Verilog Tutorial

Introduction

Verilog allows us to describe a system based on a structure of wires, gates, registers, and delays using a systematic language. This language is unlike most other programming languages, where they read like steps in a recipe. Instead, Verilog is written so that most components respond in parallel, simultaneously.

First Verilog Program

By using your favorite text processor, you can type in Verilog code to be run using Verilog and simulated in SignalScan. The simplest type of Verilog code is similar to that found in *listing 1*.

```
// Listing 1: Simple Verilog Code.
module top();
   wire out;
   reg a, b;

   assign out = a & b;

   initial
      begin
         a = 1'b0;
         b = 1'b0;
         #10;
         a = 1'b0;
         b = 1'b1;
         #10;
         a = 1'b1;
         b = 1'b1;
         #10;
         a = 1'b1;
         b = 1'b0;
         #10;
         $dumpflush;
      end

   initial
      begin
         $monitor("a=%b, b=%b, out=%b, time=%t\n", a, b, out, $time);
         $dumpfile("top.dump");
         $dumpvars(5, top);
      end
endmodule
```


Comments

Let's start by examining the code step by step. The first line is a comment defined by the two slashes. In any program, anything that follows two slashes is ignored by Verilog.

Modules

Modules are what define components in Verilog. They are remarkably similar to functions or procedures in other languages because given input, they can produce specific output. The module shown, *top*, has no input and output, making it self contained. It will be the first module evaluated when Verilog is run because of this.

A more general module follows the following form:

```
module modulename(in1, in2, ..., inout1, ..., out1, ...)
input in1, in2, ...;
inout inout1, ...;
output out1, ...;

   <module body or text>

endmodule
```

The *modulename* can be replaced with any name of the module. Although the inputs, input/outputs, and outputs can be placed in any order within the module's parameters, it is good to be consistent where you place them within the list. The keywords, *input*, *inout*, and *output* are used to define the direction that data can flow through the node named by the parameter. These keywords are used just like the *wire* and *reg* keywords which we'll discuss in more depth later. The *module body or text* is a list of expressions that define the system. Modules are always terminated by the keyword, *endmodule*.

Defining Wires and Registers

In this example, single bit wires and registers are used, although Verilog allows multi-bit sized wires (buses) and registers to be defined. Wires, registers, inputs, input/outputs, and outputs are all defined in the same fashion. The only difference being the keyword used.

```
input a, b;
output [2:0] out;
wire wire1, wire2, a, b;
wire [7:0] byte;
reg [31:0] wordreg;
```

A name by itself represents a single bit, or wire node. To produce buses, or larger arrays of bits, the bracketed expression is placed before the name. *[2:0]* defines 3 bits, 2 through 0. *[7:0]* defines 8 bits, 7 through 0. *[31:0]* defines 32 bits, 31 through 0.

Registers and wires cannot be used everywhere. In the simple program, *out* is a wire, and *a* and *b* are registers. This is required because *out* appears at the left hand side of an *assign* expression, and *a* and *b* appear at the left hand side of '=' signs in the *initial* expression. We'll cover this in more detail later.

The Assign (Declarative) Expression

The *assign* expression can have registers and wires as inputs into the expression on the right hand side of the '=' sign, but only wires on the left hand side of the '=' sign. This is due to the fact that the value at left is dependent on the values on the right at all times. This also means that a particular wire can only be found once on the left side of the '=' once with an *assign*. Having a wire assigned more than one value would be contradictory in nature.

```
assign out = a & b;
assign Cin[7:0] = {Cout[6:0], ~aluAdd};
assign Cout = Gen | (Prop & Cin);
```

In assign expressions, logic, and in some cases, arithmetic, can be used. The following are valid bitwise logic operators, /, &, ^, and ~, which are *or*, *and*, *xor*, and *not* respectively.

Note the use of *Cin[7:0]*. This refers to assigning a value to the bits 7 through 0 of *Cin*. If we only desired to set a smaller number bits of *Cin*, we could have wrote *Cin[5:3]* which would define bits 5 through 3 of *Cin*. The expression *{Cout[6:0], ~aluAdd}* defined by the braces is called a *Concatenation*. This allows multiple bits to be put together and treated like a string of bits. In the example above, *Cout[6:0]* is concatenated with the not of *aluAdd*, so *Cin[7:1]* gets *Cout[6:0]* and *Cin[0]* gets *~aluAdd.*

The Initial and Always (Procedural) Expressions

Initial and *always* are extremely similar to each other. The main difference is that *initial* defines events that only happen at the beginning of time, and *always* defines events that happen all the time a given condition occurs (if no condition is given, it defines a continuously looping list of events). In these expressions, only registers can be defined on the left hand side of the '=' sign. This happens because everything within an *always* or *initial* block takes the inputs as they are and sets the value. If the inputs change, it takes an '=' sign in an *always* or *initial* block to reset the register's value, or the value will not change. Compare this to the *assign* expression whose output will always change if one of its inputs change.

