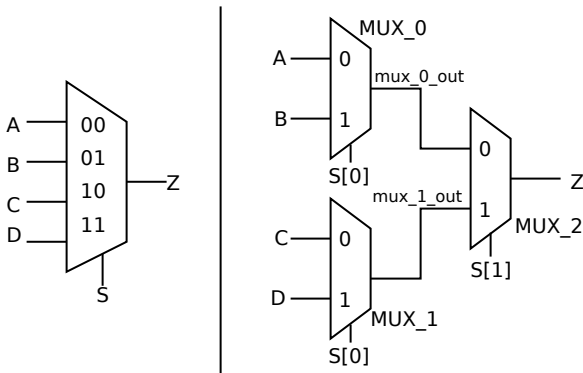
Note: a lot of examples put the port definitions inside the parenthesis on the module line instead of re-declaring them later. I recommend the following style because it makes it much easier to conditionally define things based on parameters and generate statements once you start doing more complicated modules.



```
module mux2(X,Y,S,Z);      //declare a new module (mux2), and list ports
    // Port definitions:
    input wire X, Y;
    input wire S;
    output logic Z;

    //this is the module body - where all of the module specific hardware gets
    //implemented.
    always_comb Z = S ? Y : X;

endmodule //end the module - note that there is no semicolon here
```



```
module mux_4(A,B,C,D,S,Z);      //declare a new module named mux_4to1
    //set type/size of ports
    input wire A, B, C, D;
    input wire S;
    output wire Z; // Using wires, not logics because we aren't creating new
    gates in this module (just ones that are defined in other modules).
```



```

//instantiate the module's hardware
wire [31:0] mux_0_out, mux_1_out;
mux2 #(.N(32)) MUX_0 (.X(A), .Y(B), .S(S[1]), .Z(mux_0_out));
mux2 #(.N(32)) MUX_1 (.X(C), .Y(B), .S(S[1]), .Z(mux_1_out));
mux2 #(.N(32)) MUX_2 (.X(mux_0_out), .Y(mux_1_out), .S(S[0]), .Z(Z));

endmodule //end the module - note that there is no semicolon here

```

8.1 Parameters

Our goal with using an HDL is to be able to reuse modules in many different situations. The muxes developed above work well, but they are made to reroute 32 bit signals. What if we wanted to instead route 16 or 64 bits? Verilog has a construct called a parameter that allows you to replace any *constant* in a module. The following is an example of making a parameterized 2:1 mux, and then using it to build a parameterized 4:1 mux.

```

module mux2(X,Y,S,Z);
    // parameter definitions
    parameter N = 8; // A parameter is an instantiation-time constant. Default
        value is 8, but that can be overridden at instantiation-time.

    // port definitions
    input wire [N-1:0] X, Y; // The parameter can be used later on in the code,
        // and will automatically create buses of the
        // right
        // size.
    input wire S;
    output logic [(N-1):0] Z; // One N-bit output bus.

    //this is the module body - where all of the module specific hardware gets
        implemented.
    always_comb Z = S ? Y : X;

endmodule // end the module - note that there is no semicolon here!

module mux_4to1(A,B,C,D,S,Z); // declare a new module named mux_4to1.
    //parameter definitions
    parameter N = 8; // you can name your parameter anything, but caps makes it
        easy to see it is a constant.

    //set type/size of ports
    input wire [(N-1):0] A, B, C, D; // four N-bit input buses
    input wire [1:0] S; // the switch input bus
    output wire [(N-1):0] Z; // one N-bit output bus

    //instantiate the module's hardware
    wire mux_0_out, mux_1_out;
    //we need to make sure that all of our smaller muxes have the same bus-width
    //the '#' directive is used to set the parameters of a module's
        instantiation
    //it comes before the name of the instance, and works like wiring ports does
    mux2 #(.N(N)) MUX_0 (.X(A), .Y(B), .S(S[1]), .Z(mux_0_out));
    mux2 #(.N(N)) MUX_1 (.X(C), .Y(B), .S(S[1]), .Z(mux_1_out));

```

```

mux2 #(.N(N)) MUX_2 (.X(mux_0_out), .Y(mux_1_out), .S(S[0]), .Z(Z));
//now all of the mux2 instances have bus widths that are set by the mux_4to1
's parameter

endmodule //end the module - note that there is no semicolon here

```

9 Behavioural Verilog

Testing a hardware description using hardware descriptions is not an easy task. By using *behavioural* Verilog, we can start to use some more traditional software constructs and algorithm to test our hardware in simulation. Be careful - you can only use the following behavioural verilog techniques inside of `initial` or `always` blocks. You will not receive credit for designs that use behavioural verilog!

