Verilog Cheat Sheet

updated 2015-09-09

Avinash Uttamchandani
Preceptor in EE
Harvard SEAS

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.

2 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 (input, inout, output, wire), the bus size ([most significant bit: least significant bit]), and finally the name (or names if you have multiple wires of the same size). 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;       //an 8 bit output
```

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;

assign a = b[0] ^ b[1];      //accessing bits of bus b individually
assign d = c[1:0] & c[3:2]; //and-ing the lowest two bits of c with the highest
    two bits
assign 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
```

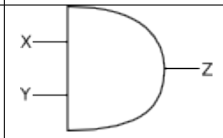
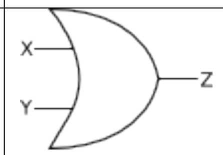
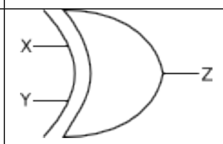
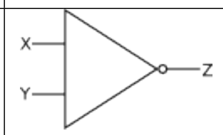
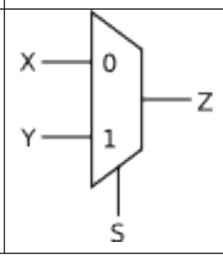
3 Constants

Since every 'number' in Verilog is actually a collection of bits, it is really important to assign constants correctly so that you don't try to fit a number into less bits than it takes to express it. 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:

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

//a, b, and c are all assigned to decimal 224
assign a = 8'b1110_0000;   //a = 14 in decimal
                           //use underscores to make numbers more reasonable
assign b = 8'hE0;
assign c = 8'd224;

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

4 Structural Gates

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

Gate	Schematic	Structural Verilog
and		**assign** Z = X & Y;
or		**assign** Z = X | Y;
xor		**assign** Z = X ˆ Y;
not		**assign** Z = ∼ X;
mux		**assign** Z = S ? Y : X;

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.

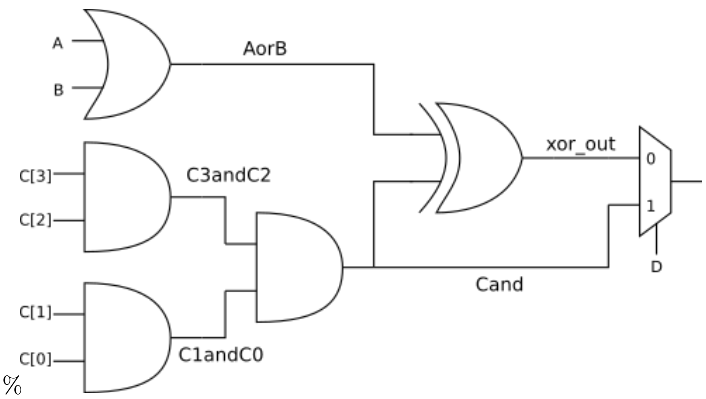
```
module four_one_bit_gates(A,B,Y,Z); //list all ports here
   //define all ports
   input  wire [3:0] A, B;
   output wire [3:0] Y, Z;

   //shorthand method
   assign Y = A & B;

   //longhand (you never need to do this)
   assign Z[0] = A[0] & B[0];
   assign Z[1] = A[1] & B[1];
   assign Z[2] = A[2] & B[2];
   assign Z[3] = A[3] & B[3];

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
    assign Y = |A;

    //longhand (you never need to do this)
    assign 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.

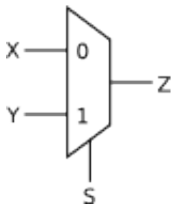



```
%

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 wire Z1, Z2;

    //create a wire for every node on the schematic that is not a port
    wire AorB, C3andC2, C1andC0, Cand, xor_out;

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

    //xor gate
    assign xor_out = AorB ^ Cand;

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

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

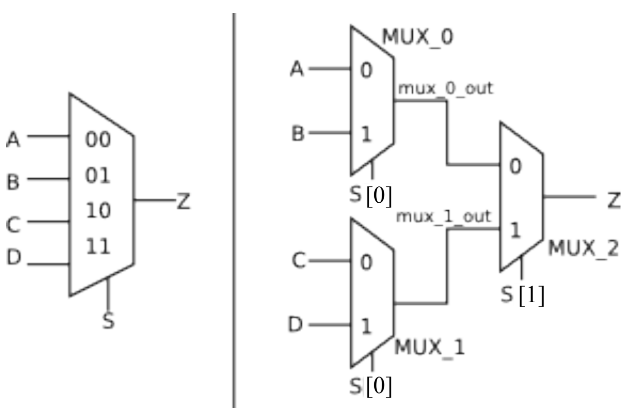
```
endmodule //end the module - note that there is no semicolon here
```

