

SystemVerilog Cheat Sheet

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Caveat emptor² - This guide started in 2011 (so might always be a bit outdated), prioritizes both open-source and proprietary implementations (so it lags on some features), and last but not most important, it's a cheatsheet!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 “behavioral” is simulateable, and *some* but not all behavioral 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.
 - 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 `always_comb` 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.

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).

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 Constants, Enums, and Custom Types

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*. Macros are defined and used with the back-tick (`) - typically found near your Esc key.

```
// Define a constant. Note that there is no semicolon!
`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, but it's not always well supported by the open source icarus verilog.

```
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!

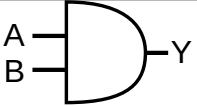
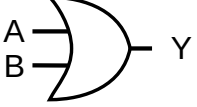


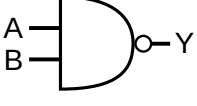


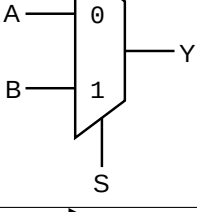
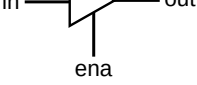
```
typedef enum logic [1:0] {IDLE, BUSY, DONE, ERROR} state_t;
state_t state; // is a logic [1:0].
always_ff @(posedge clk) begin
    case(state)
        IDLE: state <= BUSY;
        BUSY: state <= DONE;
        DONE: state <= IDLE;
        default: state <= ERROR;
    endcase
end
```

In the above example the `typedef` statement creates a new user-defined type that you can use elsewhere. Support for typedefs isn't completely present in all tools. *The ? mux operator in particular does **not** like typedef'd values!* If you are trying to mux something typedef'd you will have to use the more behavioral if/else or case methods of making a mux.

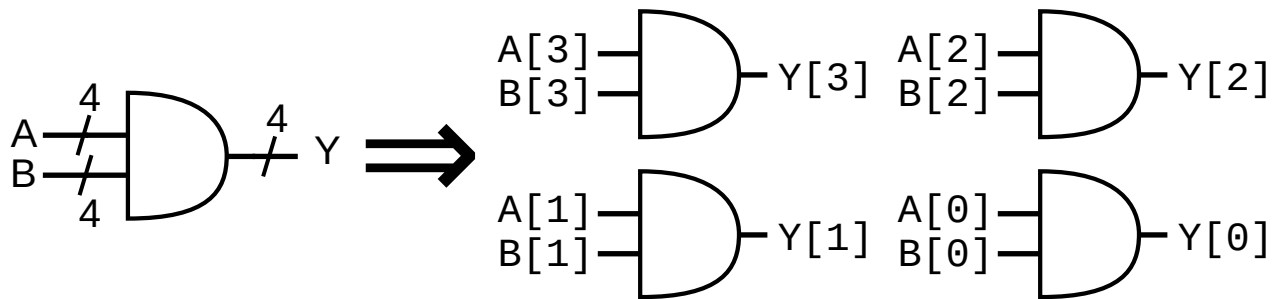
5 Combinational Logic: Structural Gates

You can create

The following is a list of gates and their structural verilog equivalents using the `always_comb` statement:

Gate	Schematic	Structural Verilog
and		<code>always_comb Y = A & B;</code>
or		<code>always_comb Y = A B;</code>
xor		<code>always_comb Y = A ^ B;</code>
not		<code>always_comb Y = ~A;</code>
nand		<code>always_comb Y = A ~& B;</code>
nor		<code>always_comb Y = A ~ B;</code>
nxor ⁴		<code>always_comb Y = A ^^ B;</code>
mux		<code>always_comb Y = S ? B : A;</code>
tristate ⁵		<code>assign out = ena ? in : 1'bz;</code>

