

# Tutorial on Using Verilog HDL in Logic Design

Developed for: **COE-202 Students and Similar level**

By: **Dr. Abdelhafid Bouhraoua**

## 1. Starting with Verilog:

Verilog-HDL is a standard hardware description language that is easy to use and learn. The expected learning curve should be very fast.

A hardware description starts from defining the *models* you are going to use for your design. A model is a construct similar to a block (hardware) or a user-defined type (in software). Designers will define all their models then they will use them to build the whole circuit by using one or several instances.

## 2. Verilog module

A model is called **module**. The module constitutes the basic container for any Verilog description. A module is defined as follows (key words are in bold and optional names are between <>)

```
module <module_name> (<port_list>);  
...  
endmodule
```

A module name should be unique for the same project. It is strongly recommended that you put every module in a separate file with the file name being the same as the module name with the extension .v:

The module *first\_circ* should be put in the file *first\_circ.v* (use the .vl extension for the simulator we are using).

## 3. Port list definition

The port list is the module interface to the external world of the module. It is in other terms, how the module communicates with the outside world. The port list contains only a list of names. The direction and wire width of each port will be defined after the module as shown in the example below:

```
module first_circ ( X,  
                  Y,  
                  Z,  
                  F );  
  
input      X;  
input      Y;  
input      Z;  
output     F;  
  
endmodule
```

From the description above it is clear that:

- X, Y and Z are three inputs with a width of a single wire (1 bit)
- F is a single wire output

## 4. Describing gates

Once the ports are defined, the next step is to describe the circuit itself in terms of gate connections so that it can be simulated.

All the usual gates are **predefined** in the verilog language. All gates, except the inverters, have an arbitrary number of inputs. Gates are **instanciated** (which means copied as instances from a model) as modules with connections starting from the **single** output followed by all the inputs. Inputs and outputs are single wires. The following illustrates the description of the following Boolean function:

$$F = X'YZ + XYZ' + X'Y'$$

```

module first_circ(
    X,
    Y,
    Z,
    F);

input    X;
input    Y;
input    Z;
output   F;

// Internal Wire Definition
wire     X_b;
wire     Y_b;
wire     Z_b;
wire     O1;

// Instances
not not1(X_b, X);           // This is the inverter with one
not not2(Y_b, Y);          // input and one output
not not3(Z_b, Z);

and and1(O1, X_b, Y, Z);   // Gate instances need to be given a name
and and2(O2, X, Y, Z_b);   // names like not1, and1 and and3 are
and and3(O3, X_b, Y_b);    // user-defined
or  or1(F, O1, O2, O3);

endmodule

```

The simplified syntax of the built-in gates is given below:

```
<gate_id> <gate_instance_name> (<output>, <input 1>, ..., <input n>);
```

The predefined gates are given in the table below:

| Gate Name | Gate Id | Equation       |
|-----------|---------|----------------|
| inverter  | not     | $O = X'$       |
| and gate  | and     | $O = XY$       |
| or gate   | or      | $O = X + Y$    |
| xor gate  | xor     | $O = X (+) Y$  |
| nand gate | nand    | $O = (XY)'$    |
| nor gate  | nor     | $O = (X + Y)'$ |

## 5. Simulation

To simulate any circuit the following operations and actions need to be carried out:

1. Place the circuit on a testing platform
2. Connect its inputs to some signal generators,
3. Program these signal generators to drive the inputs with certain values at different times,
4. Connect the output to signal capture and recording devices

5. Start the test
6. Look at the signal recordings

All these steps can be handled by the verilog language itself and by the simulation environment. Simulation environments are Computer Aided Design (CAD) software tools that are developed to support the verilog language and interpret it according to the specifications of the standard document of the IEEE Std 1364-2001.

Commercially available tools are often parts of very expensive suites. Cadence and Synopsys are the world leaders in the CAD for VLSI design. Other smaller companies have also simulators. Free of charge simulators exist and we are going to use one of them in this tutorial. It is called the Icarus Verilog simulator.

### 5.1. Testing platform or testbench

The testing platform is just another verilog module where the circuit we want to test is instantiated. In general, because circuits are different, testbenches are developed for every circuit design. The testbench development (part of the simulation setup) occurs in parallel with the design of the circuit itself.

Testbenches are modules that do not have inputs or outputs. The necessary drivers and capture devices are declared as variables and are modified and/or tested within the definition of the testbench itself.