5 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 32 bit mux module (Fig. 1), then use it to build a 4:1 32 bit mux (Fig. 2):




```
module mux_2to1(X,Y,S,Z);      //declare a new module (mux_2to1), and list ports
   //port definitions
   input  wire [31:0] X, Y;   //two 32 bit input buses
   input  wire        S;      //if you leave out the bus dimensions, it defaults
        to one bit
   output wire [31:0] Z;      //one 32 bit output bus

   //this is the module body - where all of the module specific hardware gets
       implemented.
   assign 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
   //set type/size of ports
   input  wire [31:0] A, B, C, D;   //four 32 bit input buses
   input  wire [1:0]  S;            //the switch input bus
   output wire [31:0] Z;            //one 32 bit output bus

   //instantiate the module's hardware
   wire mux_0_out, mux_1_out;
   mux_2to1 MUX_0 (.X(A), .Y(B), .S(S[0]), .Z(mux_0_out));
   mux_2to1 MUX_1 (.X(C), .Y(B), .S(S[0]), .Z(mux_1_out));
```

```
    mux_2to1 MUX_2 (.X(mux_0_out), .Y(mux_1_out), .S(S[1]), .Z(Z));

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

5.1 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 mux_2to1(X,Y,S,Z);      //declare a new module (mux_2to1), and list all
    the ports
    //parameter definitions
    parameter N = 8;  //a parameter is an instantiation-time constant.  Default
        value is 8.

    //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 wire [(N-1):0] Z;     //one 32 bit output bus

    //this is the module body - where all of the module specific hardware gets
        implemented.
    assign 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 WIDTH = 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 [(WIDTH-1):0] A, B, C, D;  //four N-bit input buses
    input  wire [1:0]  S;               //the switch input bus
    output wire [(WIDTH-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
    mux_2to1 #(.N(WIDTH)) MUX_0 (.X(A), .Y(B), .S(S[1]), .Z(mux_0_out));
    mux_2to1 #(.N(WIDTH)) MUX_1 (.X(C), .Y(B), .S(S[1]), .Z(mux_1_out));
    mux_2to1 #(.N(WIDTH)) MUX_2 (.X(mux_0_out), .Y(mux_1_out), .S(S[0]), .Z(Z));
    //now all of the mux_2to1 instances have bus widths that are set by the
        mux_4to1's parameter
```

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

6 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 algorithsm 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 for the first two labs!

6.1 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 **reg**. Inside of any behavioural (**initial** or **always**) block, you can use a **reg** 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. Regs can only be driven from one block (to prevent multiple driver conflicts).

6.2 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 you assign this to is.

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. If you'd like the checks to include x and z, you need to use === and !==. Again, these operators only return zero or one.

Be very careful about order of operations! Use parentheses if you have any doubt.

6.3 Loops

Behavioural Verilog allows for your standard **for** and **while** loops[1]. Just be sure to declare your loop index outside of the behavioural block (at the module level).

6.4 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
```

6.5 System Calls

Verilog allows for various system calls that can be very useful in writing a testbench.

The current simulation time can be accessed in the special $time variable.

You can get random numbers in Verilog[2] with the $random function. The first time you can call it you can set its seed to a particular reg (a_random_reg = $random(seed_reg);), or just call it without the paranthesis (another_random_reg = $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

[1] Since Verilog uses curly brackets for bus concatenation, they cannot be used to delimit code blocks. Instead, Verilog uses **begin** and **end**.

[2] There is an extension of the Verilog language called SystemVerilog that was invented to make generating random test vectors easier. Be wary of using random numbers to drive a simulation - it will not guarantee a good test vector without careful thought and planning.

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.

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

6.6 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. Just make sure that you only use regs in this block - assigns will not work.

6.7 Always Blocks

An always block is a pretty powerful tool - it executes once whenver 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. Always blocks are special because they are used to create synthesizable sequential logic. We’ll learn more about that later.

6.8 Behavioural Example

Its easiest to just 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
      random_number = $random(random_seed); //you need to set the seed on 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          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
```

7 Testbenches

A testbench has to do two things - suppply a set of inputs into the unit under test (UUT)[3] 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.

7.1 Test Vectors

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

```verilog
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 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.

7.2 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:

[3] Often referred to as Device Under Test or Design Under Test (DUT)

```
reg 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!