9.1 Behavioural Datatypes

The basic datatype in combinational structural verilog is a wire. Wires do *not* hold state, so they are useless when it comes to writing procedural code. There is another datatype available that works just like a wire, except that it holds state. This is the `logic`. Inside of any behavioural (`initial` or `always`) block, you can use a `logic` bus just as you would use a variable in a more procedural language like C. In addition, you can declare a bus as `signed` (buses are unsigned by default). That means that when performing behavioural mathematical operations on the bus, or when printing the bus, it will be treated as a standard two's complement number.

9.2 More operators

Behavioural Verilog offers all of the standard integer operations in C. You can add `+`, subtract `-`, multiply `*`, divide `/` or even bitshift numbers (`>>` or `<<`). Note that these are all *integer* operations - division will truncate and multiplication can easily overflow. Also, if you declare your regs as signed, the operators will treat them as such.

You can also use the relational operators from C (`>`, `<`, `>=`, and `<=`). These will always return one or zero, regardless of how large the bus in the `alwayscomb` statement.

When it comes to equality operators, it is important to realize that there are actually 4 states to any bit in Verilog. It can be 0, 1, z, or x. Z represents that a wire is not connected to anything, and x represents a wire that is based on an unknown state. So if you see x's or z's in your waveforms, it is extremely likely that you misconnected a module. To deal with these extra states, Verilog has both the traditional equals and not equals operators (`==` and `!=`). These will never evaluate to true if either of their operands is in an unknown state. **SIMULATION ONLY:** If you'd like the checks to include x and z, you need to use `===` and `x==`. Again, these operators only return one or zero.

Be very careful about order of operations, use parentheses if you have any doubt. Avoid the `!` operator unless you know what you are doing.

9.3 Loops

Behavioural verilog allows for your standard `for` and `while` loops⁴. Just be sure to declare your loop index outside of the behavioural block (at the module level).

9.4 Branching

You can use `if`, `else`, `else if`, and `case` in behavioural Verilog. The following is a simple example to show you how it works:

```
//if - else if - else branching
if(a === 0) begin
    a = a - 1;
end
else if (a === 1) begin
    a = a + 1;
end
else begin
    a = a + 2;
end

//case select structure
case (signal) //start switching based on signal
    8'd0 : begin
        //this code executes when signal is 0
    end
    8'd1 : begin
        //this code executes when signal is 1
    end
    default : begin
        //any states of signal that are not declared end up here
        //always include a default case to avoid fallthrough!
    end
endcase
```

9.5 System Calls

Verilog allows for various system calls that can be very useful in writing a testbench. Many of these only function within an `initial` or `always` block.

`$clog2` is extremely useful - you can use it to compute the ceiling (round up) of the \log_2 of any number (aka the number of bits required to represent that many levels). This is extremely useful for parametrized definitions - e.g. for a display that is 320x240, you can have the following:

```
parameter WIDTH=320;
parameter HEIGHT=240;
parameter N_X = $clog2(WIDTH);
parameter N_Y = $clog2(HEIGHT);
```

⁴Since Verilog uses curly brackets for bus concatenation, they cannot be used to delimit code blocks. Instead, Verilog uses `begin` and `end`.

You can make a procedural block consume time by using the pound sign⁵ - #100 will wait for 100x the smallest unit of time. You can set the time unit with the `'timescale 1ns/1ps` command - the first number is the smallest unit of time you can work with, and the second is the simulation precision that will be used.

`$time` will return the current simulation time. The units are based on the `'timescale 1ns/1ps` directive that should be at the top of simulation files.

You can get random numbers with the `$random` function. The first time you can call it you can set its seed to a particular value (`a_random_logic = $random(seed_value);`), or just call it without the paranthesis (`another_random_logic = $random;`).

There are three ways to print a string in Verilog. The first, `$display` is extremely similar to the `printf` command from C - it immediately prints its arguments. `$strobe` is just like `display`, except that it waits to print until the end of the current simulation time unit. Finally, there is the `$monitor` statement which, once called, will print whenever any of its arguments changes. Use it with caution as it can be extremely verbose.

You can end the simulation completely with the `$finish` command. Using `$stop` will suspend the simulation - you can run from a stopped point in discrete time increments using `run 100ns` at the isim console.

9.6 Initial Blocks

An initial block is a special non-synthesizable Verilog block that executes all of the code inside it in sequence. This means that instead of creating hardware, it just runs through each line just as you would expect the code to work in a procedural language like C.

