271/469 Verilog Tutorial

Prof. Scott Hauck, last revised 7/9/20

Introduction

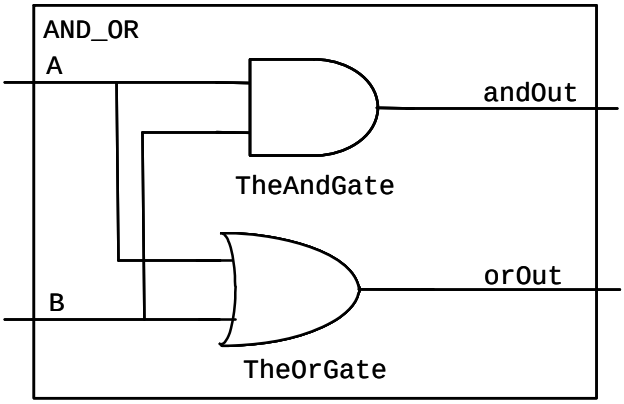
The following tutorial is intended to get you going quickly in circuit design in Verilog. It isn't a comprehensive guide to System Verilog, but should contain everything you need to design circuits for your class.

If you have questions, or want to learn more about the language, I'd recommend Vahid and Lysecky's *Verilog for Digital Design.*

Modules

The basic building block of Verilog is a module. This is similar to a function or procedure in C/C++/Java in that it performs a computation on the inputs to generate an output. However, a Verilog module really is a collection of logic gates, and each time you call a module you are creating that set of gates.

An example of a simple module:




```
// Compute the logical AND and OR of inputs A and B.
module AND_OR(andOut, orOut, A, B);
  output logic  andOut, orOut;
  input  logic  A, B;

  and TheAndGate (andOut, A, B);
  or TheOrGate (orOut, A, B);
endmodule
```

We can analyze this line by line:

```
// Compute the logical AND and OR of inputs A and B.
```

The first line is a comment, designated by the //. Everything on a line after a // is ignored. Comments can appear on separate lines, or at the end of lines of code.

```
module AND_OR(andOut, orOut, A, B);
```

```
  output logic  andOut, orOut;
  input  logic  A, B;
```

The top of a module gives the name of the module (AND_OR in this case), and the list of signals connected to that module. The subsequent lines indicate that the first two binary values (andOut and orOut) are generated by this module, and are output from it, while the next two (A, B) are inputs to the module.

```
  and TheAndGate (andOut, A, B);
  or TheOrGate (orOut, A, B);
```

This creates two gates: An AND gate, called "TheAndGate", with output andOut, and inputs A and B; An OR gate, called "TheOrGate", with output orOut, and inputs A and B. The format for creating or "instantiating" these gates is explained below.

```
endmodule
```

All modules must end with an endmodule statement.

Basic Gates

Simple modules can be built from several different types of gates:

```
buf <name> (OUT1, IN1); // Sets output equal to input
not <name> (OUT1, IN1); // Sets output to opposite of input
```

The <name> can be whatever you want, but start with a letter, and consist of letters, numbers, and the underscore "_". Avoid keywords from Verilog (i.e. "module", "output", etc.).

There are multi-input gates as well, which can each take two or more inputs:

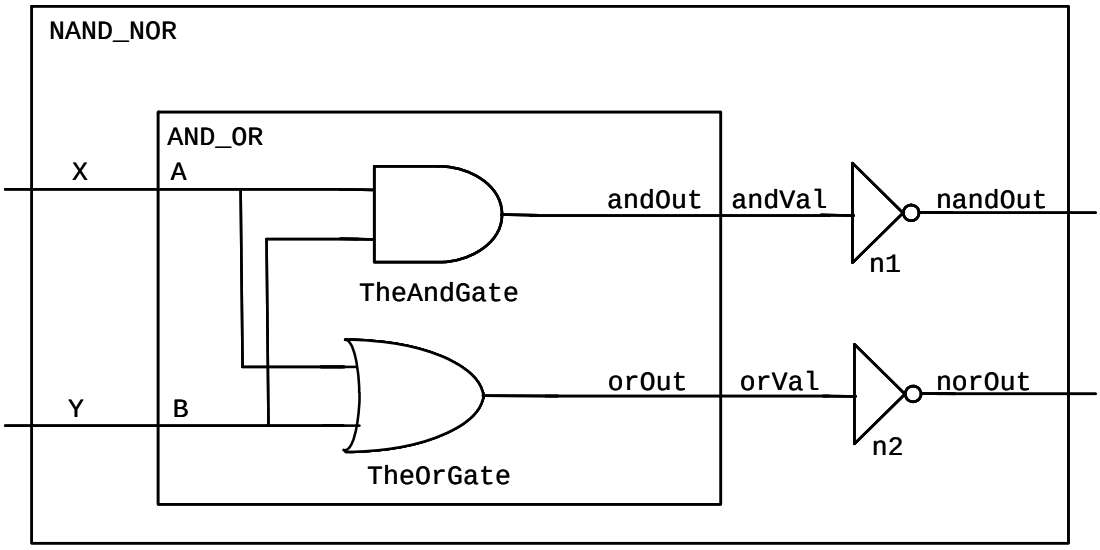
```
and <name> (OUT, IN1, IN2); // Sets output to AND of inputs
or <name> (OUT, IN1, IN2); // Sets output to OR of inputs
nand <name> (OUT, IN1, IN2); // Sets to NAND of inputs
nor <name> (OUT, IN1, IN2); // Sets output to NOR of inputs
xor <name> (OUT, IN1, IN2); // Sets output to XOR of inputs
xnor <name> (OUT, IN1, IN2); // Sets to XNOR of inputs
```