Remember that these gates operate bitwise, so if A and B are buses with N bits, N one-bit gates will be created. So in the following example, outputs $Y1$ and $Y2$ are equivalent.



```

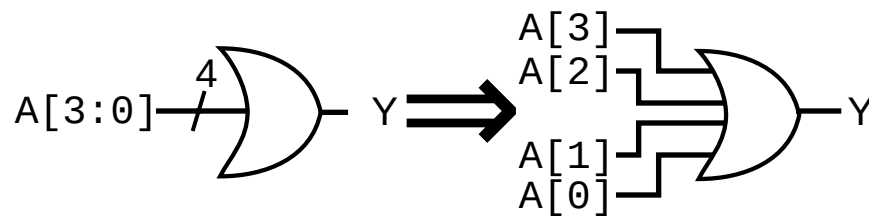
module four_one_bit_gates(A,B,Y,Z); //list all ports here
  //define all ports
  input wire [3:0] A, B;
  output logic [3:0] Y1, Y2;

  //shorthand method
  always_comb Y1 = A & B;

  //longhand (you never need to do this)
  always_comb begin
    Y2[0] = A[0] & B[0];
    Y2[1] = A[1] & B[1];
    Y2[2] = A[2] & B[2];
    Y2[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 $Y1$ and $Y2$ are equivalent:



```

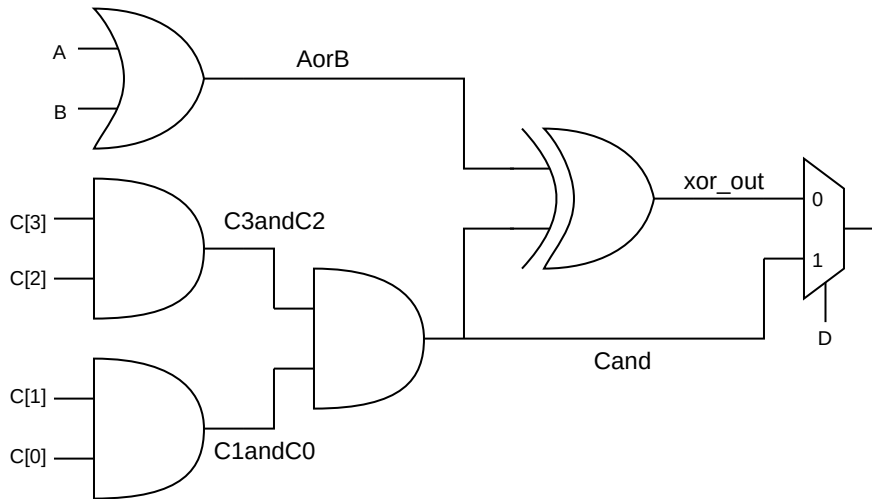
module one_four_bit_gate(A,Y,Z);
  input wire [3:0] A;
  output logic Y1, Y2;

  //shorthand method
  always_comb Y1 = |A;

  //longhand (you never need to do this)
  always_comb Y2 = 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 net of type logic for every gate output.
    logic AorB, C3andC2, C1andC0, Cand, xor_out;

    // You can combine multiple comb. logic statements in one block with begin/end.
    always_comb begin
        // Create an OR gate with A and B as inputs that outputs to AorB.
        AorB = A | B;
        // Create some AND gates.
        C3andC2 = C[3] & C[2];
        C1andC0 = C[1] & C[0];
        Cand = C3andC2 & C1andC0;

        // An XOR gate.
        xor_out = AorB ^ Cand;

        // A 2:1 mux.
        Z1 = D ? Cand : xor_out;
    end

    // Once you get the hang of it, you can combine operations in a single line.
    // Just be careful about order of operations (use parentheses!) and always
    // prioritize readability.
    always_comb Z2 = D ? &C : (A|B)^(&C);