The testbench is started by defining the module and instantiating the circuit *first\_circ* in it.

```
module first_tb;

  reg  XA;
  reg  YA;
  reg  ZA;
  wire FO;

  // Instantiate the circuit and Connect the I/Os
  first_circ first1(
    .X (XA),
    .Y (YA),
    .Z (ZA),
    .F (FO));
endmodule
```

We notice that the module definition ends with a semicolon confirming the fact that a testbench does not have input/outputs.

The different drivers of the inputs are declared with the keyword **reg** to allow the definition of the test scenario to set values to them so that they can drive the circuit inputs with these values.

The output of the circuit is declared as **wire** which means that it is a wire that let us observe what value is being driven by the circuit.

The circuit model is instantiated starting by its module name *first\_circ*. The instance is given an instance name *first1* that immediately follows the model name. The inputs outputs are connected as follows:

```
.<port> (<reg or wire variable>),
```

This means that to explicitly connect the port *X* of model *first\_circ* to the testbench variable *XA*, the following syntax is followed:

```
.X (XA),
```

## 5.2. Test Scenario

The circuit we want to test is a combinational circuit that is composed of regular gates. Any change on any of the inputs will eventually cause a change on the output. To test all the combinations we should recreate these combinations following the same procedures as in truth tables.

Having three inputs, the truth table will show:

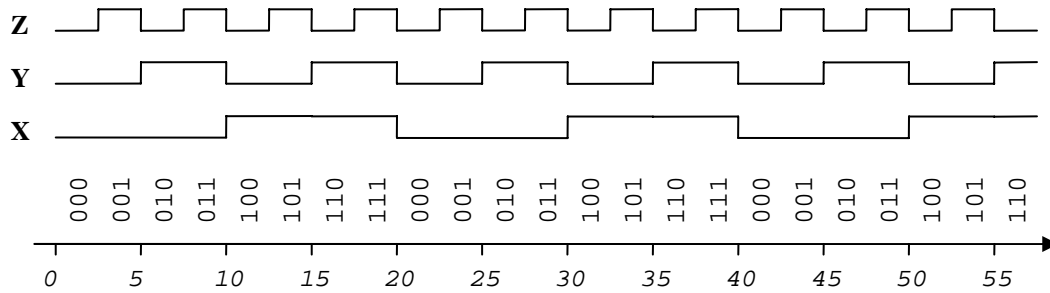
| X | Y | Z | F |
|---|---|---|---|
| 0 | 0 | 0 | ? |
| 0 | 0 | 1 | ? |
| 0 | 1 | 0 | ? |
| 0 | 1 | 1 | ? |
| 1 | 0 | 0 | ? |
| 1 | 0 | 1 | ? |
| 1 | 1 | 0 | ? |
| 1 | 1 | 1 | ? |

To recreate a continuous scan of all these combinations we use a periodic change from 0 to 1 and from 1 to 0.

Suppose that:

- signal Z will change its value from 0 to 1 and from 1 to 0 every 5 units of time
- signal Y will change its value from 0 to 1 and from 1 to 0 every 10 units of time
- signal X will change its value from 0 to 1 and from 1 to 0 every 20 units of time

The timing diagram of these three signals will be as follows:



It clearly shows how all the combinations are covered (several times).

To do that, the signals should be:

- initialized first using the *initial* statement
- changed every  $n$  units of time using:
  - The *always* statement that repeatedly does the same action
  - The *#* operator used to add delay before executing an action

To set the simulation time unit to nano-seconds, the following compiler directive is used:

```
`timescale 1ns/1ns
```

The testbench code becomes as follows:

```
`timescale 1ns/1ns
module first_tb;

reg  XA;
reg  YA;
reg  ZA;
wire FO;

// Instanciate the circuit and Connect the I/Os
first_circ first1(
    .X (XA),
    .Y (YA),
    .Z (ZA),
    .F (FO));

// initialization of the variables
initial
begin
    XA = 0;
    YA = 0;
    ZA = 0;
end

always
    #5 ZA = ~ZA;    // change ZA to its opposite every 5 ns

always
    #10 YA = ~YA;   // change YA to its opposite every 10 ns

always
    #20 XA = ~XA;   // change XA to its opposite every 20 ns

endmodule
```

### 5.3. Signal Capture

The capture of the changes on the output signal of the circuit under test is realized by two other verilog directives that enable dumping all of the changes in a file.