If you want to have more than two inputs to a multi-input gate, simply add more. For example, this is a five-input and gate:

```
and <name> (OUT, IN1, IN2, IN3, IN4, IN5); // 5-input AND
```

Hierarchy

Just like we build up a complex software program by having procedures call subprocedures, Verilog builds up complex circuits from modules that call submodules. For example, we can take our previous AND_OR module, and use it to build a NAND_NOR:




```
// Compute the logical AND and OR of inputs A and B.
module AND_OR(andOut, orOut, A, B);
  output logic  andOut, orOut;
  input  logic  A, B;

  and TheAndGate (andOut, A, B);
  or TheOrGate (orOut, A, B);
endmodule

// Compute the logical NAND and NOR of inputs X and Y.
module NAND_NOR(nandOut, norOut, X, Y);
  output logic  nandOut, norOut;
  input  logic  X, Y;
  logic         andVal, orVal;

  AND_OR aoSubmodule (.andOut(andVal), .orOut(orVal),
                      .A(X), .B(Y));
  not n1 (nandOut, andVal);
  not n2 (norOut, orVal);
endmodule
```

Notice that in the NAND_NOR procedure, we now use the AND_OR module as a gate just like the standard Verilog “and”, “not”, and other gates. That is, we list the module’s name, what we will call it in this procedure (“aoSubmodule”), and the outputs and inputs:

```
  AND_OR aoSubmodule (.andOut(andVal), .orOut(orVal),
                      .A(X), .B(Y));
```

Note that unlike C/C++/Java where we use the order of parameters to indicate which caller values connect to which submodule ports, in Verilog we explicitly name the ports. That is, when we say:

```
  .andOut(andVal)
```

We mean that the "andVal" wires in the caller module are connected to the "andOut" wires in the called submodule. This explicit naming tends to avoid mistakes, especially when someone adds or deletes ports inside the submodule. Note that every signal name in each module is distinct. That is, the same name can be used in different modules independently. In fact, if the caller module wants to hook a wire to a port of a submodule with the same name, there's a shorthand for that. For example, if we had the call:

```
AND_OR aoSubmodule (.andOut(andOut), .orOut(orVal),
                    .A(A), .B(B));
```

We could write that alternatively as:

```
AND_OR aoSubmodule (.andOut, .orOut(orVal), .A, .B);
```

This hooks andOut in the caller to andOut of the submodule, as well as A to A and B to B.

Just as we had more than one not gate in the NAND_NOR module, you can also call the same submodule more than once. So, we could add another AND_OR gate to the NAND_NOR module if we chose to – we simply have to give it a different name (like "n1" and "n2" on the not gates). Each call to the submodule creates new gates, so three calls to AND_OR (which creates an AND gate and an OR gate in each call) would create a total of 2*3 = 6 gates.

One new statement in this module is the "logic" statement:

```
logic  andVal, orVal;
```

This creates what are essentially local variables in a module. In this case, these are actual wires that carry the signals from the output of the AND_OR gate to the inverters.

Note that we chose to put the not gates below the AND_OR in this procedure. The order actually doesn't matter – the calls to the modules hooks gates together, and the order they "compute" in doesn't depend at all on their placement order in the code – all execute in parallel anyway. Thus, we could swap the order of the "not" and "AND_OR" lines in the module freely.

