SystemVerilog Tutorial

Scott Hauck, Justin Hsia, Max Arnold, Matthew Cinnamon (last updated Jan. 2021)

Introduction

The following tutorial is intended to get you going quickly in circuit design in SystemVerilog. It is not a comprehensive guide but should contain everything you need to design circuits in this class. For a more thorough reference, Prof. Hauck recommends Vahid and Lysecky's *Verilog for Digital Design*.

Table of Contents

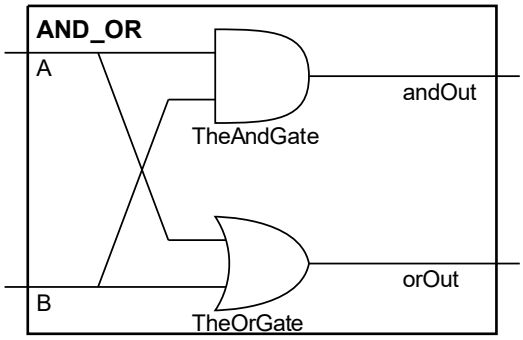
Modules

The basic building block of Verilog is a module. This is similar to a function or procedure in C/C++/Java in that it takes input values, performs a computation, and generates outputs. However, modules compile into collections of logic gates and each time you "call" a module you are creating separate instances of hardware.

Simple Module Example

Shown below is an example of a SystemVerilog module (left) and its corresponding hardware instantiation (right):

```
1 // Compute the logical AND and OR
2 //  of inputs A and B.
3 module AND_OR(andOut, orOut, A, B);
4  output logic andOut, orOut;
5  input logic A, B;
6
7  and TheAndGate (andOut, A, B);
8  or TheOrGate (orOut, A, B);
9 endmodule
```



Line-by-Line Analysis

Lines 1-2 are **single-line comments**, designated by the '`//`' (**green** syntax highlighting in Quartus) and ignored during compilation. Comments can be placed at the end of lines of code or on separate lines by themselves.

Lines 3-5 define the **module name** and **port list**, which is the list of inputs and outputs signals. Line 3 gives the port names while lines 4-5 define the port *types* and *directions*. Here, all 4 port signals are of type `logic`, `andOut` and `orOut` are outputs, and `A` and `B` are inputs. The name (`AND_OR`) is user-defined but *must* start with a letter and can only consist of letters, numbers, and underscores. Avoid using keywords (`blue` syntax highlighting in Quartus) as names.

Line 7-8 each instantiate a gate following the standard module instantiation syntax of:

```
<module_name> <instance_name> (<port_connections>);
```

Line 7 creates an AND gate called `TheAndGate` and Line 8 creates an OR gate called `TheOrGate`. The port lists are explained below in the section
Basic Gates.

The `endmodule` keyword on line 9 closes the module definition started with the `module` keyword on line 3.

ANSI-style Module Headers

SystemVerilog allows for "ANSI-style" module headers, which allows you to define the port types and directions within the port list. This can save a lot of space when working with large module headers because the port names are not repeated. The following example is equivalent to the previous module:

```
1 // Compute the logical AND and OR of inputs A and B (ANSI-style)
2 module AND_OR (output logic andOut, orOut,
3                input logic A, B);
4  and TheAndGate (andOut, A, B);
5  or TheOrGate (orOut, A, B);
6 endmodule
```

Basic Gates

Verilog comes with a number of predefined modules for basic gates that follow the standard module instantiation syntax of:

```
<module_name> <instance_name> (<port_connections>);
```

The port lists for these gates are defined such that the first connection is always the output. The following examples show a one-input gate and a multi-input gate:

```
1 not <name> (OUT, IN);     // 1-input, sets OUT to opposite of input
2 and <name> (OUT, IN1, IN2); // 2-input, OUT is logical AND of inputs
```

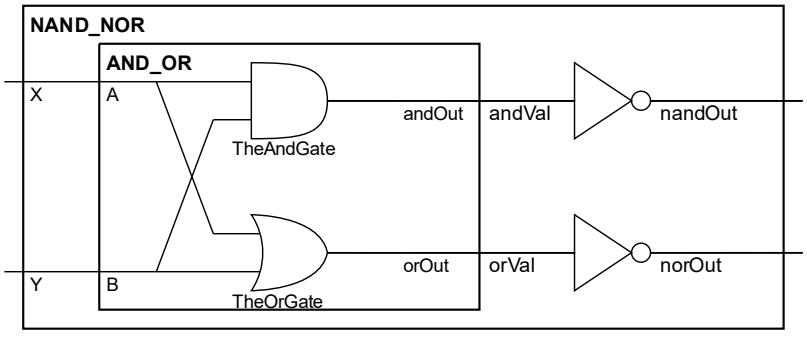
The other 1-input gate is `buf` and the other multi-input gates are `or`, `nand`, `nor`, `xor`, and `xnor`.

