Basic Verilog Tutorial

Mark L. Chang[1], last revised 10/03/2004

Introduction

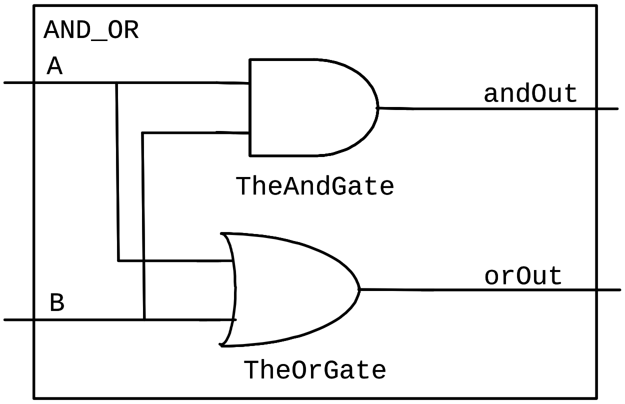
The following tutorial is intended to get you going quickly in gate-level circuit design in Verilog. It isn't a comprehensive guide to Verilog, but should contain everything you need to design circuits for your class.

If you have questions, or want to learn more about the language, I'd recommend Samir Palnitkar's *Verilog HDL: A Guide to Digital Design and Synthesis*. It is available on reserve at the library.

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 andOut, orOut;
  input A, B;

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

We can analyze this line by line:

---

[1] Material contributed by Scott Hauck, Akshay Sharma, and various faculty at the University of Washington.

```
// 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 andOut, orOut;
  input 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:

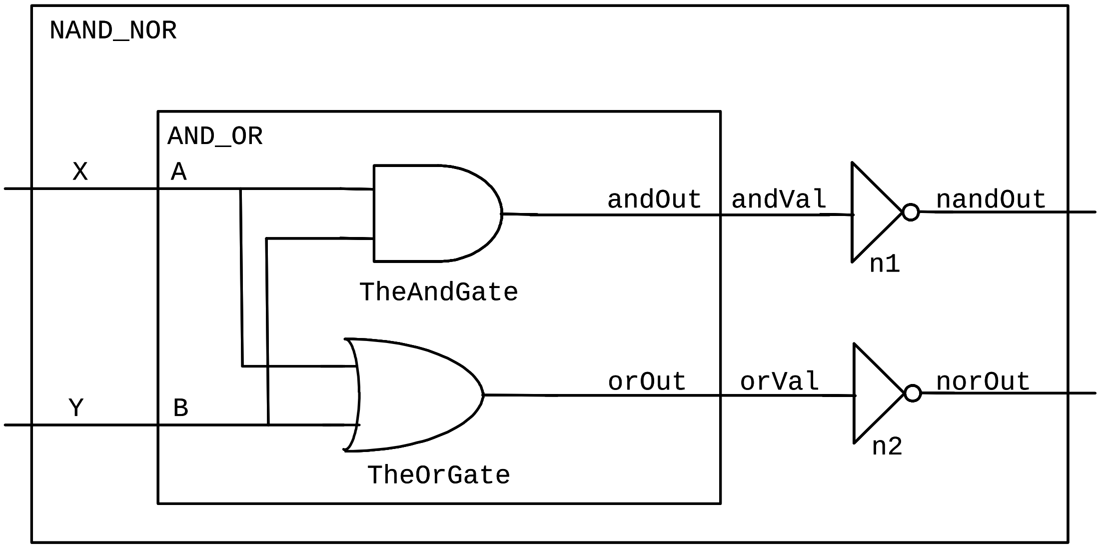
```
and <name> (OUT, IN1, IN2); // Sets output to AND of inputs
or <name> (OUT, IN1, IN2); // Sets output to OR of inputs
nand <name> (OUT, IN1, IN2); // Sets to NAND of inputs
nor <name> (OUT, IN1, IN2); // Sets output to NOR of inputs
xor <name> (OUT, IN1, IN2); // Sets output to XOR of inputs
xnor <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 andOut, orOut;
  input 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 nandOut, norOut;
  input X, Y;
  wire andVal, orVal;

  AND_OR aoSubmodule (andVal, orVal, X, 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 (andVal, orVal, X, Y);
```

Note that the connections to the sub-module work the same as parameters to C/C++/Java procedures. That is, the variable andVal in the NAND_NOR module is connected to the andOut output of the AND_OR module, while the X variable in the NAND_NOR module is connected to the A input of the AND_OR module. Note that every signal name in each module is distinct. That is, the same name can be used in different modules independently.

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 “wire” statement:

```
  wire 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.