Boolean Equations and "Assign"

You can also write out Boolean equations in Verilog within an "assign" statement, which sets a "logic" variable to the result of a Boolean equation. Or is "|", and is "&", negation is "~", xor is "^". For example, we can compute not((A and B) or (C and D)) by:

```
assign F = ~((A & B) | (C & D));
```

True and False

Sometimes you want to force a value to true or false. We can do that with the numbers "0" = false, and "1" = true. For example, if we wanted to compute the AND_OR of false and some signal "foo", we could do the following:

```
AND_OR aoSubmodule (.andOut(andVal), .orOut(orVal),
                    .A(0), .B(foo));
```

Delays

Normally Verilog statements are assumed to execute instantaneously. However, Verilog does support some notion of delay. Specifically, we can say how long the basic gates in a circuit take to execute with the # operator. For example:

```
// Compute the logical AND and OR of inputs A and B.
module AND_OR(andOut, orOut, A, B);
  output andOut, orOut;
  input A, B;

  and #5 TheAndGate (andOut, A, B);
  or #10 TheOrGate (orOut, A, B);
endmodule
```

This says that the and gate takes 5 "time units" to compute, while the or gate is twice as slow, taking 10 "time units". Note that the units of time can be whatever you want – as long as you put in consistent numbers.

Defining constants

Sometimes you want to have named constants - variables whose value you set in one place and use throughout a piece of code. For example, setting the delay of all units in a module can be useful. We do that as follows:

```
// Compute the logical AND and OR of inputs A and B.
module AND_OR(andOut, orOut, A, B);
  output logic  andOut, orOut;
  input  logic  A, B;
  parameter delay = 5;

  and #delay TheAndGate (andOut, A, B);
  or #delay TheOrGate (orOut, A, B);
endmodule
```

This sets the delay of both gates to the value of "delay", which in this case is 5 time units. If we wanted to speed up both gates, we could change the value in the parameter line to 2.

Parameterized Design

Parameters can also be inputs to designs, that allow the caller of the module to set the size of features of that specific instance of the module. So, if we have a module such as:

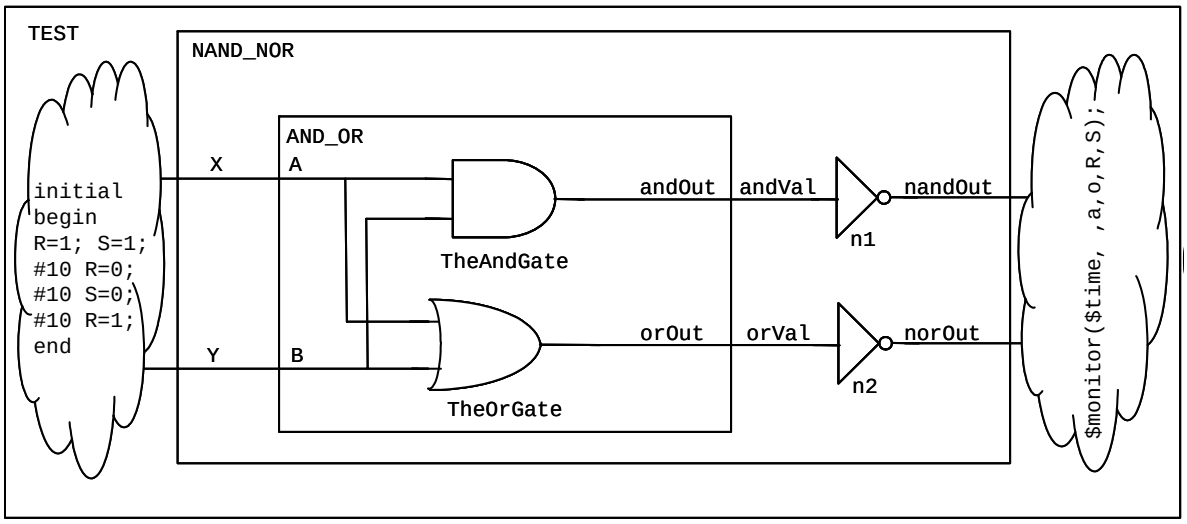
```
module adder #(parameter WIDTH=5) (out, a, b);
  output logic  [WIDTH-1:0]  out;
  input  logic  [WIDTH-1:0]  a, b;

  assign out = a + b;
endmodule
```

This defines a parameter "WIDTH" with a default value of 5 – any instantiation of the adder module that does not specify a width will have all of the internal variable widths set to 5. However, we can also instantiate other widths as well:

```
  // A 16-bit adder
  adder #(.WIDTH(16)) add1 (.out(o1), .a(a1), .b(b1));
  // A default-width adder, so 5-bit
  adder add2 (.out(o2), .a(a2), .b(b2));
```