If you want to have more than two inputs to a multi-input gate, simply add more connections to the port list. For example, the following is a five-input AND gate:

```
1 and bigAnd (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 instantiate submodules. For example, we can take our previous `AND_OR` module and use it to build a `NAND_NOR` module:



```
1 // Compute the logical NAND and NOR of inputs X and Y.
2 // The AND_OR module definition can be in the same file or
3 //  in a separate file in the same project.
4 module NAND_NOR (nandOut, norOut, X, Y);
5  output logic nandOut, norOut;
6  input logic X, Y;
7  logic andVal, orVal; // local signals (not ports)
8
9  AND_OR aoSubmodule (.andOut(andVal), .orOut(orVal), .A(X), .B(Y));
10  not n1 (nandOut, andVal);
11  not n2 (norOut, orVal);
12 endmodule
```

Notice that we now instantiate the AND_OR module just like the standard Verilog gates. We also happen to have multiple NOT gates in this module. You can instantiate the same type of module (basic or user-defined) more than once as long as the instance names are different (here, n1 and n2 for the not gates).

Local Signals

Line 7 creates local signals andVal and orVal, which are essentially local variables in the NAND_NOR module. In this case, these are wires that carry the signals from the output of the AND_OR gate to the inverters.

Structural Verilog and Code Ordering

The creation/instantiation of signals and modules as seen so far is considered **structural Verilog**: the code only describes the connections between different pieces of hardware. None of this code has any notion of sequencing or timing—all pieces of hardware will execute in parallel—so the statement order does not matter. Thus, we could freely swap the ordering of lines 9-11.

Port Connections

Like C/C++/Java arguments and parameters, Verilog will, by default, connect the ports in order of the port list of the module definition when you instantiate a module. However, we can also explicitly name the ports in Verilog. That is, when we use .andOut(andVal) in the port list, we are connecting the andVal wire in the NAND_NOR module to the andOut input port of the AND_OR module instance. This **explicit connection** tends to avoid mistakes, especially when someone adds or deletes ports in a module definition.

Note that every signal name in a module must be distinct. However, the same name can be used in different modules independently. You can connect a module signal to a submodule port of the same name using an **implicit connection**. For example, if we had renamed the X and Y input ports as A and B, then:

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

could be equivalently written as:

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

While the andOut and orOut ports are still explicitly connected, the A and B ports will be implicitly connected to the A and B signals of the NAND_NOR module (which just happen to also be input ports).

There is another style of implicit connection which will implicitly connect as many ports as it can (`.*`). This saves a lot of space when connecting many signals. The following is again equivalent to the two examples above.

```
AND_OR aoSubmodule (.andOut(andVal), .orOut(orVal), .*);
```

Boolean Equations and "Assign"

You can also write out Boolean equations in Verilog within a continuous assignment statement (`assign`), which sets the value of a variable to the result of a Boolean expression. The Boolean expressions can be constructed using the bitwise operators NOT (~), OR (|), AND (&), and XOR (^). An example:

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

Note that the value of F will automatically update (though it may not change) after *any* of its inputs (A, B, C, D) change. F should not be assigned anywhere else in its module or there will be a conflict.

True and False

Sometimes you want to force a single-bit value to true/high or false/low. We can do that with the constants `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));
```

Multi-bit Signals

So far, we have created/declared single-bit signals (*i.e.*, they can only have the value `0` or `1`) using `logic` statements. However, Verilog also supports multi-bit signals. The following multi-bit declaration example creates a 3-bit variable named `x`:

```
logic [2:0] x; // a 3-bit signal that can hold values 0-7
```