These are called procedural because they can be set to only happen at certain times or during particular conditions, whereas the *assign* defines something that happens continuously.

```
initial
   begin
      toggle = 1'b1;
   end

always @(a or b)
   begin
      out = a & b;
      toggle = ~toggle;
   end
```

In the above example, the *initial* block initializes the *toggle* register at binary 1.[1] The *always* block is only called when either *a* or *b* changes values. If the expression"*toggle =*

[1] I'll cover the number format later.

~toggle;" had not be added to the always block, it would essentially act like "*assign out = a & b*". However, now the value of *toggle* flips back and forth whenever *a* or *b* change their values.

Always expressions can also be written without any conditions as shown below.[2] In this case they need delays or it will be called all the time, even after it finishes being called (causing an infinite loop).

```
// 50% duty cycle clock, 20 time unit period
always
   begin
      clock = 0;
      #10;    // Delay control = 10 units
      clock = 1;
      #10;
   end
```

Numbers in Verilog

Numbers in Verilog follow the form shown in *figure 1*. The first number tells Verilog how many bits the number takes up. The character represents the base of the value representation, *h* for hexadecimal, *b* for binary, *d* for decimal, and *o* for octal.

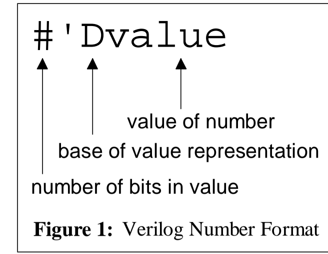

Figure 1: Verilog Number Format

```
16'h1A23
14'b01101000100011
20'd6691
```

All of the Verilog values shown above have the same decimal value, but each contains a different number of bits and is represented in a different base.

Wire & Procedural Delays in Verilog

The results of an *assign* can be delayed from output by a given number of time units. Although your logic should not be dependent on these delays, it may come in handy if you have feedback loops which would go unstable without some type of delay. The delay time appears after the '#' mark, and before the equality statement.

```
assign #3 out = a & b;
```

This statement would cause the value of *out* to output 3 units of time late.

Within *initial* or *always*, the delay can be used to separate commands by a specific amount of time. Each delay is evaluated in sequence and can be used to separates groups of assignments as shown in the simple Verilog code, and the 50% duty cycle clock shown above. If the delay is on a line by itself, be sure to place a semicolon at the end of the line, or it will be assumed that the next line is being delayed by the value. (If there is no next line, an error will occur.)

[2] This is a modified example taken from section 2.6.1 of *Verilog According to Tom.*

Monitoring Values

While testing Verilog code, you will need to make sure values change as you would expect. This can be done using the *$monitor* command in Verilog. This sets up a statement which prints out text every time a value changes, causing the previous output text to become invalid. The first parameter in *$monitor* is a string with text and symbols, like *%h*, *%b*, and *%t*, which refer to the following parameters, the monitored values.

```
$monitor("a=%b, b=%b, out=%b, time=%t\n", a, b, out, $time);
```

The *$time* variable contains the number of time units that have passed in the simulation. *%h* displays a hexadecimal number, *%b* displays a binary number, and *%t* is specially designed to display a time value.

The Dump File

To use the simulatior SignalScan (or the past SimWaves), a dump file must be generated. This is done by using the three commands, *$dumpfile*, *$dumpvars*, and *$dumpflush*. *$dumpfile* is used to indicate in what file to put the dump information.

```
$dumpfile("simple.dump");
```

$dumpvars is used to indicate what to dump. In extremely large simulations, dump files may become huge in extent. In these cases only special variables may want to be included in the dump file. The first number represents how many *modules* to descend into. The following are names of modules or values to be included in the dump file. In the simple case, the number was made arbitrarily large and the all inclusive *top* module was told to be dumped. This should dump everything into the dump file.

```
$dumpvars(5, top);
```

$dumpflush is used whenever new information needs to be forced into the dump file. Because Verilog quits after running the simple code, it probably is not necessary, but for the sake of example it has been added to the code.

Running Verilog Code

At the shell prompt, the Verilog code can be run by typing:

```
verilog file1.v file2.v ... fileN.v
```

Or, if the files, file1.v through fileN.v, are listed in a text file, each residing on a separate line, the Verilog code can be run by:

```
verilog -f filelist
```

The output of the simple code should look similar to:

```
Compiling source file "simple.v"
```



```
Highest level modules:
top

a=0, b=0, out=0, time=                   0

a=0, b=1, out=0, time=                  10

a=1, b=1, out=1, time=                  20

a=1, b=0, out=0, time=                  30

0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.1 secs to compile + 0.0 secs to link + 0.1 secs in
simulation
End of VERILOG-XL 2.5   Oct 27, 2000
10:43:28
```

Using SignalScan to View the Dump File

To use SignalScan, read the *Using SignalScan With Verilog* document.