These directives are inserted within another *initial* block. By the way, *initial* blocks should be defined only in testbenches and not in designs. This is because a design's aim is to be translated into hardware components (gates) where *initial* statements are meaningless.

The *initial* block used to dump simulation signal values is:

```
initial
begin
    $dumpfile("first_wave.vcd");
    $dumpvars(0,first_tb);
end
```

The dump file or wave file should have the *.vcd* extension. The *\$dumpvars* statement has two parameters:

- The first parameter indicates the level of hierarchy. A value of 0 means all levels; 1 means the current level; 2 means the current and one level below etc... These levels correspond to modules. For example, the first level in our case is the module *first\_tb* and the second level is the module instance *first1*.
- The second parameter is the module instance name from where the dump starts.

#### 5.4. Ending the simulation

If simulated as is, the simulation will run forever unless stopped explicitly by the user. Some simulators can handle unexpected stops from the users while some others cannot. Therefore, it is important to specify when the simulation should end. A verilog directive has been defined for that purpose. It is the *\$finish* directive.

A statement with a delay corresponding to the amount of time the simulation needs to wait (from that point where the *\$finish* statement is inserted) before ending itself. The statement should always be part of a *initial* block. The statement is:

```
initial
    #<simulation_time> $finish;
```

The simulation time in our case should be sufficient for executing all the combinations of the input signals which is approximately 50 ns.

The testbench final look is:

```
`timescale 1ns/1ns
module first_tb;

reg  XA;
reg  YA;
reg  ZA;
wire FO;

// Instanciate the circuit and Connect the I/Os
first_circ first1(
    .X (XA),
    .Y (YA),
    .Z (ZA),
    .F (FO));

// initialization of the variables
initial
begin
    XA = 0;
    YA = 0;
    ZA = 0;
end

always
    #5 ZA = ~ZA;    // change ZA to its opposite every 5 ns

always
    #10 YA = ~YA;  // change YA to its opposite every 10 ns

always
    #20 XA = ~XA;  // change XA to its opposite every 20 ns

initial
begin
    $dumpfile("first_wave.vcd");
    $dumpvars(0,first_tb);
end

initial
    #50 $finish;

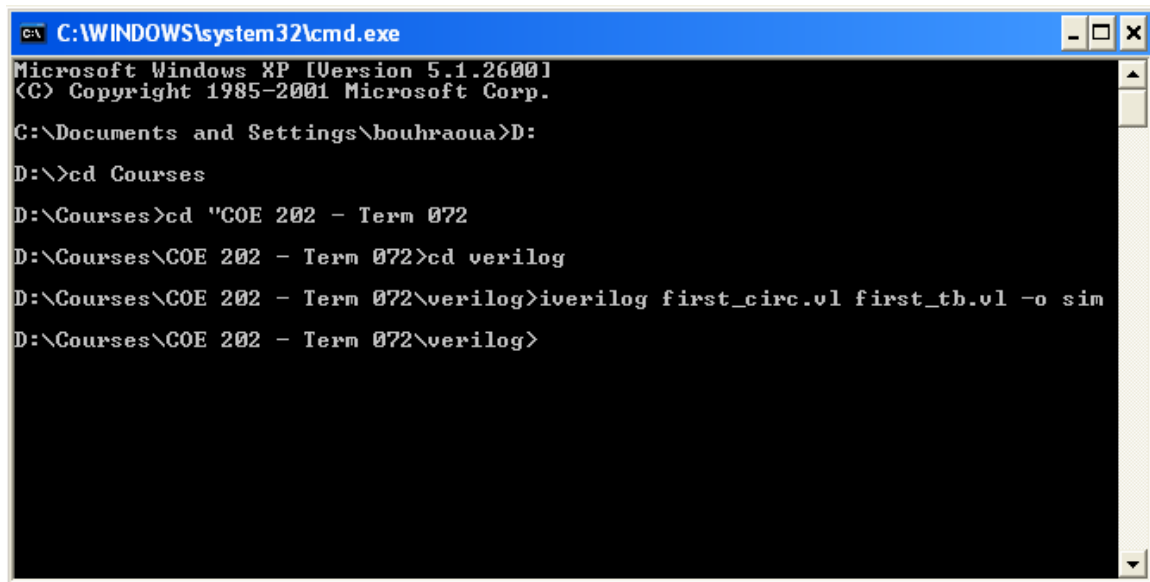
endmodule
```

### 5.5. Compiling the design

Once the design and testbench finalized, they should be compiled. The installed verilog compiler that we are using is used through a command line window. The following steps should be followed:

- Open a command line window (old DOS window) by typing *cmd* in the *run* prompt of the *Start* menu in *Windows*.
- Go to the directory where your files are
- invoke the compiler by typing: **iverilog <verilog files> -o <simulation output file>**

The following screen capture shows the compilation step as it is done for the current design:



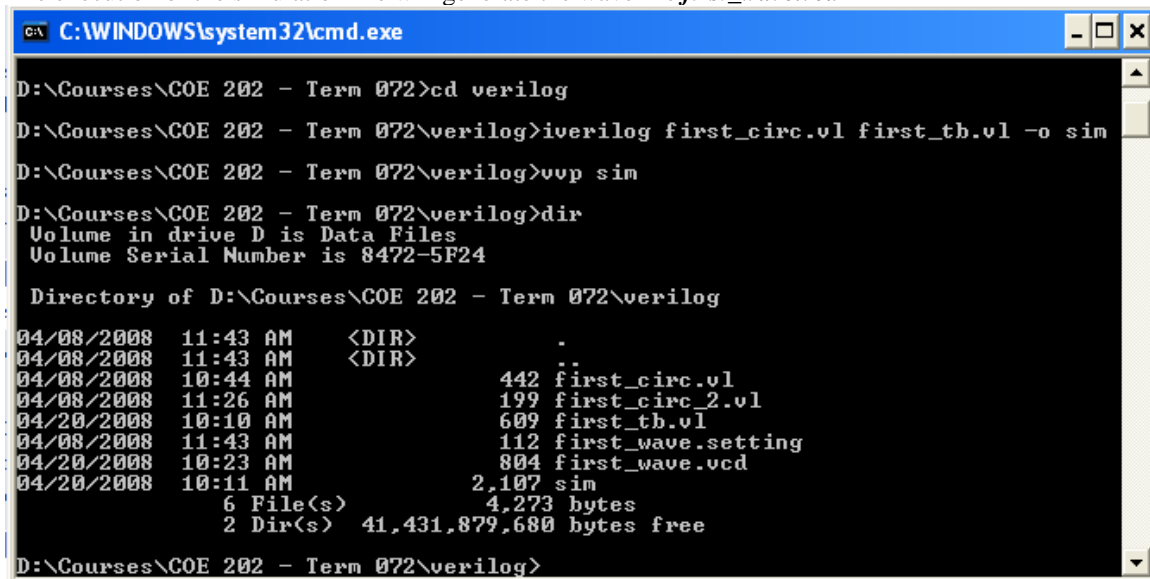
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\bouhraoua>D:
D:\>cd Courses
D:\Courses>cd "COE 202 - Term 072"
D:\Courses\COE 202 - Term 072>cd verilog
D:\Courses\COE 202 - Term 072\verilog>iverilog first_circ.vl first_tb.vl -o sim
D:\Courses\COE 202 - Term 072\verilog>
```

### 5.6. Executing the simulation

The previous step compiled the simulation file into a simulator executable file. This file is not directly executable in Windows. It is executed by a simulation kernel called *vvp*.

The execution of the simulation file will generate the wave file *first\_wave.vcd*



```
C:\WINDOWS\system32\cmd.exe

D:\Courses\COE 202 - Term 072>cd verilog
D:\Courses\COE 202 - Term 072\verilog>iverilog first_circ.vl first_tb.vl -o sim
D:\Courses\COE 202 - Term 072\verilog>vvp sim
D:\Courses\COE 202 - Term 072\verilog>dir
Volume in drive D is Data Files
Volume Serial Number is 8472-5F24

Directory of D:\Courses\COE 202 - Term 072\verilog
04/08/2008  11:43 AM    <DIR>          .
04/08/2008  11:43 AM    <DIR>          ..
04/08/2008  10:44 AM                442 first_circ.vl
04/08/2008  11:26 AM                199 first_circ_2.vl
04/20/2008  10:10 AM                609 first_tb.vl
04/08/2008  11:43 AM                112 first_wave.setting
04/20/2008  10:23 AM                804 first_wave.vcd
04/20/2008  10:11 AM            2,107 sim
               6 File(s)              4,273 bytes
               2 Dir(s)  41,431,879,680 bytes free

D:\Courses\COE 202 - Term 072\verilog>
```

### 5.7. Viewing the waveform

Finally the waveform is viewed in a waveform viewer. The one available is called Wave. The window will look like:

