MODULE 1: INTRODUCTION TO HARDWARE DESCRIPTION LANGUAGE

Learning Outcomes:

1. Identify the different Hardware programmable technology software
2. Describe the functionality and applications of different Hardware description Language software
3. Differentiate the functionality and applications of different HDL technology Platform Explain different hierarchy and modelling structures of HDL's
4. Compose and modify simple modules to produce specified outputs
5. Predict the output of simple modules

A **hardware description language (HDL)** is similar to a typical computer programming language except that an HDL is used to describe hardware rather than a program to be executed on a computer. Many commercial HDLs are available. Some are proprietary, meaning that they are provided by a particular company and can be used to implement circuits only in the technology provided by that company.

There are a lot of HDL but let us focus on the recognized and officially endorsed **of Institute of Electrical and Electronics Engineers (IEEE)** standard. The IEEE is a worldwide organization that promotes technical activities to the benefit of society in general. One of its activities involves the development of standards that define how certain technological concepts can be used in a way that is suitable for a large body of users.
Two HDLs are IEEE standards: Verilog HDL and VHDL (Very High Speed Integrated Circuit Hardware Description Language). Both languages are in widespread use in the industry.

* Verilog HDL
* ABEL
* VHDL

VHDL is the most common

* Large standard developed by US DoD
* VHDL = VHSIC HDL
* VHSIC = Very High Speed Integrated Circuit

Verilog was started initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is rumored that the original language was designed by taking features from the most popular HDL language of the time, called HiLo as well as from traditional computer language such as C. At that time, Verilog was not standardized and the language modified itself in almost all the revisions that came out within 1984 to 1990.

Verilog simulator was first used beginning in 1985 and was extended substantially through 1987.The implementation was the Verilog simulator sold by Gateway. The first major extension was Verilog−XL, which added a few features and implemented the infamous "XL algorithm" which was a very efficient method for doing gate−level simulation.

The time was late 1990. Cadence Design System, whose primary product at that time included Thin film process simulator, decided to acquire Gateway Automation System. Along with other Gateway product, Cadence now became the owner of the Verilog language, and continued to market Verilog as both a language and a simulator. At the same time, Synopsys

was marketing the top−down design methodology, using Verilog. This was a powerful combination.

In 1990, Cadence recognized that if Verilog remained a closed language, the pressures of standardization would eventually cause the industry to shift to VHDL. Consequently, Cadence organized Open Verilog International (OVI), and in 1991 gave it the documentation for the Verilog Hardware Description Language. This was the event which "opened" the language.

OVI did a considerable amount of work to improve the Language Reference Manual (LRM), clarifying things and making the language specification as vendor−independent as possible.

Soon it was realized, that if there were too many companies in the market for Verilog, potentially everybody would like to do what Gateway did so far − changing the language for their own benefit. This would defeat the main purpose of releasing the language to public domain. As a result in 1994, the IEEE 1364 working group was formed to turn the OVI LRM into an IEEE standard. This effort was concluded with a successful ballot in 1995, and Verilog became an IEEE standard in December, 1995.

When Cadence gave OVI the LRM, several companies began working on Verilog simulators. In 1992, the first of these were announced, and by 1993 there were several Verilog simulators available from companies other than Cadence. The most successful of these was VCS, the Verilog Compiled Simulator, from Chronologic Simulation. This was a true compiler as opposed to an interpreter, which is what Verilog−XL was. As a result, compile time was substantial, but simulation execution speed was much faster.

In the meantime, the popularity of Verilog and PLI was rising exponentially. Verilog as a HDL found more admirers than well−formed and federally funded VHDL. It was only a matter of time before people in OVI realized the need of a more universally accepted standard. Accordingly, the board of directors of OVI requested IEEE to form a working committee for establishing Verilog as an IEEE standard. The working committee 1364 was formed in mid 1993 and on October 14, 1993. The standard, which combined both the Verilog language syntax and the PLI in a single volume, was passed in May 1995 and now known as IEEE Std. 1364−1995.

After many years, new features have been added to Verilog, and new version is called Verilog 2001. This version seems to have fixed lot of problems that Verilog 1995 had. This version is called 1364−2000. Only waiting now is that all the tool vendors implementing it.

II. Explore it More

Computer Aided Design (CAD) software makes it easy to implement a desired logic circuit by using a programmable logic device, such as a field-programmable gate array (FPGA) chip. A typical FPGA CAD flow is illustrated in Figure 1.