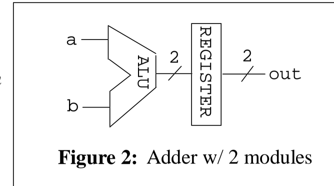

Figure 2: Adder w/ 2 modules

Example with Modules

More complex designs will have multiple modules. Each module acts like its own component, or black box, with particular inputs and outputs. Modules can be just like the *top* module shown above. As an example of a device with multiple modules, we will design an adder. We will have two modules, an ALU and a register.

```
// Listing 2: Adder Verilog Code.

// ALU module
module adder(a, b, sum);
   input a, b;
   output [1:0] sum;

   assign sum = {a & b, a ^ b};
endmodule

// Register module
module register(clk, in, out);
   input clk;
   input [1:0] in;
   output [1:0] out;

   reg out_reg;

   assign out = out_reg;

   always @(posedge clk)
      begin
         out_reg = in;
      end
```

```
endmodule

// Top module
module top();
   wire [1:0] out;
   wire [1:0] add_out;
   reg a, b, clk;

   adder adder1(a, b, add_out);
   register reg1(clk, add_out, out);

   always
      begin
         clk = 1'b0;
         #3;
         clk = 1'b1;
         #3;
      end

   initial
      begin
         a = 1'b0;
         b = 1'b0;
         #10;
         a = 1'b0;
         b = 1'b0;
         #10;
         a = 1'b0;
         b = 1'b1;
         #10;
         a = 1'b1;
         b = 1'b1;
         #10;
         a = 1'b1;
         b = 1'b0;
         #10;
         $dumpflush;
         $finish;
      end

   initial
      begin
         $monitor("a=%b, b=%b, out=%b, time=%t\n", a, b, out, $time);
         $dumpfile("top.dump");
         $dumpvars(5, top);
      end
endmodule
```


The ALU Module

The ALU module is essentially an assign statement which preforms a simple carry adder arithmetic. Notice the use of *[1:0]* to define a two bit output, and the braces to define a concatenation.

```
// ALU module
module adder(a, b, sum);
   input a, b;
   output [1:0] sum;

   assign sum = {a & b, a ^ b};
endmodule
```

The Register Module

This defines a 2-bit register. The input and output are considered wires, so an assignment is made for the *out* wire to be equal to the output of the register, *out_reg*. This particular *always* block is only called when the value of *clk* transitions from low to high, the positive edge of *clk*.

```
// Register module
module register(clk, in, out);
   input clk;
   input [1:0] in;
   output [1:0] out;

   reg out_reg;

   assign out = out_reg;

   always @(posedge clk)
      begin
         out_reg = in;
      end
endmodule
```

The Top Module : Defining Modules

The definitions of the ALU and register modules are incomplete because they don't have any specific inputs and outputs. These definitions are primarily blue prints for making components which look and act as the blue print specifies. In the *top* module, the definitions are finished by giving them a component name and attaching wires or registers to them.

```
adder adder1(a, b, add_out);
register reg1(clk, add_out, out);
```

The first name is the blue print name. The second name is the realized component name, and the values within the parenthesis are the connected wires or registers.

```
blueprint component(wire1, wire2, ..., wireN);
```

The Top Module : Always Block

The *always* block in the top module is used to generate a clock to test the functionality of the register. Ever 3 time units, it transitions to a new value. Because it doesn't have a condition when the *always* is to begin, it simple makes an infinite loop.

```
always
   begin
      clk = 1'b0;
      #3;
      clk = 1'b1;
      #3;
   end
```

The Top Module : $finish

At the end of the *initial* block with the test inputs, *$finish* is added to tell Verilog the simulation is completed. Verilog would otherwise continue simulating because the clock continues to transition indefinitely. In order to create a Verilog prompt for testing, *$stop*, can be used instead.

The Adder Output

The output of the adder should look as shown below. Notice how the register causes the output to only transition at the positive edge of *clk*.

```
Compiling source file "adder.v"
Highest level modules:
top

a=0, b=0, out=xx, time=                   0

a=0, b=0, out=00, time=                   3

a=0, b=1, out=00, time=                  20

a=0, b=1, out=01, time=                  21

a=1, b=1, out=01, time=                  30

a=1, b=1, out=00, time=                  33

a=1, b=0, out=00, time=                  40

a=1, b=0, out=01, time=                  45

L63 "adder.v": $finish at simulation time 50
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.1 secs to compile + 0.0 secs to link + 0.1 secs in
simulation
End of VERILOG-XL 2.5   Oct 27, 2000  11:29:10
```