EE382N Verilog Manual

Y. N. Patt
H. Kim, M. Qureshi TAs
Department of Electrical and Computer Engineering
The University of Texas at Austin

Spring, 2004

Table of Contents

1. Introduction

This semester, you will design your project using Verilog HDL, a hardware description language commonly used in industry. While behavioral Verilog can be used to describe designs at a high level of abstraction, you will design your processor at the gate level in order to quantify the complexity and timing requirements of your design. Hence you will use structural Verilog only.

You will be provided Verilog libraries containing modules that will be the basic building blocks for your design. These library parts include simple logic gates, registers, and memory modules, for example. While the library parts are designed behaviorally, they incorporate some timing information that will be used in your simulations. Using the class libraries ensures a uniform timing standard for everyone.

Structural Verilog allows designers to describe a digital system as a hierarchical interconnection of modules. The Verilog code for your project will consist only of module definitions and their instances, although you

may decide to use some behavioral Verilog for debugging purposes. This manual will cover all aspects of the Verilog language that you will need to be familiar with.

2. Syntax

Verilog uses a C-like syntax. It is case sensitive, and all keywords are in lower case letters. Declarations, assignments, and statements end with a semicolon. It also uses C-style comments:

```
// comment to end of line
/* closed comment */
```

Numbers are represented as `<number_of_bits>'<base><number>`, where *base* can be `b`, `o`, `d`, or `h`, for binary, octal, decimal, or hex, respectively. Some examples:

```
8'hFF     //8-bit hex number
5'b101    //5-bit binary number 00101
1         //decimal number 1.  (decimal is the default base)
3'o5      //3-bit octal number 5
```

3. Data Types

The data types that you will use are `reg` (register) and `wire`. `Wires` cannot hold a value; they are used to connect modules of combinational logic. `Regs` are used to store values. Since `regs` keep state, a `reg` cannot be the output of combinational logic.

Note: The reg datatype is NOT what you use to make flip-flops and registers. When implemented at the structural level, flip-flops and registers are still made out of wires and logic gates or other modules, as you will see in the d-latch example in section 4. Regs are only used in behavioral Verilog.

You can declare regs and wires as follows:

```
reg a,b;              //two scalar registers
reg [0:7] Byte;       //8-bit vector, bit 0 is MSB
wire [31:0] Word;     //32-bit vector, bit 31 is MSB
```

You can reference a subset of bits in a vector as `Word[0]` or `Byte[3:0]`, for example.

You can use braces for concatenating two or more signals, like this:

`{Word[7:0], Word[31:8]}` is the 32-bit vector obtained by rotating `Word` to the right by one byte.

`{24'b0, Word[7:0]}` is the 32-bit vector obtained by zero extending the first byte of `Word`.

You can specify repetition by putting a number before the opening brace, like this:

`4{Byte[0]}` is the same thing as `{Byte[0], Byte[0], Byte[0], Byte[0]}`.

A 2-dimensional memory of type `reg` can be declared as follows:

```
reg [7:0] memory [0:1023];
```

You can reference bytes of memory as `memory[5]`, for example. If you want to reference an individual bit, you must first copy the byte into another variable.

We are using 4-value logic; that is, each bit can take on one of four possible values:

`0`: logic 0
`1`: logic 1
`z`: high impedance (for tri-state driver output)
`x`: unknown or undefined.

4. Module definitions and instances

The **module** is the basic logic entity in Verilog.

A **module definition** is delimited by the keywords `module` and `endmodule`, as shown in the example below.

The **module header** consists of the `module` keyword, the name of the module, and the port list in parenthesis, followed by a semicolon:

```
module d_latch (d, q, qbar, wen);
```

Following the module header are the **port declarations**. Ports may be of type `input`, `output`, or `inout`, for input, output, or bidirectional ports. Ports may be either scalar or vector. The port names in the declarations do not have to occur in the same order as they did in the port list.

The body of the module (consisting of `wire` and `reg` declarations and module instances) follow the port declarations.
Here is an example of a module definition for a gated D-latch:

```
//-----------------------------------------------------

// GATED D LATCH

//-----------------------------------------------------

module d_latch (d, q, qbar, wen);
   input d, wen;
   output q, qbar;

   wire dbar, r, s;

   inv1$ inv1 (dbar, d);
   nand2$ nand1 (s, d, wen);
   nand2$ nand2 (r, dbar, wen);

   nand2$ nand3 (q, s, qbar);
   nand2$ nand4 (qbar, r, q);

endmodule
```

In this example, there are 5 **module instances**. `inv1$` and `nand2$` (1-input inverter and 2-input nand gate) are modules that are defined in the class libraries. For both of these gates, the first port is the output of the gate, and the remaining ports are inputs.