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

```

6 Combinational Logic: Behavioral

Once you prove that you understand which gates are used to make building blocks, you can start using some behavioral statements (if, else, switch) in `always_comb` 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;

// It's best practice to label a begin with something that describes what the
// block is doing.
// Some tools will use this label on the generated synthesis outputs which can
// greatly aid in debugging.
always_comb begin : truth_table_for_xor;
    case(xor_in)
        2'b00 : xor_out = 0;
        2'b01 : xor_out = 1;
        2'b10 : xor_out = 1;
        2'b11 : xor_out = 0;
    endcase
end

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

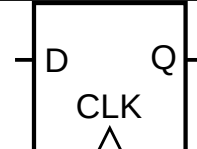
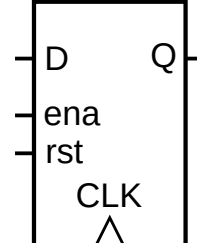
Please don't write behavioral combinational logic until you are comfortable with structural!
You lose a lot of control over how the tools will realized your description into circuits with this method.

7 Sequential Logic: State Elements

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

7.1 Synchronous Flip Flops

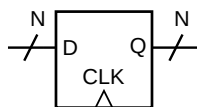
You can create flip-flop elements in Verilog with the `always_ff` block:

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, (b) many FPGAs have scarcer asynchronous primitives, and (c) modern FPGA clock speeds are high enough that the latency cost of asynchronous inputs is relatively low.

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

Schematic	Verilog
	<pre>parameter N = 8; wire [N-1:0] d; logic [N-1:0] q; always_ff @(posedge clk) q <= d;</pre>

7.2 Latches

There is an `always_latch` statement you can use to create asynchronous latches. Your mileage may vary, it's the least used always type and as such doesn't always work well with the tools. You will probably never need it, but for completeness here is how to define an SR latch.

```
module sr_latch(s,r,q);  
  input wire s, r;  
  output logic q;  
  
  always_latch begin  
    if(s) q <= 1'b1;  
    if(r) q <= 1'b0;  
  end  
  
endmodule
```

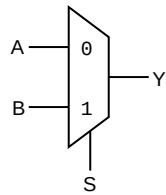
Again, this should work just fine in simulation, but is unlikely to be used for this course.

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, parametrize it, then use it to build a 4:1 mux.

You should only have one module per file! Try to keep the module name and the file name the same, e.g. `module foo` should exist within a file `foo.sv`.

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(in0, in1, s, out); // Declare a new module (mux2), and list port names
    .

    // Define the direction and type of ports.
    input wire in0, in1;
    input wire s;
    output logic out; // If you will drive a net from an always block it should be a
        logic type.

    // This is the module body - where all of the module specific hardware gets
        implemented.
    always_comb out = s ? in1 : in0;

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 mux above works for 1-bit signals, but 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 N-bit 2:1 mux.

```
module mux2(in0, in1, s, out);
    // Parameter definitions.
    N = 8; // A parameter is an instantiation-time constant. Default value is 8,
           // but that can be overridden when instantiated.

    // Port definitions - note that these can use parameters for widths.
    input wire [N-1:0] in0, in1; // Two N-bit input busses.
    input wire s;
    output logic [N-1:0] out; // Two N-bit input busses.

    // Module body. Verilog operators work per-bit, so the same statement works for
    // 1-bit or N-bit busses.
    always_comb out = s ? in1 : in0;
endmodule // end the module - note that there is no semicolon here!
```

8.2 Instantiating Modules

Modules can be instantiated from other modules. The syntax is:

```
module_name #(*parameter overrides */)INSTANCE_NAME (*port connections */).
```

These instantiations should stand alone within the body of a module (not in an always block!).

Parameter overrides let you change the value of a parameter per created instance. Port connections are how you connect the different ports of a submodule to the nets in the higher level module. While you can just put comma separated terms in these parenthesis, it is much better to use dot notation to explicitly name which parameter you are overriding and which port connects to which net. This can avoid very difficult to find miswiring bugs.

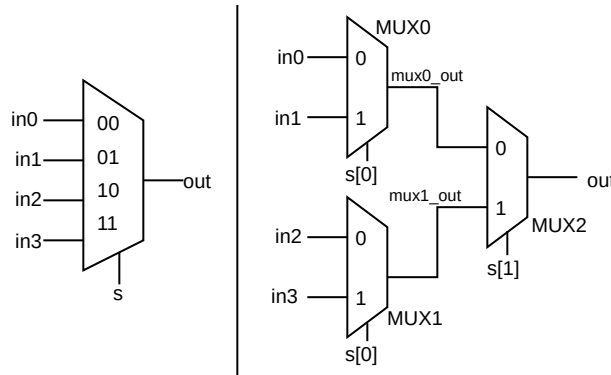
Dot notation for parameters:

```
.name_of_parameter(parameter_value).
```

Dot notation for ports:

```
.name_of_port_in_the_submodule(name_of_wire_in_the_higher_level_module).
```

The following is an example of composing a 4:1 mux out of 2:1 muxes (in a binary tree).