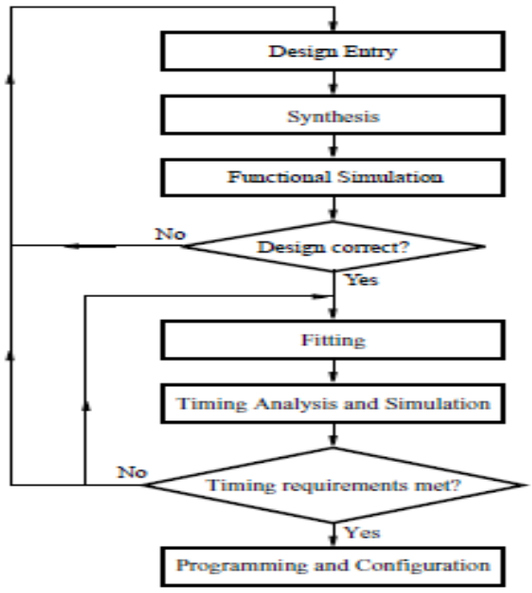


Figure 1. Typical Simulation Flow.

The CAD flow involves the following steps:

· **Design Entry** – the desired circuit is specified either by means of a schematic diagram, or by using a hardware description language, such as Verilog or VHDL

· **Synthesis** – the entered design is synthesized into a circuit that consists of the logic elements (LEs) provided in the FPGA chip

· **Functional Simulation** – the synthesized circuit is tested to verify its functional correctness; this simulation does not take into account any timing issues

· **Fitting** – the CAD Fitter tool determines the placement of the LEs defined in the netlist into the LEs I an actual FPGA chip; it also chooses routing wires in the chip to make the required connections between specific Les.

· **Timing Analysis** – propagation delays along the various paths in the fitted circuit are analyzed to provide an indication of the expected performance of the circuit

· **Timing Simulation** – the fitted circuit is tested to verify both its functional correctness and timing

· **Programming and Configuration** – the designed circuit is implemented in a physical FPGA chip by programming the configuration switches that configure the LEs and establish the required wiring connection.

In this course, we are going to use Verilog in because it is close to a C programming language standard. Although the VHDL is the most common and the two languages differ in many ways, the choice of using one or the other when studying logic circuits is not particularly important, because both offer similar features.

ModelSim Simultator is an Altera ASIC family simulator used to test the functionality of the design.