Test benches

Once a circuit is designed, you need some way to test it. For example, we'd like to see how the NAND_NOR circuit we designed earlier behaves. To do this, we create a test bench. A test bench is a module that calls your device under test (DUT) with the desired input patterns, and collects the results. For example consider the following:




```
// Compute the logical AND and OR of inputs A and B.
module AND_OR(andOut, orOut, A, B);
  output logic  andOut, orOut;
  input  logic  A, B;

  and TheAndGate (andOut, A, B);
  or TheOrGate (orOut, A, B);
endmodule

// Compute the logical NAND and NOR of inputs X and Y.
module NAND_NOR(nandOut, norOut, X, Y);
  output logic  nandOut, norOut;
  input  logic  X, Y;
  logic         andVal, orVal;

  AND_OR aoSubmodule (.andOut(andOut), .orOut(orVal),
                      .A(X), .B(Y));
  not n1 (nandOut, andVal);
  not n2 (norOut, orVal);
endmodule
```

```
module NAND_NOR_testbench;  // No ports!
  logic  X, Y;
  logic  nandOut, norOut;

  initial begin // Stimulus
    X = 1; Y = 1; #10;
    X = 0;        #10;
          Y = 0; #10;
    X = 1;        #10;
  end

  NAND_NOR dut (.nandOut, .norOut, .X, .Y);

endmodule
```

The code to notice is that of the module “NAND_NOR_testbench”. It instantiates one copy of the NAND_NOR gate, called “dut” (device under test), and hooks up “logic” signals to all of the I/Os.

In order to provide test data to the dut, we have a stimulus block:

```
  initial begin // Stimulus
    X = 1; Y = 1; #10;
    X = 0;        #10;
          Y = 0; #10;
    X = 1;        #10;
  end
```

The code inside the “initial” statement is only executed once. It first sets X and Y to true. Then, due to the “#10” the system waits 10 time units, keeping X and Y at the assigned values. We then set X to false. Since Y wasn’t changed, it remains at true. Again we wait 10 time units, and then we change Y to false (X remains at false). If we consider the entire block, the inputs XY go through the pattern 11 -> 01 -> 00 -> 10, which tests all input combinations for this circuit. Other orders are also possible. For example we could have done:

```
  initial begin // Stimulus
    X = 0; Y = 0; #10;
          Y = 1; #10;
    X = 1; Y = 0; #10;
          Y = 1; #10;
  end
```

This goes through the pattern 00 -> 01 -> 10 -> 11. In fact, there’s a shorthand for doing this format:

```
  integer i;
  initial begin // Stimulus
    for(i=0; i<4; i++) begin
      {X,Y} = i; #10;
    end
```

```
end
```

We use the fact that integers are encoded in binary, and the binary values go through the pattern 000, 001, 010, 011, 100, 101, … If you want to put N binary signals through all combinations of inputs, then use the same code, but replace the upper limit of 4 in the loop condition with the integer whose value is 2^N.

Printing values to the console

Most development will use the waveforms in the simulator to show the state of wires over time. However, sometimes in debugging it is useful to print messages as well – for example, any time an error condition is found, you may want to print a text message saying what the error is, and print the value of some variables. The $display command does this. Specifically, if we did:

```
initial begin // Stimulus
  #1000;
  if (err != 0)
    $display($time, , "Found an error, code: ", err);
end
```

$time prints the time the $display routine happens. The comma without anything before it adds some space. The string and the value of err are also printed.

$display fires once, at the time specified. If you would like to be alerted whenever a value changes, use $monitor:

```
initial  // Response
  $monitor($time, , SEL, I, J, , V);
```

This code prints whenever any of the variables being monitored changes.

Register Transfer Level (RTL) Code

In the earlier sections, we showed ways to do structural designs, where we tell the system exactly how to perform the design. In RTL code we instead state what we want, and allow the Verilog system to automatically determine how to do the computation.

> ***Note: in RTL it is easy to forget how the hardware actually works, and pretend it's just C or Java. This is a great way to design AWFUL hardware. Because of this, we will give you stylized ways of using the constructs, which will guide you towards better versions. Our introductory logic class spends a lot of time showing what hardware is generated from various Verilog structures – think about the hardware your Verilog actually requires!***

In each of these cases, RTL code is done in an "always" block. If the computation is combination, it should be in an "always_comb" block, while computations that remember things, and thus are state-holding, should be in an "always_ff".