This syntax, called a *packed array*, creates a set of individual signals that can also be treated as a group/bus. The numbers inside the brackets (`[]`) define the signal indices `L:R` with `L` being the most-significant bit (`x[2]` in this case) and `R` being the least significant bit (`x[0]`). By convention, for an n-bit signal we use n-1 as `L` and `0` as `R`. The individual signals can be used just like any other single-bit signal in Verilog. For example:

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

This computes `c` AND the right-most bit of `x` and ties the result to the left-most bit of `x`. Multi-bit signals can also be passed as a group to a module through multi-bit ports:

```
module foo (bus1, bus2);
 output logic [31:0] bus1;
 input logic [19:0] bus2;
 logic c;
 bar ar1 (.bus1, .bus2, .c); // connect ports, including buses
endmodule
```

Note that the bus widths of multi-bit ports and connected signals must match.

Multi-bit Signal Declaration Issue

Be aware that all signals declared in the same statement take on the same properties. A common error with signal declarations looks something like:

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

What this line does is declare all three signals as 32-bit variables, whereas `reset` and `clk` are generally one-bit signals. You may be tipped off to this error if you see a compiler warning about signal or port size mismatches. What you actually want is:

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

Array Slices

Sometimes you want to work with a subsection of a multi-bit value, called a *slice*. We can create a slice by specifying the start and end indices of the slice separated by a colon. For example, the following code sets `foo[3]` and `foo[1]` to `1` and `foo[2]` to `0`, leaving all other bits unchanged:

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

A common use case for slices is breaking apart large packed arrays to pass to submodules. For example, a single byte can be represented as two hex digits. If we have a module named `hex_converter` which converts a 4-bit value to the segments of a 7-segment display, the following code displays the hex code for a byte:

```
1 logic [7:0] value;
2 logic [6:0] out1, out2; // the signals for the 7-seg displays
3 hex_converter hi (.in(value[7:4]), .out(out1));
4 hex_converter lo (.in(value[3:0]), .out(out2));
```

Multi-bit Literals

You can assign a value to a multi-bit signal using a *multi-bit literal*. The general format contains 5 parts in the following order, most of which can be omitted and will assume a default value:

1) the width of the literal in bits (default: 32)
2) a single quote (`'`)
3) an ‘s’ character if the number is signed (default: unsigned)
4) the radix indicator (‘b’ for binary, ‘o’ for octal, ‘d’ for decimal, or ‘h’ for hex; default: decimal)
5) the value expressed in the specified radix

For example, the following statements produce equivalent behavior for a 16-bit signal `test`:

```
1 test = 27;     // 32-bit unsigned decimal
2 test = 16'h1B;   // 16-bit unsigned hexadecimal
3 test = 16'sb11011; // 16-bit signed decimal
```

On line 1, the value will be automatically truncated to 16 bits by the compiler (you will get a warning for this).

On line 3, the literal is specified as a signed constant, but has the same binary representation as its unsigned equivalent.

Unspecified bits default to `0` (*i.e.*, can be treated as leading zeros). So, the following is equivalent to Line 3:

```
3 test = 'b0000000000011011;
```

Concatenations

You can concatenate multiple signals together into a new grouping by separating them with commas inside curly braces (`{}`). For example, if we have two 8-bit signals called `b1` and `b2`, then `{b1, b2}` creates a 16-bit signal. Note that the ordering of bits of the new grouping is determined by the ordering of the signals within the braces.

All types of signals can be used in a concatenation – constants, subsets, buses, single wires, expressions, etc. For example, the following is a concatenation of a constant with a multi-bit signal:

```
1 logic [3:0] val;
2 logic [7:0] result;
3 assign result = {4'hC, val};
```

Bit Replication

Sometimes you would like to replicate a bit or set of bits multiple times. The syntax uses two sets of nested curly braces with the value to be replicated on the inside and a constant for the number of replications between the opening braces. The following example replicates the 4-bit signal `val` three times:

```
1 logic [3:0] val;     // 4 bits
2 logic [11:0] result;    // 12 bits
3 assign result = {3{val}}; // 4 bits replicated 3 times = 12 bits
```

Like all other signal expressions, the replication operator can be used inside of a concatenation and vice versa.