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 (andVal, orVal, 0, foo);
```

This also means that if you need to have a module that always outputs true or false, we can do that with a buf gate:

```
// Always return TRUE.
module TRUE(Out);
  output Out;

  buf b1(Out, 1);
endmodule
```

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:

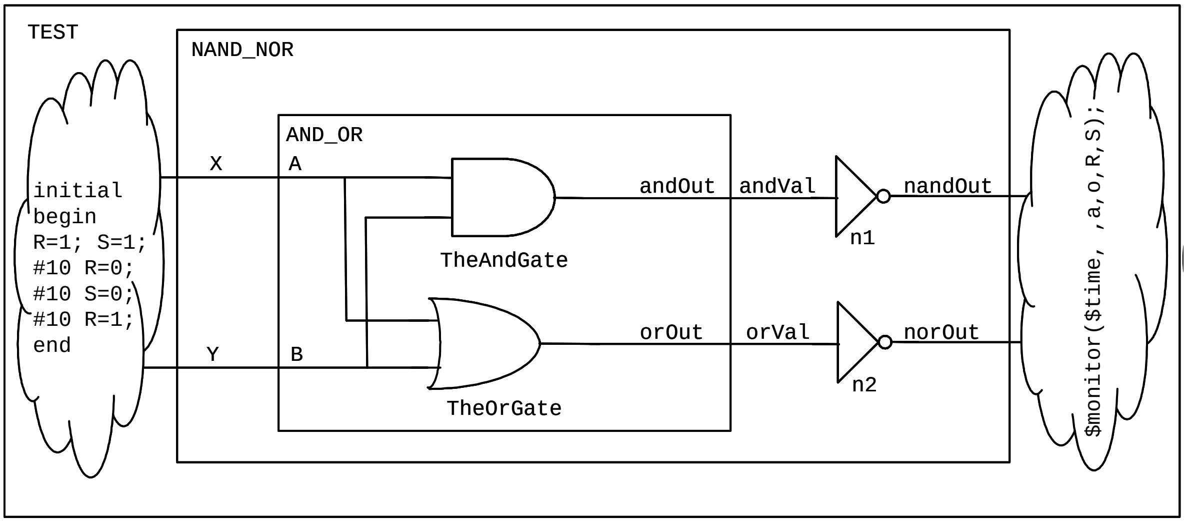
```
// Compute the logical AND and OR of inputs A and B.
module AND_OR(andOut, orOut, A, B);
  output andOut, orOut;
  input A, B;
  parameter delay = 5;

  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.

Testbeds

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 testbed. A testbed is a module that calls your unit under test (UUT) with the desired input patterns, and collects the results. For example consider the following:



```
initial
begin
R=1; S=1;
#10 R=0;
#10 S=0;
#10 R=1;
end
```

```
$monitor($time, ,a,o,R,S);
```

```
// 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 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 nandOut, norOut;
  input X, Y;
  wire andVal, orVal;

  AND_OR aoSubmodule (andVal, orVal, X, Y);
  not n1 (nandOut, andVal);
  not n2 (norOut, orVal);
endmodule

module TEST;  // No ports!  No-one else calls this unit
  reg R, S;
  wire a, o;

  initial  // Stimulus
  begin
    R = 1; S = 1;
    #10 R=0;
    #10 S=0;
    #10 R=1;
  end

  NAND_NOR UUT (a, o, R, S);

  initial  // Response
    $monitor($time, , a, o, , R, S);
endmodule
```

The code to notice is that of the module “TEST”. It instantiates one copy of the NAND_NOR gate, called “UUT”. All the inputs to the UUT are declared “reg”, while the outputs are declared “wire”. The inputs are declared “reg” so that they will have memory, remembering the last value assigned to them – this is important to allow us to use the “initial” block for stimulus.

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

```
  initial  // Stimulus
  begin
    R = 1; S = 1;
    #10 R=0;
    #10 S=0;
    #10 R=1;
  end
```

The code inside the “initial” statement is only executed once. It first sets R and S to true. Then, due to the “#10” the system waits 10 time units, keeping R and S at the assigned values. We then set R to false. Since S wasn’t changed, it remains at true. Again we wait 10 time units, and then we change S to false (R remains at false). If we consider the entire block, the inputs RS 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  // Stimulus
begin
  R = 0; S = 0;
  #10 S=1;
  #10 R=1; S=0;
  #10 S=1;
end
```

This goes through the pattern 00 -> 01 -> 10 -> 11.

In order to capture the results of the computation, we display to the user the information with the stimulus code:

```
initial  // Response
  $monitor($time, , a, o, , R, S);
```

This tells Verilog to show information about values a, o, R, and S whenever any of them change. It will print the time (in Verilog’s “time units”), a space, the values of a and o, a space, and the values of R and S. Any variables can be included, in any order, in the monitor statement. For the original code above, the resulting display would be:

```
 0 00 11
10 10 01
20 11 00
30 10 10
```

Time is the left column (advancing by 10 because that is when the inputs change), with the NAND and NOR being shown respectively, then R and S.

$monitor in more detail

Since $monitor will be one of your primary ways of getting information about your circuit, there are a couple things that will be important to know:

1. **X values:** sometimes a signal will display an “X” instead of 0 or 1. This is Verilog telling you “I can’t figure out what the value is”. Mostly this means you’re debugging a sequential circuit (a circuit with state) and you didn’t reset a stateholding element. It might also mean you hooked two gates to the same output (a BAD idea) and they’re saying different things. In general, this is a warning sign. “Z” can also be a problem, often meaning a signal isn’t hooked up to anything at all.
2. **Multiple $monitor printings for each input change:** $monitor outputs data every time a signal being monitored changes. If your circuit has delays in it, then for each input change there can be multiple $monitor outputs, since the signals being monitored change at slightly different times. For example, if I monitor the

input and output of a gate with 5 units of delay, the monitor will print twice – once for when the input changes, and once 5 units later when the output changes. Give lots of time between input changes, and always look at the last $monitor statement before the next input change – this is the one where the circuit has had long enough to stabilize at a new value.

3. **Debugging Hierarchical designs:** You will often design circuits with modules, sub-modules, sub-sub-modules, etc. Debugging them can be hard, because you only see the inputs and outputs of a cell, but really want to see what's going on inside the sub-modules. The good news – the $monitor statement can look inside submodules via the instance's name. For example, in the testbed example above we have a $monitor statement in the module TEST. It has a submodule named UUT, which has a submodule aoSubmodule. In the module TEST I could have had the following $monitor statement:
   ```
   $monitor($time, , a, UUT.X, UUT.aoSubmodule.orOut);
   ```
   Giving the name of a module instance and then a dot says to look inside that instance of the module, and get the variable specified after the dot. So "UUT.X" says look in the UUT instance of the NAND_NOR module, and monitor it's variable X. "UUT.aoSubmodule.orOut" says look in the UUT instance, and then inside it's aoSubmodule, for the aoSubmodule's orOut variable. This format lets you look at any signal anywhere within a design.
4. **Strings in $monitor:** If you want to display some text, you can also include that in a $monitor statement:
   ```
   $monitor(“The value of A is: “, a);
   ```

Multiple files

You can break your code into multiple files. To put them together, one file can include the contents of another file via the include statement:

```
`include "alu.v"
```

Note that there is no semicolon on this line.

Sequential Logic

You will likely build all of your sequential elements out of D flip-flops:

```
module D_FF (q, d, clk);
  output q;
  input d, clk;
  reg q; // Indicate that q is stateholding

  always @(posedge clk) // Hold value except at edge
    q = d;
endmodule
```

Most of this should be familiar. The new part is the "always @(posedge clk) q = d;" and the "reg" statement.

For sequential circuits, we want to have signals that remember their prior value until it is overwritten. We do this by the "reg" statement, which declares a variable that remembers the last value written to it – an implicit flip-flop. This came in handy before in making testbeds, since we usually want to set up the inputs to remember their last setting, until we change it. Here, the variable "q" in the D_FF module remembers the value written to it on the last important clock edge.

We capture the input with the "always @(posedge clk)", which says to only execute the following statement (q=d;) 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 @(negedge clk)".

Clocks

A sequential circuit will need a clock, supplied by the testbed. We can do that with the following code:

```
reg clk;
parameter period = 100; // 2*period = length of clock
                        // Make the clock LONG to test
initial clk = 0;
always #(period) clk = ~clk;
```

This code would be put into the testbed code for your system, and all modules that are sequential (are D_FFs, or contain D_FFs) will take the clock as an input.

Declaring Multi-bit Signals

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

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

These statements set up a set of individual wires, which can also be treated as a group. For example, the "wire [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 [31:0] bus1;
     input [19:0] bus2;
     wire 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 Constants

In testbeds 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:

```
reg [15:0] test;
initial begin // stimulus
     test = 12;
     #(10) test = ‘h1f;
     #(10) test = ‘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.

