Verilog Troubleshooting and Tips

CS-173 Fundamentals of Digital Systems

27th April 2025

Mirjana Stojilović
Parallel Systems Architecture Lab (PARSA)

Introduction

Here is a small guide designed to help you better understand all the relevant aspects of Verilog for this course. The first section also includes some general tips. If something should be added, feel free to mention it on Ed Discussion or in the exercise sessions, and we will consider updating this document. Foremost, even if this document is reread before publication, please consider that there may be some errors. If you find them, please report them on Ed Discussion so we can address them quickly.

This document is not meant to replace the Verilog coding guideline you can find on Moodle. This document is a general extension that aims to help you with related tools, such as the terminal and debugging.

1 Terminal Tips and Tricks

1.1 General Commands

Here is a nonexhaustive list of useful commands to know (or to keep under your reach when using any terminal):

- `ls` (list): lists all files and directories in the current directory;
- `cd` (change directory): helps you navigate through the terminal;
- `./`: executes the binary file given directly after.

Here is a small example: Inside the `/folder` directory of our computer, we have a folder named `/Folder1` and a file named `file1`. Inside the `Folder` we have two files `file2` and `/file3`. Imagine that we want to execute the binary file `file2`. We would do the following:

```
/folder > ls
Folder1 file1

/folder > cd Folder1

/folder/Folder1 > ls
file1 file2

/folder/Folder1 > ./file2
(Will execute the code of the file)
```

Now, a useful command if you fall in the wrong directory, don't worry, you don't have to restart your terminal: use the .. (this means: go back to the last directory visited, the parent directory). For example, with the same directories and files as above:

```
/folder > ls
Folder1 file1

/folder > cd Folder1

/folder/Folder1 > ls
file1 file2

/folder/Folder1 > cd ..

/folder > ls
Folder1 file1
```

As you can see, we go from `/folder` into `/Folder1` and then back from `/Folder1` to `/folder`! A general remark is, as you may have noticed, you can see the full path of where you are inside the terminal at the beginning of the command line.

A last command that can prove helpful is the `rm` (remove). This command removes a file (or a folder when you add `-r`). But be aware that this is a definitive choice, that means that once you have removed something with the `rm` or `rm -r` command, the item deleted is unrecoverable. Special note to the `-f` flag (that stands for "force") that you can use with the `rm`, you should not use it unless you are absolutely sure what you are doing and why.

1.2 Verilog Commands

As you have probably seen, we often use the same set of commands in the terminal. Here they are listed and explained in detail (for trouble while running them, please take a look at the next section):

- `iverilog`: compiles a Verilog file;
- `gtkwave`: runs a .vcd file to get the graphical interface of gtkwave;
- `./ (for mac and linux)`: executes the file given directly after (the output produced by the iverilog command below);
- `vvp (for windows)`: executes the file given directly after (the output produced by the iverilog command below).

```
iverilog -o OutputName module.v tb.v
```

The command can be decomposed as follows:

- `-o OutputName`: creates the file named `OutputName` which will be the result of the compilation;
- `module.v`: the name of the module file that you want to compile;
- `tb.v`: the name of the testbench file associated with the file `module.v` that you are compiling.

For example, imagine that we have a module named `mymodule.v` and a testbench `mytestbench.v` in the directory `folder`.

Demo for Mac and Linux users

```
/folder > ls
mymodule.v mytestbench.v

/folder > iverilog -o NewCreatedFile mymodule.v mytestbench.v
/folder > ls
mymodule.v mytestbench.v NewCreatedFile

/folder > ./NewCreatedFile
(Will execute the code of the file)

/folder > ls
mymodule.v mytestbench.v NewCreatedFile theFileForgtkwave.vcd

/folder > gtkwave theFileForgtkwave.vcd
(Will open gtkwave with the content of the file)
```

Here, you can see that you don't need to add an extension name to the file you create; your computer will do it automatically.