Delays

Normally, Verilog statements are assumed to execute instantaneously. However, Verilog does support some notion of execution delay for basic gates via the `#` operator (*simulation only!!!*). For example:

```
1 // Compute the logical AND and OR of inputs A and B
2 module AND_OR(andOut, orOut, A, B);
3  output andOut, orOut;
4  input A, B;
5
6  and #5 TheAndGate (andOut, A, B);
7  or #10 TheOrGate (orOut, A, B);
8 endmodule
```

This says that the AND gate takes 5 arbitrary time units to compute, while the OR gate takes 10 units. Note that the units of time can be whatever you want as long as you are consistent relative to other delays. This can be handy when testing and simulating designs to mimic combination delays through actual gates in the real world.

Defining Named Constants

Sometimes you want to have named constants – values you associate with a meaningful name that you can reuse throughout a piece of code. For example, you could set the same delay for all units in a module:

```
1 // Compute the logical AND and OR of inputs A and B
2 module AND_OR(andOut, orOut, A, B);
3  output logic andOut, orOut;
4  input logic A, B;
5  parameter delay = 5;
6
7  and #delay TheAndGate (andOut, A, B);
8  or #delay TheOrGate (orOut, A, B);
9 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 would only have to reduce the constant in the parameter line.

You may also see parameters used to define names for states in a state machine. However, this particular use case can be accomplished via an `enum`, which is detailed later.

Parameterized Design

Parameters can also be inputs to designs, allowing 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:

```
1 module adder #(parameter WIDTH=5) (out, a, b);
2  output logic [WIDTH-1:0] out;
3  input logic [WIDTH-1:0] a, b;
4
5  assign out = a + b;
6 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:

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

Note that this allows us to instantiate modules of varying size from a single Verilog definition, similar to templates in C++. You cannot instantiate “on the fly” so parameter values must be known at compile time.

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 generally clearer. 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 a hypothetical FSM:

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

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

```
1 always_comb begin
2  if (ps == RED)
3    ns = BLUE;
4  else
5  ...
```

The compiler will automatically assign values to each of these variables or you can also specify one or more specific values:

```
1 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 (*e.g.*, if the current value of `ps` is `0`, `ps.name` will return "`RED`").

Register Transfer Level (RTL) Code – Behavioral Verilog

In the earlier sections, we showed ways to create structural designs, where we tell the system exactly how to arrange the design. In RTL (*i.e.*, behavioral) code, we instead state only the behavior we want and allow the Verilog compiler to determine the actual circuit layout.

In behavioral code, it is easy to forget how the hardware actually works and pretend that it is just C or Java (*i.e.*, software). 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 designs – think about the hardware that your system actually requires!

RTL code is divided into two types, combinational logic and sequential logic. In each of these cases, RTL code is written in `always` block variants, which allow for the use of powerful constructs like `if-else` `if-else` and `case` statements. This helps greatly when creating large and complex designs.

Begin-end

`begin` and `end` statements denote a block of code, much like the "`{}`" braces in C and Java, and group multiple statements together in determining computation "flow." These can be used with procedural blocks (*e.g.*, `initial`, `always`) and conditional statements (*e.g.*, `if-else` `if-else`, `case`). Like C, you do not *have* to use them for single statements, but they generally help to create more readable and maintainable code.

Combinational Logic

The output of combinational logic is determined purely based on the current value of the inputs. Simple computations can be written as `assign` statements, but complex designs generally go in `always_comb` blocks, which will execute/trigger whenever any of the input signals change.

Sequential Logic

Sequential logic only updates its output when a specific event, typically a clock trigger, occurs. This allows the design to hold/store state information and behave differently based on previous inputs. The event (*i.e.*, expression that evaluates to true) is specified after the `always_ff` keyword and usually triggers from low to high, `posedge`, or high to low, `negedge`. For example, the following block triggers on a rising edge of `clk`:

```
always_ff @(posedge clk)
```

Blocking (=) and Non-blocking (<=) Assignment

Verilog includes two subtly different types of ways to assign values to variables, blocking (=) and non-blocking (<=):

- A blocking assignment takes effect *before* subsequent statements in the same block, which will see the new value. So, `a = b; c = a;` will set both a and c to the value of b.
- A non-blocking assignment occurs simultaneously (*i.e.*, in parallel) with all others. So, `a <= b; b <= a;` will swap the values of a and b. Similarly, `a <= b; c <= a;` will set a to the value of b, and c to the original/old value of a.

The two kinds of assignment can be confusing and mixing them in one `always` block is a recipe for disaster. The following rules are an effective "rules of thumb" to avoid any issues:

1. Use <= for everything inside `always_ff` blocks (except the iteration variable in a `for` loop).
2. Use = for everything inside `assign` statements and `always_comb` blocks.
3. Avoid complex logic in `always_ff` blocks. Instead, compute complex logic in `always_comb` blocks, assign the results to internal signals, and then use simple statements like "`ps <= ns`" in the `always_ff` blocks.

If-else if-else

The `if-else if-else` constructs look similar to software, but with the important distinction that e*ach signal must be defined in all cases*. This is because the statement is equivalent to defining a logic function. `V1` is true only when `A = 1`, or when `A = 0` and `B = 1`. This is equivalent to the function `V1 = A + B`. Similarly, `V2 = ĀB`.

```
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
```

Failing to define a signal in all cases will result in one of two outcomes:

1) If the conditional block is inside an `always_comb` block, it will fail with the message: *always_comb construct does not infer purely combinational logic*.
2) If not in an `always_comb` block, the code may compile with the warning: *inferring latch(es) for variable "var", which holds its previous value in one or more paths through the always construct.*

Although it may compile successfully, the design will likely exhibit unexpected behavior that is not intended to be used in this class and therefore should be avoided.

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 parenthesis of the `case` statement, and then different values are listed with the associated action. For example in the code below, `HEX` is set to `0` when the `state` variable is `0` or `3` and is set to `1` if `state` is `1`, `2`, or `4`. There must also always be a `default` case to catch values that don't match any other case. Any variable set in any part of the `case` statement should be set in all cases. That is, dropping the `HEX` assignment from any of the cases would be incorrect. Here, we also use the value `1'bX` to indicate that we do not care if the value is `1` or `0`, allowing the compiler to optimize the design.

```
1 logic HEX;
2 always_ff @(posedge clk) begin
3  case (state)
4    0: HEX = 0;
5    1: HEX = 1;
6    2: HEX = 1;
7    3: HEX = 0;
8    4: HEX = 1;
9    default: HEX = 1'bX;
10  endcase
11 end
```

Repeat

A `repeat` loop is a simple statement that repeats a block of code a specified number of times. If you don't need to use the value of the loop variable, this is a much simpler syntax than writing out a `for`-loop. For example, the following code adds `5` to `val` ten times with an 8 arbitrary time unit delay between each addition.

```
1 repeat (10) begin
2  #8 val = val + 5;
3 end
```

For-Loops

A `for`-loop in SystemVerilog is a much more limited structure compared to other programming languages. It should only be used as a way of simplifying repetitive statements, not as a way of performing any form of logic. A good rule to follow is:

⚠ If you cannot manually expand a `for`-loop in your code, do not use it!

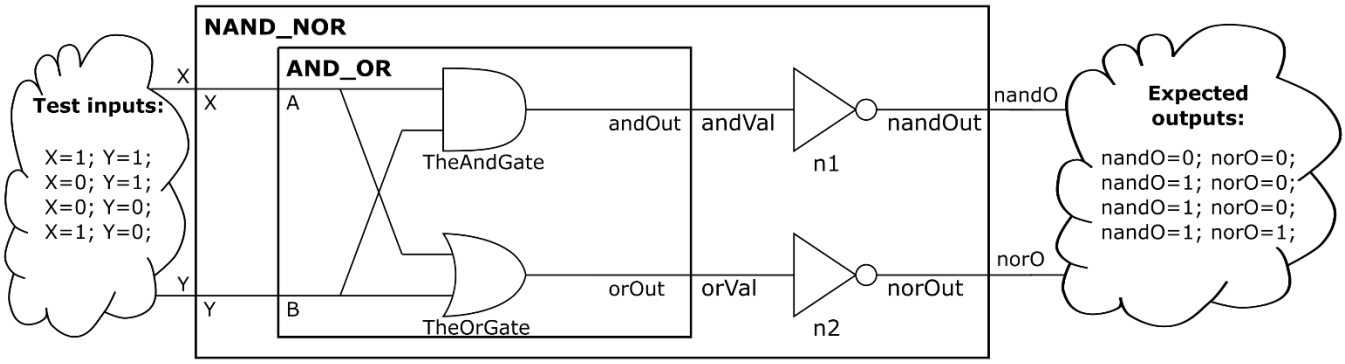
The main scenario where `for`-loops are useful in SystemVerilog is working with multi-bit signals. Sometimes array slices are insufficient, for example if you want to assign bits in reverse order:

```
1 logic [7:0] LEDG;
2
3 always_comb begin
4  for (int i = 0; i < 8; i = i + 1)
5    LEDG[7-i] = GPIO_0[28+i];
6 end
```

`for`-loops can also be useful in test benches, as they often involve repetitive inputs which can be simplified.

Test Benches

Once a circuit is designed, you need some way to test it. For example, we would 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 original module, often called the Device Under Test or DUT, with the desired input patterns and collects the results. For example, consider the following setup:




```
1  // Compute the logical AND and OR of inputs A and B
2  module AND_OR(andOut, orOut, A, B);
3   output logic andOut, orOut;
4   input logic A, B;
5
6   and TheAndGate (andOut, A, B);
7   or TheOrGate (orOut, A, B);
8  endmodule
9
10 // Compute the logical NAND and NOR of inputs X and Y
11 module NAND_NOR(nandOut, norOut, X, Y);
12  output logic nandOut, norOut;
13  input logic X, Y;
14  logic andVal, orVal;
15
16  AND_OR aoSubmodule (.andOut, .orOut, .A(X), .B(Y));
17  not n1 (nandOut, andVal);
18  not n2 (norOut, orVal);
19 endmodule
```

```
20
21 module NAND_NOR_testbench(); // No ports
22  logic nandOut, norOut;
23  logic X, Y;
24
25  initial begin // Stimulus
26    X <= 1; Y <= 1; #10;
27    X <= 0; #10; // unchanged signals retain their old value
28    Y <= 0; #10;
29    X <= 1; #10;
30    $stop(); // explicitly halt/suspend the simulation (optional)
30  end
31  NAND_NOR dut (.nandOut, .norOut, .X, .Y);
32 endmodule
```

The new section of code in this example is the module `NAND_NOR_testbench`. It instantiates one copy of the `NAND_NOR` module, called `dut`, and connects input signals that we control and output signals that we can read.

In order to provide test data to the device under test, we create a stimulus block (lines 25-30). The code inside the `initial` statement is executed once at the beginning of the simulation. It sets `X` and `Y` to `1` immediately and then waits 10 arbitrary time units, keeping `X` and `Y` at the assigned values. It then set `X` to `0`, keeping `Y` unchanged, and again waits 10 time units. If we consider the entire block, the inputs `XY` go through the bit patterns `2'b11` → `2'b01` → `2'b00` → `2'b10`, which tests all possible input combinations for this circuit. Other orderings are possible and valid as long as they contain all desired input combinations. For example:

```
1 initial begin // Stimulus
2   X <= 0; Y <= 0; #10;
3   Y <= 1; #10;
4   X <= 1; Y <= 0; #10;
5   Y <= 1; #10;
6 end
```

A `for`-loop can be used to simplify sweeping through inputs. The following code is identical in function to the previous example.

```
1 initial begin // Stimulus
2   for(int i = 0; i < 4; i++) begin
3     {X,Y} = i; #10;
4   end
5 end
```

We use the fact that integers are encoded in binary and therefore can be assigned to other variables. This code easily scales to an arbitrary number of signals of total width `N` – the only changes required are to replace the upper limit (*i.e.*, `4`) with 2^N (written as `2**N` in Verilog) and to change the concatenation to include all the input signals being tested.

Testbench Clock Simulator

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

```
logic clk;
parameter PERIOD = 100; // period = length of clock cycle
            //  arbitrary value, but should be much > 1
initial begin
  clk <= 0;
  forever #(PERIOD/2) clk = ~clk;
end
```

Many other code variants exist that can produce a valid simulated clock signal. The simulated clock would be put into the testbench for your system and all sequential modules would take `clk` as an input.

Waiting for a Signal

So far, testbench examples have used `#<number>` to delay instructions, but it is also possible to wait for a signal to change instead of waiting a specific amount of time. This can be useful for debugging modules which take multiple cycles to finish working. The syntax to wait for a signal to change is `@(posedge sig)` or `@(negedge sig)`, similar to the sensitivity list for an `always_ff` block. For example, the following stimulus block waits for the done signal to go high before changing the simulated inputs.