A module instance consists of the module name followed by an instance identifier , port list, and semicolon. Several instantiations of the same type of module can be made in a list like this:

```
   nand2$ nand1 (s, d, wen),
          nand2 (r, dbar, wen),

          nand3 (q, s, qbar),
```

```
        nand4 (qbar, r, q);
```

When instantiating a module, you can omit ports from the port list. For example, if we wanted to instantiate `d_latch` in another module but did not care about the `qbar` output, we could instantiate it as follows:

```
    d_latch latch1 (din, q_out, , wen);
```

In this example, `din` and `wen` could be either wires, registers, or ports declared in the higher-level module, but `q_out` cannot be a register.

You should know that any undeclared identifiers are implicitly declared as scalar wires. So watch out for typos in your port lists.

5. Continuous assignments (for wires)

`assign` can be used to tie two wires together or to continually assign a *behavioral* expression to a wire. Examples:

```
wire a, b, c;

assign a = b;
assign c = ~b;  //c is assigned to NOT b
```

With a continuous assignment, whenever the value of a variable on the right-hand side changes, the expression is re-evaluated and the value of the left-hand side is updated. The left-hand side of the assignment *must* be a wire.

6. Procedural Assignments (for regs)

6.1 Initial

You will use registers primarily in testing your code. Unlike wires, regs will hold a value until it is re-assigned. In other words, it maintains state. Delays, indicated by a number following a pound sign, can be used to specify an amount of time before the value held by the `reg` changes. Here is an example.

```
reg [3:0] A;
initial
    begin

    A = 4'hf;
    #10         //delay 10 time units
    A = 0;
    #10         //delay 10 time units
    A = 4'h5;

    end  //end initial description
```

This example makes procedural assignments to register A. Any code in a begin-end block following the `initial` keyword executes only once, starting at the beginning (time 0) of a simulation. The first assignment to register A is made at time 0. The `#10` indicates a delay of 10 time units before the next assignments are made. `A` is set to 0 at time 10, and `A` is set to 5 at time 20.

An `initial` description may be used without a block delimited by `begin` and `end` if there is only one statement (see example in next section).

Note that your code can have multiple procedural assignments. If you have more than one `initial` block in your code, they will be executed concurrently.

6.2 Always

`Always` descriptions will execute repeatedly throughout a simulation. To initialize and cycle a clock at 10 time units, you could do the following:

```
reg clk;

initial clk = 1'b0;

always #5 clk = ~clk;  //clock cycle is 10 ns
```

6.3 Repeat

The `repeat` statement can be used to specify how many times a begin-end block is executed. The block following the repeat statement is executed the number of times indicated by the expression in parenthesis following the `block` keyword. In the following example, the block is executed four times. Note that I added a time delay after the last statement in the block to avoid a race condition.

```
reg [3:0] A;
initial
begin

    repeat(4)
    begin
        A = 4'hf;
        #10        //delay 10 time units
        A = 0;
        #10        //delay 10 time units
        A = 4'h5;
        #10        //delay 10 time units
    end  //end repeat(4)

end //initial description
```

7. Compiler Directives

You can use an accent grave (`) followed by a keyword to control compilation and simulation. The compiler directive is *not* followed by a semicolon.

7.1 `define

The `define` keyword can be used for making text macros. During compilation, when the macro name is preceded by an accent grave, the compiler substitutes the macro text. Here are a couple of examples:

```
reg clk;
reg [1:0] a;

`define half_cycle 5
```

```
`define set_a #10 a = 2'b

always #(`half_cycle) clk = ~clk;

initial
    begin

    clk = 1'b0;
    `set_a 00;
    `set_a 01;
    `set_a 10;
    `set_a 11;

    end    //initial begin
```

In the example above, the value of register `a` is changed every clock cycle.