Demo for Windows users

```
/folder > ls
mymodule.v mytestbench.v

/folder > iverilog -o NewCreatedFile.vvp mymodule.v mytestbench.v
/folder > ls
mymodule.v mytestbench.v NewCreatedFile

/folder > vvp NewCreatedFile.vvp
(Will execute the code of the file)

/folder > ls
mymodule.v mytestbench.v NewCreatedFile theFileForgtkwave.vcd

/folder > gtkwave theFileForgtkwave.vcd
(Will open gtkwave with the content of the file)
```

Here, you have to note that to run, you have to create a file with the extension `.vvp` and then use the command `vvp` to run it.

Note: With those examples, you can see that the .vcd file for gtkwave was created only after the execution of the file `NewCreatedFile` (or `NewCreatedFile.vvp`).

Note: If you ever need to compile more than one file at a time (for example, with the ripple carry adder), you can add multiple files after the output name of the file you want to create.

After reading this, you should be ready to move on to the next section, where we will discuss a terrible subject… compiling issues!

2 Compiling Issues

Here comes the most painful yet helpful section of this guide, which will help you master compiling issues.

As a refresher, here is the complete version of the command to use: `iverilog -o outputname module.v tb.v`

- *No such file or directory / No top-level modules, and no -s option*
  - Try to save all files (the module and the testbench).
  - Check that you are in the right directory and that all files are present (with the `ls` command).
  - Check if your command is not missing some part, check if you have the `-o` option, the name of the file that you want to create, and then the module and the testbench (the correct command is above).
- *module.v:XX: syntax error*
  - For specific syntax errors, refer to the next section.
  - If your line XX seems correct, check the line above if you are not missing a semicolon (;).
- *These modules were missing: moduleName referenced 1 times.*
  - Check the module name you want to instantiate (does it correspond to the real name).
  - Check that the module you give in the command is used in the testbench.

3 Verilog Global Syntax

3.1 Boolean Logic

These examples will be straightforward, so we will not detail them much. Based on those simple blocks, you can easily make more complicated ones (think of implementing a NOR, a NAND, and other more complex ones).

3.1.1 And, Or, and Xor

```
a & b;
a | b;
a ^ b;
```

3.2 If, Else If, Else

See the example below. Note that as the instructions are only one line long, you can omit the `begin` and `end` tags, which makes it much easier to read; this is also referenced in the last section. Also, you have to use conditional statements only in the `always` block.

```
module mymodule (
    input a,
    input b,
    output reg [1:0] out
);

    always @* begin
        if (a & b) begin
            out = 2'b01;
        end else if (a | b) begin
            out = 2'b10;
        end else begin
            out = 2'b00;
        end
    end
endmodule
```

3.3 Case Switch

Note that here the `case` syntax can also be used only inside an always block.

```
module mymodule (
    input [2:0] in,
    output reg [2:0] out
);

    always @* begin
        case (in)
            valueIn0: out = valueForOutput0;
            valueIn1: out = valueForOutput1;
            valueIn2: out = valueForOutput2;
            valueIn3: out = valueForOutput3;
            default:  out = defaultOutputValue;
        endcase
    end

endmodule
```

3.4 Always Blocks

As you know, an `always` block will be triggered when an input changes, but how can we tell which inputs can trigger the execution of the `always` block? Before diving into it, you must know that the `@`, also called a sensitivity list, is used to specify precisely that.

3.4.1 always @* / always@(*)

The star * means every input. So whenever you have input that changes, the block will always be evaluated! For instance, in the example below, as long as `a` or `b` or `c` (or multiple of them) changes, the always block will be evaluated:

```
module mymodule(
   input a, b, c,
   output reg [2:0] out
);

    always @* begin
        out <= a + b + c + 1;
    end

endmodule
```

3.4.2 always @(posedge clk)