```
initial begin // Stimulus
  X <= 0; Y <= 0;
  @(posedge done) Y <= 1;
  @(posedge done) X <= 1; Y <= 0;
  @(posedge done) Y <= 1;
end
```

Printing Values to the Console

Most development will use simulated waveforms to examine the state of signals over time. However, sometimes in debugging it is useful to print messages as well (*e.g.*, any time an error condition is found, print a message about the error and the values of relevant variables). The `$display` command is one way to do this:

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

`$display` takes an arbitrary number of arguments which are concatenated and printed to the console. `$time` is a special function which returns the simulated time when it is called. The empty value between the `$time` call and the string simply adds extra space to make the output more readable.

`$display` prints once at the time specified. If you would prefer constant monitoring, use `$monitor`:

```
initial begin // response
 $monitor($time, , SEL, I, J, , V);
end
```

`$monitor` prints once at the time specified and then any time afterward when any of the signals being monitored changes.

Advanced Features – Multi-Dimensional Buses

Sometimes it can be useful to have signals with more than one dimension – for example, we might want to hold sixteen 8-bit values. Verilog allows you to define multiple sets of indices (*i.e.*, multiple packed dimensions):

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

To access a multi-dimensional packed variable, successive indices refer to the packed dimensions from left-to-right. For example, the following code sets all the bits of a 3-dimensional bus to `0`:

```
logic [15:0][9:0][7:0] vals;

initial begin
 for (int i = 0; i < 16; i++)
   for (int j = 0; j < 10; j++)
    for (int k = 0; k < 8; k++)
      vals[i][j][k] = 1'b0;
end
```

Unpacked Arrays

It is also possible to declare an array with the size *after* the variable name. This is called an **unpacked array**, because successive elements of an unpacked dimension *cannot* be treated as one unit. A packed array of 8 bits can be used as a byte, but an unpacked array of 8 bits can only be accessed one bit at a time:

```
logic [7:0] arr1;
logic arr2 [7:0];

my_module works (.in(arr1), .out); // correct
my_module fails (.in(arr2), .out); // incorrect
```

A mismatched array type and will give an error message about the unpacked array type, often something like: "*unpacked array type cannot be assigned to integer vector type - types do not match*"

It is possible to have a variable with both packed and unpacked dimensions. In this case, the unpacked dimensions are access first from left-to-right, followed by the packed dimensions from left-to-right.

Advanced Features – Assert Statements

As you design larger systems, you will often have assumptions you would like to verify are true. For example, you may have a parameterized module with a limited number of legal parameter values. 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. During simulation, it 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:

```
1 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:

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

Advanced Features – Generate Statements

Earlier in this tutorial the `for` and `if` constructs were introduced in the context of RTL code, which dealt with changing the values of existing signals. In order to put submodule calls and other logic within `for`-loops and `if`-statements, the `generate` statement must be used. It begins with `generate` and ends with `endgenerate`. Additionally, instead of using `int` variables, `genvar` variables must be used instead. All structures to be generated, including modules, `always_comb` blocks, `always_ff` blocks, and `assign` statements, should be placed inside all `for` and `if` statements.

Any `for`-loops or `if`-statements must have a `begin` block followed by a colon and a name, called the label. Each construct generated will be identifiable by this label. These names can help when inspecting error messages and will also appear when examining signals in ModelSim. The following example module creates a number of `D_FF` submodules based on the parameter `WIDTH`.

```
 1 module DFF_VAR #(parameter WIDTH=8) (q, d, clk);
 2  output logic [WIDTH-1:0] q;
 3  input logic [WIDTH-1:0] d;
 4  input logic clk;
 5
 6  initial assert(WIDTH>0);
 7
 8  genvar i;
 9  generate
10    for(i=0; i<WIDTH; i++) begin : eachDff
11     D_FF dff (.q(q[i]), .d(d[i]), .clk);
12    end
13  endgenerate
14 endmodule
```