Begin-end

Begin and end statements merge together multiple statements into one, like the "{ }" braces in C and Java. For statements below such as "if-then-else" and "case", you can use begin and end to merge together multiple statements.

If-then-else

```
    logic  V1, V2;
```

```
always_comb begin
      if (A == 1) begin
            V1 = 1;
            V2 = 0;
      end else if (A == 0 & B == 1) begin
            V1 = 1;
            V2 = 1;
      end else begin
            V1 = 0;
            V2 = 0;
      end
end
```

You can set the output of a wire based upon the value of other signals. The if-then-else is similar to software programming languages. Note however that you should make sure that all signals are defined in all cases (i.e. it would be a problem to delete either V1 or V2 from any of these clauses).

If you think through this code, it is equivalent to a logic function. For example, V1 is true only when A == 1, or when A == 0 and B == 1. This is equivalent to V1 = A+(not(A) * B) = A + B. Similarly V2 = not(A) * B.

case

As we move to multi-bit signals, that can take on values more than just 0 and 1, the case statement becomes quite useful. The variable to be considered is placed in the "case ()" statement, and then different values are listed, with the associated action. For example, in the code below when the "state" variable is equal to 0, HEX is set to 0, while if the "state" variable is equal to 1, HEX is set to 1, and so on. There must also always be a "default" case, which is used when no other case matches. Also, like the if-then-else statement, any variable set in any part of the case statement should be set in all states. That is, dropping HEX from any of the "state" value lines would be incorrect.

In this code we use "1'bX" to indicate a 1-bit binary don't care in the default case, allowing the Verilog system to use Don't Cares in the minimization.

```
logic HEX;
always_comb begin
      case (state)
            0: HEX = 0;
            1: HEX = 1;
            2: HEX = 1;
            3: HEX = 0;
            4: HEX = 1;
            default: HEX = 1'bX;
      endcase
end
```

Declaring Multi-bit Signals

So far we have seen "logic" statements that create single-bit signals (i.e. they are just 0 or 1). Often you'd like to represent multi-bit values (for example, a 3-bit variable that can represent values 0..7). We can do this type of operation with the following declarations:

```
logic  [2:0] foo;   // a 3-bit signal (a bus)
logic  [15:0] bar;  // a 16-bit signal
```

These statements set up a set of individual wires, which can also be treated as a group. For example, the “logic [2:0] foo;” declares a 3-bit signal, which has the MSB (the 2^2’s place) as foo[2], the LSB (the 2^0’s place) as foo[0], and a middle bit of foo[1].

The individual signals can be used just like any other binary value in Verilog. For example, we could do:

```
and a1(foo[2], foo[0], c);
```

This AND’s together c and the 1’s place of foo, and puts the result in the 4’s place of foo.

Multi-bit signals can also be passed together to a module:

```
module random(bus1, bus2);
     output logic  [31:0]  bus1;
     input  logic  [19:0]  bus2;
     logic                 c;

     another_random ar1(.c, .bus2, .bus1);
endmodule
```

This module connects to two multi-bit signals (32 and 20 bits respectively), and passes both of them to another module “another_random”, which also connects to a single-bit wire c.

Multi-bit Signals – Common Error

When you are declaring multi-bit signals, you may get a warning message like:

"Warning! Port sizes differ in port connection (port 2) [Verilog-PCDPC] "

look for something like the following in your code:

```
     input logic  [31:0] d, reset, clk;
```

What that line does is declare 3 32-bit values. That is, d is [31:0] AND reset is [31:0] AND clk is [31:0]

What you actually want is:

```
     input  logic [31:0] d;
     input  logic        reset, clk;
```

Which declares d to be a 32-bit value, and reset and clk are 1-bit values.

Multi-bit Constants

In test benches and other places, you may want to assign a value to a multi-bit signal. You can do this in several ways, shown in the following code:

```
logic  [15:0] test;
initial begin // stimulus
     test = 12;
     #(10) test = 16'h1f;
     #(10) test = 16'b01101;
end
```

The 16-bit variable test is assigned three different values. The first is in decimal, and represents twelve. The second is a hexadecimal number (specified by the 'h) 1f, or 16+15

= 31. The last is a binary number (specified by the 'b) 01101 = 1+4+8 = 13. In each case the value is assigned, in the equivalent binary, to the variable test. Unspecified bits are padded to 0. So, the line:

```
test = 12;
```

is equivalent to:

```
test = 'b0000000000001100;
```

It sets test[2] and test[3] to 1, and all other bits to 0.