A syntax that you will soon love, in which the always block will be triggered only when the input (named `clk` here as a reference to the clock) goes from 0 to 1 (i.e., at each rising edge of the clock). For instance, the following program will count the clock's rising edges.

```
module mymodule (
    input clk,
    input rst,
    output reg [4:0] out
);

    always @(posedge clk) begin
        if (rst)
            out <= 0;
        else
            out <= out + 1;
    end
endmodule
```

3.4.3 always@(negedge clk)

After reading the previous section, guessing what appears here may be very easy: it's the opposite. Here, the `negedge` means 'negative edge', so it will count the number of times the clock will go from 1 to 0. So, for example, if we consider the following program:

```
module mymodule (
    input clk,
    input rst,
    output reg [4:0] out
);

    always @(negedge clk) begin
        if (rst)
            out <= 0;
        else
            out <= out + 1;
    end
endmodule
```

Even if it is almost the same program as the one in the previous section, it fulfills a very different role!

3.4.4 Reg or Not Reg ?

The answer is simple: Will the variable be modified in an `always` block? If the answer is yes, it must be declared a `reg`; otherwise, it shouldn't be.

With all those syntaxes, you have everything you need to move on to the next section, which will advise you to avoid trouble while coding in Verilog.

4 Verilog Coding Tips to Avoid Issues

This section is more of a recap of what has been said in the lectures. We tried to keep track of the most recurrent issues and errors that students make during exercise sessions to provide you with some help on more specific things. **Let us reemphasize that the goal of this is not to replace the PDF available on Moodle but rather to complement it, you should therefore read it carefully.**

4.1 Begin and End

As you always put initial and closing parentheses in Java, remember always to put your `begin` and `end` tags to avoid unnecessary problems. This is relatively easy to do as Verilog uses a lot of color to help you. Also, as mentioned above, if you have one-liner execution (for example, in an `if`), the program may be easier to read if you omit the begin and end tags (this is a supported syntax, like in most programming languages).

4.2 Always Block

You have three good practices to adopt in an `always` block that describes a combinational circuit: (1) always use blocking assignments, (2) use * so that always block reacts to changes to any of the inputs, and (3) give variables a default value at the beginning of the `always` block (if this is possible, of course).

Also, if you were to code different `always` blocks for some reason (and that can happen), make sure that the blocks are independent, which means that you don't modify the same variable at two distinct places simultaneously.

4.3 Case Statement

As simple as this can sound, always set your `case` statement to a default value (something you should also do in Java). This way, the program won't break for bad input, and finding out what goes wrong while debugging will be easier.

4.4 Module Name

Verilog doesn't have strict naming conventions, but we suggest always naming your module the same way you name your file. This can help avoid unnecessary debugging.

4.5 Testbenches

4.5.1 Unknown Module Name

A standard error while writing test benches is when the name of the module you instantiate is underlined in red. This can happen even if your code is correct. One way to avoid this is to ignore it, as it doesn't cause any compiling issues or bugs in the behavior of your program. Another, better idea is to add the following line at the top of your testbench:

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

If the module name is still declared as an error even with this, you have a fundamental error and won't be able to compile. An easy way to solve this problem is to check if all your modules have the same name as their respective files (as mentioned in the previous point).

4.5.2 Module Instantiation

In Verilog, module instantiation is quite easy. It is done in the way shown in the module testbench below.

```
module testbench;
 reg  inputFromTestBench1;
 reg  inputFromTestBench2;
 wire outputFromTestBench;

 module_name TheNameThatYouWant(
   .inputOfTheModule1(inputFromTestBench1),
   .inputOfTheModule2(inputFromTestBench2),
   .outputFromTheModule(outputFromTestBench));

 initial begin
    // test your module in here
 end

endmodule
```

Let's slow down a little bit and analyze this in detail. First, the variables we will give to the program are declared as `reg` because we will modify them into an initial block, whereas the output is declared as a `wire`. After all, it will only store the output value of the program. Note: In testbenches, we use `initial` and not `always` because `always` may not terminate, as opposed to `initial`, which will be called only once.