7.2 `` `include ``

This is used to include more source files.

```
`include "file.v"
```

8. System Tasks and Functions

System tasks are commands built into the simulator. The system task calls end with a semicolon. You may find some of these useful:

8.1 $finish

Use `$finish` to end the simulation after a specified amount of time. Example:

```
initial #1000 $finish;
```

Stops the simulation after 1000 time units. Unless you are stepping through the simulation interactively using VirSim, you will need to use `$finish`.

8.2 $time

Returns the current simulation time.

8.3 $strobe

Use `$strobe` to display a message on your screen. The format is similar to that of `printf` in C. Note that if the `$strobe` command is called at the same simulation time that other registers or wires are changing values, the value changes will happen *before* the strobe command is executed. Example:

```
always @(posedge clk)
    $strobe ("at time %0d, a = %b", $time, a);
```

This statement is executed every time `clk` transitions from 0 to 1, printing the simulation time in decimal and the value of `a` in binary. If `a` changed right at the clock edge, the new value would be printed.

8.4 $readmemb and $readmemh

These are used for reading in data from a file of binary or hex numbers to initialize a memory module. You can use it like this:

```
initial
begin
    $readmemb("filename",instance_name.mem,
                          starting_address);
end
```

starting_address is the address of the memory module where you begin initialization. If *starting_address* is greater than zero, the memory locations at the beginning will remain unitialized (logic value `x`). The data file is just a text file that contains a list of binary or hex numbers. An example of a data file will be given later in the semester.

8.5 Dumping waveforms *(for creating a waveform file in text format using VCS)*

`$dumpfile` is used to specify the name of the waveform file.

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

`$dumpvars` specifies the wires and registers whose values you want to record. The syntax is

```
    $dumpvars (number_of_levels, instance_name);
```

If *number_of_levels* is 1, it will dump all the signals instantiated in the top level of the instance specified. If *number_of_levels* is greater than 1, it will dump all signals instantiated *number_of_levels* levels hierarchically below the specified instance. If *number_of_levels* is 0, it will dump ALL signals instantiated under *instance_name* at any level. In the examples below, suppose you have an instance called `reg_file` in an instance called `data_path` in an instance called `exec_core` in your TOP module.

```
$dumpvars (0, TOP.exec_core.data_path.reg_file);
$dumpvars (2, TOP.exec_core);
```

The first statement will dump all wires and registers instantiated at any level under the module called `reg_file`.
The second statement will dump all wires and registers instantiated in the top level of `exec_core`, and the top level of `data_path`, or any other modules instantiated in `exec_core`.

9. Timing information

You will need to examine the library files to figure out the timing delays of the library modules. Below is an example of a module from the library. The delays between the rise or fall of any input and the change of the output are specified. Two sets of three numbers are specified for each input. The first set are the minimum, typical, and maximum delays when the output transitions from 0 to 1. The second set are the delays when the output transitions from 1 to 0. For this module, the delays are the same regardless of which way the inputs are changing.

In this class we will always use the typical delays, so you need only use the middle number (0.2 in this case for both rise and fall times) when making your critical path and cycle time calculations.

```
module  nand2$(out, in0, in1);
        input   in0, in1;
        output  out;

        nand (out, in0, in1);

        specify
            (in0 *> out) = (0.18:0.2:0.22, 0.18:0.2:0.22);
            (in1 *> out) = (0.18:0.2:0.22, 0.18:0.2:0.22);
        endspecify
endmodule
```

10. A complete TOP module

Here is a small example of a complete program illustrating use of the d-latch. You can read the EE382N Tools Tutorial to learn how to simulate this. When simulated, it will dump a waveform file of all signals instantiated under TOP. It will also print any transitions of the `d` register to standard output.

11. Structural vs. Behavioral Verilog

To clarify the difference between structural and behavioral verilog:
Structural verilog is composed of module instances and their interconnections (by wires) only. The use of regs, explicit time delays, arithmetic expressions, procedural assignments, or other verilog control flow structures are considered behavioral verilog.

As stated earlier, your project code will consist primarily of structural verilog. *You will use behavioral statements for debugging purposes only.* In fact, you will probably only instantiate two `regs` in your whole design: one for the clock and one for a RESET signal that is asserted at the beginning of your simulation.

This section has described all of the Verilog functionality you will need for your final project. If you want more information on behavioral Verilog, try reading the Bucknell CSCI Verilog Manual or the verilog manual at The University of Edinburgh.

MDB created Spring 2000
updated 20-Jan-2004