For Loops for Multi-bit Signals

Sometimes when we need to reorganize signals in a bus, a FOR loop can be helpful, particularly with mathematical calculations for the indexes.

```
logic  [7:0] LEDG;
integer i;

always_comb begin
      for (i=0; i<8; i=i+1)
            LEDG[7-i] = GPIO_0[28+i];
      for (i=0; i<10; i=i+1)
            LEDR[9-i] = GPIO_0[18+i];
end
```

In this code we set LEDG[7] = GPIO_0[28], LEDG[6] = GPIO_0[29], etc.

Multi-Dimensional Buses

Sometimes it can be useful to have structures with more than one dimension – for example, we might want to hold 16 8-bit values. Verilog allows you to define multiple sets of indexes for a variable:

```
logic  [15:0][7:0] string;
```

To index a value, you move left-to-right through the indices. For example, the following code sets all the bits of a 4-dimensions bus to 0:

```
logic  [15:0][9:0][7:0][3:0] vals;
integer i, j, k, l;

always_comb begin
   for(i=0; i <= 15; i++)
      for(j=0; j<=9; j++)
         for(k=0; k<=7; k++)
            for(l=0; l<=3; l++)
               vals[i][j][k][l] = 1'b0;
end
```

Subsets

Sometimes you want to break apart multi-bit values. We can do that by selecting a subset of a value. For example, if we have

```
logic  [31:0] foo;
initial foo[3:1] = 3'b101;
```

This would set foo[3] = 1, foo[2] = 0, and foo[1] = 1. All other bits of foo will not be touched. We could also use the same form to take a subset of a multi-bit wire and pass it as an input to another module.

Note that this subdividing can be done to save you work in creating large, repetitive structures. For example, consider the definition of a simple 16-bit register built from a base D_FF unit:

```
module D_FF16(q, d, clk);
  output logic  [15:0] q;
  input  logic  [15:0] d;
  input  logic  clk;

  D_FF d0 (.q(q[0]),  .d(d[0]),  .clk);
  D_FF d1 (.q(q[1]),  .d(d[1]),  .clk);
…
  D_FF d15(.q(q[15]), .d(d[15]), .clk);
endmodule
```

with the 16 separate D_FF lines there's a good likelihood you'll make a mistake somewhere. For a 32-bit register it's almost guaranteed. We can do it a bit more safely by repeatedly breaking down the problem into pieces. For example, write a 4-bit register, and use it to build the 16-bit register:

```
module D_FF4(q, d, clk);
  output logic  [3:0]  q;
  input  logic  [3:0]  d;
  input  logic         clk;

  D_FF d0(.q(q[0]), .d(d[0]), .clk);
  D_FF d1(.q(q[1]), .d(d[1]), .clk);
  D_FF d2(.q(q[2]), .d(d[2]), .clk);
  D_FF d3(.q(q[3]), .d(d[3]), .clk);
endmodule

module D_FF16(q, d, clk);
  output logic  [15:0]  q;
  input  logic  [15:0]  d;
  input  logic          clk;

  D_FF4 d0( .q(q[3:0]),   .d(d[3:0]),  .clk);
  D_FF4 d1( .q(q[7:4]),   .d(d[7:4]),  .clk);
  D_FF4 d2(.q(q[11:8]),  .d(d[11:8]),  .clk);
  D_FF4 d3(.q(q[15:12]), .d(d[15:12]), .clk);
endmodule
```

Concatenations

Sometimes instead of breaking apart a bus into pieces, you instead want to group things together. Anything inside {}'s gets grouped together. For example, if we want to swap the low and high 8 bits of an input to a D_FF16 we could do:

```
  logic  [15:0]  data, result;

  D_FF16 d1(.q(result), .d({data[7:0], data[15:8]}), .clk);
```

Pretty much anything can go into the concatenation – constants, subsets, buses, single wires, etc.

Bit Replication in Concatenations

Sometimes you would like to copy a bit multiple times in a concatenate (very useful for sign-extension in 2's Complement numbers). You can do that with the following construct:

```
  logic  [15:0]  large;
  logic  [7:0]   small;

  assign large = {{8{small[7]}, small}};
```

Here, the first bit is copied 8 times, then the entire number appears. This means you will have 9 instances of the top bit, followed by one of each of the next bits.