```

module mux4(in0, in1, in2, in3, s, out); // Declare module name, list of ports.

// Parameter definitions. Note that the default here is different.
parameter N=32;

// Port definitions.
input wire [N-1:0] in0, in1, in2, in3;
input wire [1:0] s;
output logic [N-1:0] out;

// Body

// Create wires to connect between submodule ports.
wire [N-1:0] mux0_out, mux1_out;

// Instantiate modules using dot notation.
mux2 #(.N(N)) MUX0 (.in0(in0), .in1(in1), .s(s[0]), .out(mux0_out));
mux2 #(.N(N)) MUX1 (.in0(in2), .in1(in3), .s(s[0]), .out(mux1_out));

mux2 #(.N(N)) MUX2 (.in0(mux0_out), .in1(mux1_out), .s(s[1]), .out(out));

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

```

8.3 Advanced parametrization: localparam and defparam

It is common to have constants in a module that are set by some combination of instantiation-dependent parameters. A good example is setting the width of a counter based on a maximum value (or the width of an address field based on the length of the memory).

```

module up_counter(rst, clk, count)
  parameter MAX_COUNT;
  localparam COUNTER_WIDTH=clog2(MAX_COUNT);
  input wire rst, clk;
  output logic [COUNTER_WIDTH-1:0] count;

  // ... logic omitted for clarity ...
endmodule // up_counter

```

The end user of the above module can then instantiate it based purely on a design-level parameter MAX_COUNT and not worry about how many bits it takes to implement.

8.3.1 Testing and Parameters

Once you start adding parameters to your modules the amount of test cases can dramatically increase (by 2^N , where N is the number of bits in your parameters). Be sure to run tests on modules with the final parameters used wherever possible! Handling this in a less direct/brute-force manner is beyond the scope of this document.

9 Behavioral Verilog

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

9.1 Behavioral 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 behavioral (`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 behavioral 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

Behavioral 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 `always_comb` 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

Behavioral verilog allows for your standard `for` and `while` loops⁶. Just be sure to declare your loop index outside of the behavioral block (at the module level).

9.4 Branching

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

⁶Since Verilog uses curly brackets for bus concatenation, they cannot be used to delimit code blocks. Instead, Verilog uses `begin` and `end`.

```

//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 parameterized 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);

```

You can make a procedural block consume time by using the pound sign⁷ - `#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 parenthesis (`another_random_logic = $random;`).

⁷Or hashtag, whatever, don't @ me.

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 logic⁸! Use `always_comb`, `always_ff`, or `always_latch`⁹ instead.

⁸As of 2023 you should avoid tools that don't support `always_comb` or `always_ff`.

⁹Only use `always_latch` if it is absolutely required. Many FPGA synthesis tools don't support them well

9.8 Behavioral Example

Its easier to see all of these behavioral 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 every time the sim is run
                               // we will get the same "random" output
        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
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          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 = 00000000000000000000000000001000 in binary
@ 180000 : a or b changed and the always block noticed
@ 200000 : a = 0000005a in hex and i = 00000000000000000000000000001001 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)¹⁰ 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
```

You can also load in longer test vectors with the \$readmemb 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.mem:

```
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.mem", 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.

¹⁰Often referred to as Device Under Test or Design Under Test (DUT)

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 behavioral 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) ) begin
        $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
```

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 with whichever directories have sv files that your modules might reference.

```
verilator --timing --lint-only -Wall -I./path/to/hdl -I/another/path/to/hdl 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.