- `inputFromTestBench1` and `inputFromTestBench2` are the variables you will modify in your testbench and feed to your module as input.
- `outputFromTestBench` is the output you will link to the appropriate port of your module to get back the output to check its validity.
- `module_name` is the name of the module you want to instantiate.
- `TheNameThatYouWant` is the name you want to give to your module (as suggested here, the name doesn't matter, you can put whatever you want here).

Now, for the most complicated part, how do we link the values of the testbench that we will use to the actual module to test? If you think a bit about it, you should be able to figure out the syntax from the above example. You first use a "." to indicate that you want to access a specific

port of your module, then you put the port's name, and after that, between the parentheses, you put the actual variable of your testbench.

4.5.3 $ Tags

In the testbenches already provided, you may have noticed some $ followed by some keywords. But what do they do exactly?

Display tag: $display. The display tag is used to print a line in the terminal. For instance, the following line will print out in the terminal "The value of i = 2 and the value of j = 3":

```
$display("The value of i = %d and the value of j = %d", 2, 3);
```

As you might have noticed, you can put as many `%d` as you want; for each one, the compiler will look up the corresponding variable after the end of the string to print. For more info on the string format specifiers, such as `%d`, visit the following page: https://www.chipverify.com/verilog/verilog-display-tasks.

VCD-related Tags: $dumpfile and $dumpvars. We decided to put both in the same part because they are closely related: with both, you create the .vcd file to visualize with gtkwave. For example:

```
$dumpfile("nameAsYouWish.vcd");
$dumpvars;
```

This will create a file named "nameAsYouWish.vc" that you will be able to give to gtkwave for the visualization.

$finish tag is there to help you see if the program executes correctly, as it will print out a line like the following to let you know that the program executed (this will help you avoid searching for a bug that doesn't exist in your module):

```
yourTB.v:XX: $finish called at 5410 (1s)
```

Note that the finish time may vary. The general idea is to have this line printed in the terminal to confirm that everything is executed correctly.

4.5.4 Time Waiting with a Clock #XX

Something to consider when writing a testbench is the waiting time we should give to our circuit to compute what it has to calculate correctly. The rule is quite simple: when you switch your clock up and down periodically, each n time units, you have to make your program wait for $2n$ time units before changing the input. Why is that? To change the input at each complete clock cycle.

In the `tb` module below, you can see that for each 5 time units, the clock will switch state, and after assigning the value that we want to test to the input, we make it wait 10 time units.

Another interesting thing is that before looping, we assign a default value to all inputs (here only to the clock, as it is the only one), and we make the program wait for a full clock cycle. This is done to avoid undefined states.

```
module tb;

    reg clk;
    wire [4:0] out;
    integer expected;

    periodic_counter nom (.clk(clk), .out(out));

    initial begin
        clk = 0;
        #10;
        for (integer i = 0; i < 540; i = i + 1) begin
            expected = i % 32;
            #10;
            if (out != expected)
                $display("%d, error at time ", i);
        end
        $finish;
    end

    always begin
        #5 clk <= ~clk;
    end

endmodule
```

4.5.5 Error Code

When running your program, you may have an unwanted result if your program is not well implemented:

- `U (unknown)`: Your wire or reg is not connected.
- `X (error)`: a wire that may be connected to multiple outputs (modifying the same reg inside two different blocks).

4.5.6 i++ or ++i or i+=1 ?

Spoiler alert: None of them! In Verilog, the only valid syntax is i = i + 1, so be careful when using it in a loop. This is a basic thing to know, but it may save you some trouble, so always write i = i + 1 when you need to increment something.

5 Conclusion

Here we come to the end of this document. We hope it is helpful to you and that you have learned something interesting or confirmed what you were thinking. However, if you have any persistent doubts, please get in touch with an assistant through Ed Discussion or in an exercise session. Again, if something should be added, don't hesitate to discuss it with us!

Acknowledgments

Authors and contributors: Martin Bouvet, Simon Lefort, and Juan Iaconucci (teaching assistants in Spring 2025).