9.7 Always Blocks (simulation only)

An `always` block is a pretty powerful tool - it executes once whenever any of its inputs changes. For example, if you are testing a block that has two inputs X and Y, you would start the `always` block like so: `always @(X or Y)`. Whenever X or Y change at any point in the simulation, the code in this block will be executed. We mostly use these for checkers (see below).

Just remember that you should never use a bare `always` for synthesizable logic! Use `always_comb`, `always_ff`, or if desperate `always_latch` instead.

9.8 Behavioural Example

Its easier to see all of these behavioural tools in action - here's a simple module that shows off most of these features of the language:

```
'timescale 1ns/1ps
module system_calls;
    reg [31:0] random_seed, random_number, a, b, c, i;
    reg unset; //a reg that we will never set

    initial begin
        //this line will print whenever c changes ($time is not monitored)
        $monitor("@%8t : c has changed to value %d", $time, c);
        random_seed = 32'd3; //this seed will ensure that everytime the sim is run
                               // we will get the same "random" output
    end
endmodule
```

⁵Or hashtag, whatever, don't @ me.

```

random_number = $random(random_seed);
//
                                the                first
                                //random call

a = 0;
b = 0;
c = 0;
#10;
$display("@%8t : display : b = %d", $time, b); //this will print now
$strobe("@%8t : strobe : b = %d", $time, b);
b = 3;
b = 5;
b = -33; //this is the only value the strobe will print
c = 1;
#10;

while(a < 10) begin
    for(i = 0; i < 10; i = i + 1) begin
        a = a + 2*i;
        $display("@%8t : a = %h in hex and i = %b in binary", $time, a, i);
        #20;
    end
    c = c + 1;
end
$finish; //end the simulation
end

always @(a or b) begin
    $display("@%8t : a or b changed and the always block noticed", $time);
end

initial begin //this initial block will run in parallel to the first block,
    not after it!
    $display("@%8t : This is the first statement from a parallel initial block
        .", $time);
    #15;
    $display("@%8t : This is the second statement from a parallel initial
        block.", $time);
    #1000;
    $display("This statement will never print because the other initial block
        will call $finish first.");
    $finish;
end
endmodule

```

Yields this output:

```

@      0 : This is the first statement from a parallel initial block.
@      0 : a or b changed and the always block noticed
@      0 : c has changed to value          0
@ 10000 : display : b =          0
@ 10000 : a or b changed and the always block noticed
@ 10000 : strobe : b = 4294967263
@ 10000 : c has changed to value      13      1

```

```

@ 15000 : This is the second statement from a parallel initial block.
@ 20000 : a = 00000000 in hex and i = 00000000000000000000000000000000 in binary
@ 40000 : a = 00000002 in hex and i = 00000000000000000000000000000001 in binary
@ 40000 : a or b changed and the always block noticed
@ 60000 : a = 00000006 in hex and i = 00000000000000000000000000000010 in binary
@ 60000 : a or b changed and the always block noticed
@ 80000 : a = 0000000c in hex and i = 00000000000000000000000000000011 in binary
@ 80000 : a or b changed and the always block noticed
@ 100000 : a = 00000014 in hex and i = 00000000000000000000000000000100 in binary
@ 100000 : a or b changed and the always block noticed
@ 120000 : a = 0000001e in hex and i = 00000000000000000000000000000101 in binary
@ 120000 : a or b changed and the always block noticed
@ 140000 : a = 0000002a in hex and i = 00000000000000000000000000000110 in binary
@ 140000 : a or b changed and the always block noticed
@ 160000 : a = 00000038 in hex and i = 00000000000000000000000000000111 in binary
@ 160000 : a or b changed and the always block noticed
@ 180000 : a = 00000048 in hex and i = 000000000000000000000000000001000 in binary
@ 180000 : a or b changed and the always block noticed
@ 200000 : a = 0000005a in hex and i = 000000000000000000000000000001001 in binary
@ 200000 : a or b changed and the always block noticed
@ 220000 : c has changed to value          2

```

10 Testbenches

A testbench has to do two things - supply a set of inputs into the unit under test (UUT)⁶ and ensure that the outputs match the specifications for the block. You can set the set of test inputs (also known as the test vectors) with an initial block. Always blocks are good for checking the output whenever the inputs change since they can execute in parallel and be used with different test vectors.