Monitoring Multi-bit Signals

Multi-bit constants can be put into $monitor statements just like any other values. They will be displayed in decimal by default. However, if you want to show them in another format, you can do the following:

```
$monitor($time, " Hex: %h Dec: %d Bin: %b", f, g, h);
```

This displays the time, and the value of variable f in Hexadecimal, g in Decimal, and h in Binary respectively. This is controlled by the string in the monitor statement. The string can contain any string of normal characters. Also, whenever a % appears, it is followed by the base (h, d, or b). It tells the $monitor statement to replace the %h, %d, or %b with the value of the next variable in the list following the string, formatted in the specified base. That is, the first % takes the first variable in the list after the string, the second % takes the second variable, etc.

You can display an actual % symbol by including %% in the string.

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

```
wire [31:0] foo;
initial foo[3:1] = ‘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 our D_FF unit defined above:

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

  D_FF d0(q[0], d[0], clk);
  D_FF d1(q[1], d[1], clk);
…
  D_FF d15(q[15], 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 [3:0] q;
  input [3:0] d, clk;
  reg q;

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

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

  D_FF4 d0(q[3:0], d[3:0], clk);
  D_FF4 d1(q[7:4], d[7:4], clk);
  D_FF4 d2(q[11:8], d[11:8], clk);
  D_FF4 d3(q[15:12], 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 DFF_16 we could do:

```
  Wire [15:0] data, result;
  D_FF16 d1(result, { data[7:0], data[15:8] });
```

Anything can go into the concatenation – constants, subsets, buses, single wires, etc.

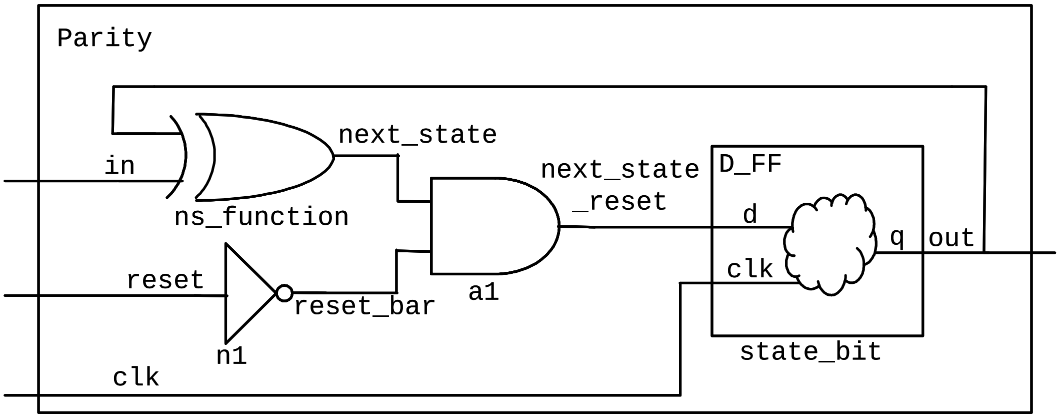
Example Finite State Machine

Here's an example of a simple sequential circuit, with all of its gory details. Two notes of interest, both in the stimulus portion:

1.) We delay changing inputs to the system to the negative clock edge, while the Flip-flops are positive edge triggered. This ensures you don't have a "race" condition – the ambiguity of whether the inputs change before or after the flip-flops do their thing.

2.) This circuit has a clock that runs on to infinity, so we need to explicitly tell Verilog when we are done. The $finish command tells it when we are done simulating things.

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 D_FF (q, d, clk);
  output q;
  input d, clk;
  reg q; // Indicate that q is stateholding

  always @(posedge clk) // Hold value except at clock edge
    q = d;
endmodule

module Parity (out, in, reset, clk);
  output out;
  input in, reset, clk;
  wire next_state, reset_bar, next_state_reset;

  D_FF state_bit (out, next_state_reset, clk);
  xor ns_function (next_state, in, out);
  not n1 (reset_bar, reset);
```

```
  and a1 (next_state_reset, next_state, reset_bar);
endmodule

module stimulus;
  reg clk, reset, data;
  wire value;
  parameter period = 100;

  Parity UUT(value, data, reset, clk);

  initial // Set up the clock
    clk = 0;
  always
    #(period) clk = ~clk; // Toggle clock every 100 units

  initial // Set up the inputs
  begin
    reset = 1; data = 0;
    @(negedge clk) reset = 0;
    @(negedge clk);
    @(negedge clk) data = 1;
    @(negedge clk) data = 0;
    @(negedge clk) data = 1;
    @(negedge clk);
    @(negedge clk);
    @(negedge clk);
    @(negedge clk) $finish; // end the simulation
  end

  initial // View the results
    $monitor($time, " clk: %b reset: %b data: %b value:
%b", clk, reset, data, value);

endmodule
```