Sequential Logic

In combinational logic we start an always block with "always_comb", which means the logic output is recomputed every time any of the inputs changes. For sequential logic, we need to introduce a clock, which will require a somewhat different always statement:

```
// D flip-flop w/synchronous reset
module D_FF (q, d, reset, clk);
  output logic  q;
  input  logic  d, reset, clk;

  always_ff @(posedge clk) begin // Hold val until clock edge
    if (reset)
      q <= 0;  // On reset, set to 0
    else
      q <= d; // Otherwise out = d
  end

endmodule
```

Most of this should be familiar. The new part is the "always_ff @(posedge clk)". We capture the input with the "always_ff @(posedge clk)", which says to only execute the following statements at the instant you see a positive edge of the clk. That means we have a positive edge-triggered flip-flop. We can build a negative edge-triggered flip-flop via "always_ff @(negedge clk)".

Assignment styles: = vs. <=

Verilog includes two different types of ways to assign values to variables: = vs. <=. They are subtly different:

- = The assignment occurs immediately. A subsequent line will see the new value. So, a = b; c = a; will set both a and c to the value contained in b.
- <= The value to be assigned is computed immediately, but the assignment itself is delayed until all simultaneously executing code has been evaluated. So, a<=b; b<=a; will swap the values of a and b. However, a<=b; c<= a; will set a to the value of b, and c to the old value of a (the value before the value of b is written to a).

The two kinds of assignment can be confusing, and mixing them in one "always" block is a recipe for disaster. However, things are much easier if you obey the following rules:

1. Inside always_ffs @(posedge clk) blocks use <= for everything (except the iteration variable in a for loop).
2. For assign statements and always_comb blocks use = for everything.
3. Avoid complex logic in always_ff @(posedge clk) blocks – instead, compute complex logic in always_comb blocks, and then just have statements like "ps <= ns" in always_ff @(posedge clk) blocks.

Clocks

A sequential circuit will need a clock. We can make the test bench provide it with the following code:

```
logic  clk;
parameter PERIOD = 100; // period = length of clock
                        // Make the clock LONG to test
initial begin
  clk <= 0;
  forever #(PERIOD/2) clk = ~clk;
end
```

This code would be put into the testbench code for your system, and all modules that are sequential will take the clock as an input.

Enumerations

For FSMs and the like, we want to have variables that can take on one of multiple named values – while we could just use numbers, names are easier to use. Sometimes we may use PARAMETER statements to set up names for variables, but for FSM state variables enumerations work better. For example, the following code defines the allowable states for an FSM:

```
  enum { RED, BLUE, GREEN} ps, ns;
```

This defines two variables, ns and ps, and requires that their values be either RED, GREEN, or BLUE. We can test the value of variables, and assign new values, using those names:

```
  always_comb begin
    if (ps == RED)
      ns = BLUE;
    else
      …
```

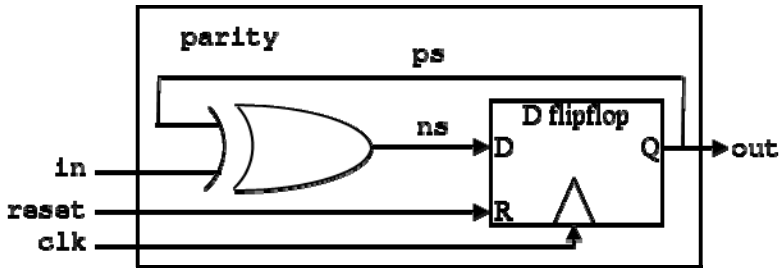
Verilog will then assign specific values to each of these variables. If you want to set specific values, you can use:

```
enum { RED=0, BLUE=1, GREEN=2 } ps, ns;
```

Make sure to have one of the values be equal to 0, but the other values can be whatever you want. One last tip – if you want to print the value of an enum variable ps, you can call ps.name to return the string for that value (i.e. if ps is 0, ps.name will return RED).

Example Finite State Machine

Here's an example of a simple sequential circuit, with all of its gory details. Note that this circuit computes parity – the output is true when the circuit has seen an odd number of trues on its input.




```
// Parity example

module parity (out, in, reset, clk);
  output logic  out;
  input  logic  in, reset, clk;
  logic         ps, ns;

  always_comb begin
    ns = ps ^ in;
  end

  always_ff @(posedge clk) begin
    if (reset)
      ps <= 0;
    else
      ps <= ns;
  end

  assign out = ps;
endmodule

module parity_testbench;
  logic  in, reset, clk;
  logic  out;
```

```
  parameter period = 100;

  parity dut (.in, .reset, .clk, .out);

  initial begin
    clk <= 1;
    forever #(period/2) clk <= ~clk;
  end

  initial begin
    reset <= 1; in <= 0; @(posedge clk);
    reset <= 0;          @(posedge clk);
                in <= 1; @(posedge clk);
                         @(posedge clk);
                in <= 0; @(posedge clk);
                in <= 1; @(posedge clk);
                in <= 0; @(posedge clk);
                         @(posedge clk);
    $stop();
  end

endmodule
```

Advanced Features – assert statements

As you design larger systems, you will often have assumptions you'd like to make sure are true. For example, you may have a parameterized module, but there are only a few legal values of that parameter. Or, you may have a module that assumes the inputs obey certain requirements. You could check this via simulation, but as the design gets larger you are more and more likely to miss things.