10.1 Test Vectors

The simplest test vectors consist of just setting values and putting in delays like so:

```

reg a, b, c; //inputs
initial begin
    a = 0; b = 0; c = 0;
    #10; a = 0; b = 0; c = 1;
    #10; a = 0; b = 1; c = 0;
    #10; a = 0; b = 1; c = 1;
    #10; a = 1; b = 0; c = 0;
    $finish;
end

```

⁶Often referred to as Device Under Test or Design Under Test (DUT)

You can also load in longer test vectors with the `$readmemh` and `$readmemb` tasks (note this is also how we initialize memories). For example to test a UUT with an 8 bit input, you could first create a text file `test_vector.memb`:

```
1010_0101
0000_0000
1111_0000
0000_1111
1111_1111
```

And then use it in a testbench like so:

```
logic [7:0] test_vector[0:4];

initial begin
    // other initialization code

    // load this file into our test_vector memory
    $readmemb("test_vector.memb", test_vectors);
    for (int i = 0; i < 5; i = i + 1) begin
        test_input = test_vector[i][3:0];
        #1;
        if (test_output != test_vector[i][7:4]) begin
            $display("Error, in = %b, out = %b, should be %b", test_input,
                test_output, test_vector[i][7:4]);
        end
    end
end
```

`$readmemh` is the same, except you can write out your constants in hexadecimal.

You can also use loops, if statements, etc. to make setting up appropriate test vectors easier. Just be sure to delay when you want a value to actually be computed.

10.2 Checkers

A checker should check the outputs of a UUT *after* the inputs have been changed to make sure that they are proper. By using any of the behavioural operators and if statements, you can easily catch error conditions. For example, if a block is supposed to compute the product of two inputs X and Y and store the result in Z, you could write the following checker:

```
logic X, Y;
wire Z;

multiplier UUT (.X(X), .Y(Y), .Z(Z));

//checker
always @(X or Y) begin //make this fire whenever X or Y change
    #1; //wait the smallest step to make sure we are checking the output after
        the change
    if( Z != (X*Y) ) beginb
        $display("@%8t : Error: Z (%d) is not equal to X(%d) * Y(%d)", $time, Z, X
            , Y);
        $stop; //you can stop the simulation here to make debugging easy
    end
end
```

```
end
```

Combine checkers and test vectors to create rich testbenches!

11 Advanced Techniques

11.1 Linting

Linting is a way to search code for errors, ideally before running time consuming compilation or synthesis. Most Verilog tools are pretty bad at this, but one open source project **verilator** does an amazing job of providing sane and legible error messages. I recommend including automatic linting into your text editor, but you can always run **verilator --lint-only -Wall foo.sv** to help with your debugging. If you have modules referenced in other directories, remember to add the include options - most of my examples have hdl and tests folders, so the following works well for simulation:

```
verilator --lint-only -Wall -I./hdl -I./tests foo.sv
```

11.2 Functions and Tasks

You can use **functions** and **tasks** to abstract behavior. The only rule is that functions cannot consume time (and therefore can be used in synthesizable modules). Tasks can consume time, and are useful for testbenches and CANNOT be synthesized. Here's an example that's great for finding the index of items arranged in a square $N \times N$ grid:

```
parameter N;  
function int get_index(int i, int j);  
    cell_index = N*j + i;  
endfunction
```

Tasks are a good way to model things changing over time to make better testbenches. I use this test to model “bouncy” inputs, like buttons and switches:

```
int bounces, delay;  
task bounce_button();  
    bounces = ($urandom % 20) + 10;  
    $display("Starting a %d bounce sequence.", bounces);  
    for(int i = 0; i < bounces; i = i + 1) begin  
        delay = ($urandom % 15) + 1;  
        $display("bouncing with delay %d", delay);  
        #(delay) buttons[1] = $urandom;  
    end  
endtask
```

11.3 Generate Statements


```

module ripple_carry_adder(a, b, c_in, sum, c_out);

parameter N = 8;

input wire [N-1:0] a, b;
input wire c_in;
output logic [N-1:0] sum;
output wire c_out;

wire [N:0] carries;
assign carries[0] = c_in;
assign c_out = carries[N];
generate
    genvar i;
    for(i = 0; i < N; i++) begin : ripple_carry
        full_adder ADDER (
            .a(a[i]),
            .b(b[i]),
            .c_in(carries[i]),
            .sum(sum[i]),
            .c_out(carries[i+1])
        );
    end
endgenerate

endmodule
// to instantiate
// ripple_carry_adder #(.N(32)) ADDER_RCA_32 ( /* port list */ );

```

12 Sources and Resources

- I learned a lot of this through Sutherland's books, the latest (and sadly last) book is **RTL Modeling with SystemVerilog for Simulation and Synthesis: Using SystemVerilog for ASIC and FPGA Design** by Stuart Sutherland.
- Find the latest version of UG901 Vivado Synthesis Guide - it shows you the correct VHDL, Verilog, or SystemVerilog that infers specific circuitry in Xilinx FPGAs. It's a long document, but well worth having around as a reference.