The solution to this is the "assert" statement, that in simulation will raise an error whenever the value inside the assertion is false. So, if we have a parameter with only a few legal values, we can test it with an assertion inside the module:

```
  initial assert(WIDTH>1 && WIDTH<=19);
```

If we require that at least one input to a unit must always be true, we can test it with an always-running assertion:

```
  always_ff @(posedge clk) begin
    assert(reset || a != 3'b000 || b);
  end
```

Advanced Features – generate statements

Earlier in this tutorial we showed how to build a 16-bit register by using a hierarchy of modules, one that does a 4-bit register, and another that uses 4 of these 4-bit registers to build a 16-bit register. If we want to make a completely parameterized version, where the size can be any length at all, we can use a generate statement. Generate allows us to put submodule calls and other logic within "for" loops and "if" statements, allowing the logic to decide the number of modules actually instantiated. Note that any iteration variables must be declared "genvar", and any for loops or if statements must have a begin – end block with a label (an identifier, such as "eachDff" in the code below).

```
module DFF_VAR #(parameter WIDTH=8) (q, d, clk);
  output logic  [WIDTH-1:0]  q;
  input  logic  [WIDTH-1:0]  d;
  input  logic               clk;

  initial assert(WIDTH>0);

  genvar i;

  generate
    for(i=0; i<WIDTH; i++) begin : eachDff
      D_FF dff (.q(q[i]),  .d(d[i]),  .clk);
    end
  endgenerate
endmodule
```

Note that for the case of the register, one could just use a parameter and the FSM format to do the same thing:

```
module DFF_VAR #(parameter WIDTH=8) (q, d, clk);
  output logic  [WIDTH-1:0]  q;
  input  logic  [WIDTH-1:0]  d;
  input  logic               clk;

  initial assert(WIDTH>0);

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

But, the register is a simple way to show the power of generate statements, which can be useful in some situations that often cannot be handled in any other way. Below are a few other examples to help you see how generate statements can be used.

First is a version of a 99 position tug-of-war game, with edge conditions. Note that you could also do the left and right edge positions outside of the generate loop and adjust the loop variable accordingly.

```
module tugOfWar99 (clk, rst, L, R, leds);

      input  logic         clk, rst, L, R;
      output logic [98:0]  leds;

      genvar i;
      generate
            for(i = 0; i < 99; i++) begin : eachLight

                  // The right edge light
                  if (i == 0)
                        normal_light led (.clk, .rst, .L, .R,
.NL(leds[i+1]), .NR(1'b0),      .lightOn(leds[i]));

                  // The center light
                  else if (i == 50)
                        center_light led (.clk, .rst, .L, .R,
.NL(leds[i+1]), .NR(leds[i-1]), .lightOn(leds[i]));

                  // The left edge light
                  else if (i == 98)
                        normal_light led (.clk, .rst, .L, .R,
.NL(1'b0),      .NR(leds[i-1]), .lightOn(leds[i]));

                  // Any other light
                  else
                        normal_light led (.clk, .rst, .L, .R,
.NL(leds[i+1]), .NR(leds[i-1]), .lightOn(leds[i]));

            end
      endgenerate

endmodule
```

Here is an example for setting up a control FSM for each position of a 16x16 LED array.

```
module controller2D(clk, rst, row_sel, col_sel, leds);

      input logic  clk, rst;
      input logic  [15:0]       row_sel, col_sel;
      output logic [15:0][15:0] leds;

      genvar x,y;
      generate
            for(x = 0; x < 16; x++) begin : eachRow
                  for(y = 0; y < 16; y++) begin: eachCol
```

```
                        light pixel (.clk, .rst, .row(row_sel[x]),
.col(col_sel[y]), .lightOn(leds[x][y]));

                end
            end
      endgenerate
endmodule

module light (clk, rst, row, col, lightOn);

      input  logic clk, rst, row, col;
      output logic lightOn;

      always_ff @ (posedge clk) begin
            if (rst)
                  lightOn <= 1'b0;
            else
                  lightOn <= row & col;
      end

endmodule
```