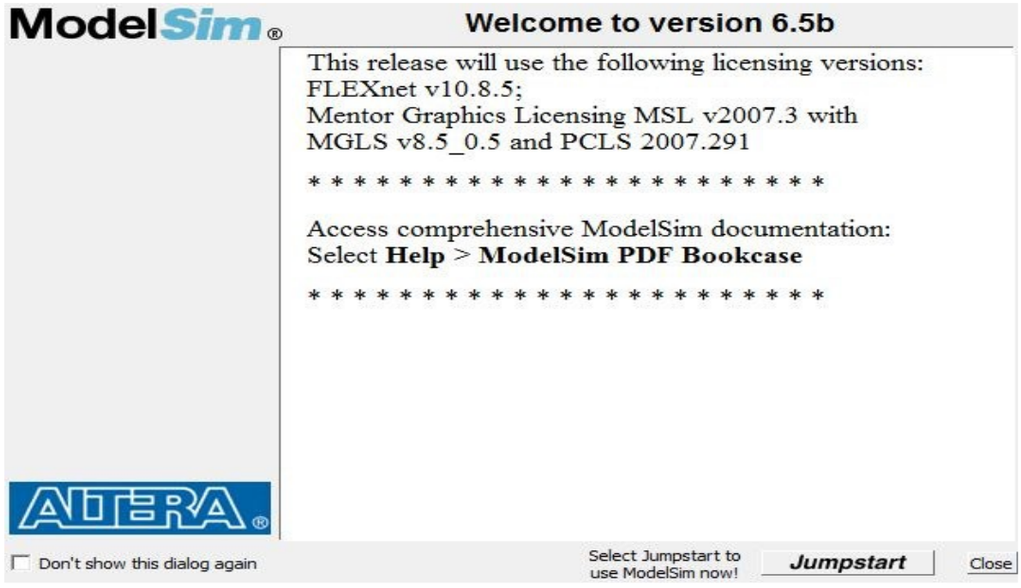


Figure 2. Model Sim Welcome Window

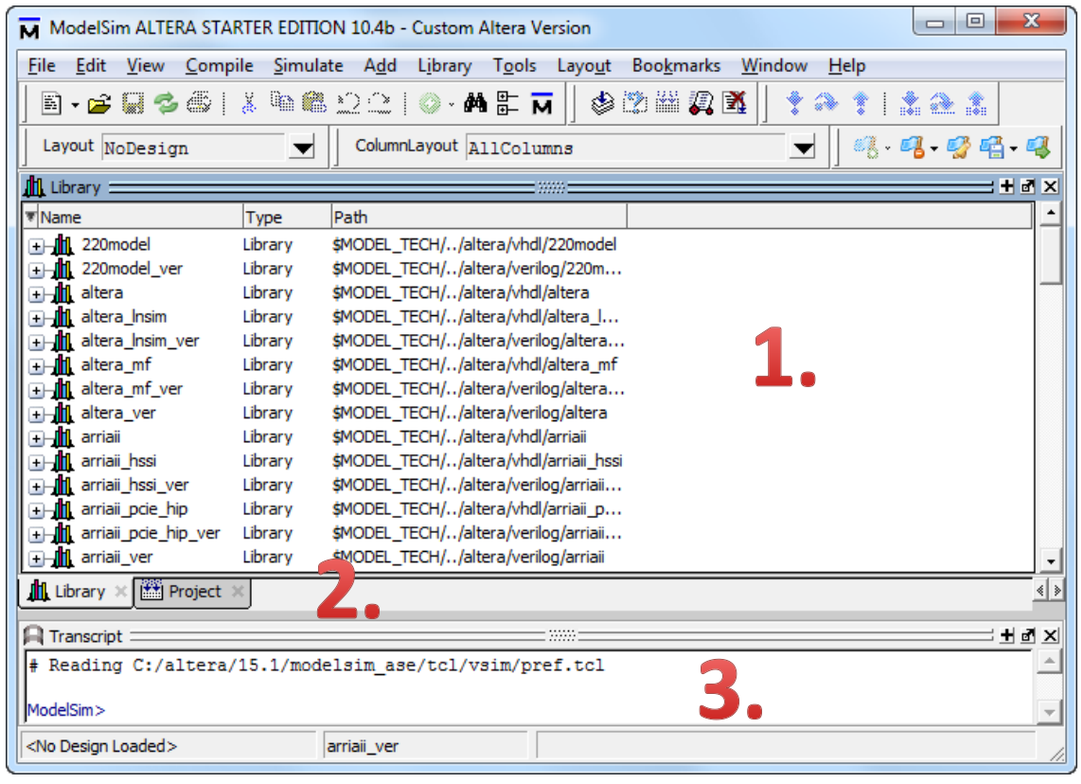


Figure 3. The ModelSim Window in Showing the working library

The **ModelSim** Program window consists of three sections:

1. main menu at the top
2. set of workspace tabs
3. command prompt

The menu is used to access functions available in ModelSim.

The workspace contains a list of modules and libraries of modules available to you, as well as details of the project you are working on. A new work area will appear on the right of the libraries of modules when needed to display waveforms and/or text files. Finally, the command prompt at the bottom shows feedback from the simulation tool and allows users to enter commands. We will begin by creating a project where all design files to be simulated are included. We compile the design and then run the simulation. Based on the results of the simulation, the design can be altered until it meets the desired specifications.

III. Explain Further

After you have set up the software by installing it to your device. We will be creating a sample project. We will be following a simple AND gate following the basic structure of a Verilog module.

Lexical Token

Documentation is important in every programming language. Placing **comments** in a Verilog code is allowed. You just need to follow correct symbol and delimiters. Since Verilog code and C language has many things alike, so is its symbols.

```
// This is a short comment
/*This is a long Verilog comment that spans two lines */
```

Another is the **White space** such as **SPACE** and **TAB** and blank lines are ignored by the Verilog Compiler. Multiple statements can be written on a single line such as

```
f = w0; if (s  == 1) f = w1;
```

Verilog Keywords

As most programming languages have there are keywords that programmers cannot use for any other purpose intended in the language, Verilog have keywords which defines the language constructs. A **keyword** signifies an activity to be carried out, initiated or terminated. Verilog keywords are in small letters and are case sensitive.

Example:

module- signifies the beginning of a module definition
endmodule- signifies the beginning of a module definition
begin- signifies the beginning of a block statement
end- signifies the end of a block statement
if- signifies a conditional activity to be checked
while- signifies a conditional activity to be carried out

Structure of a module

In Verilog, a module is declared by the keyword module. A corresponding keyword endmodule must appear at the end of the module definition. Each module must have a module_name, which is the identifier for the module, and a module_terminal_list, which describes the input and output terminals of the module.

module module name [(port name{, port name})];
[*parameter* declarations]
[*input* declarations]
[*output* declarations]
[*inout* declarations]
[*wire* or *tri* declarations]
[*reg* or *integer* declarations]
[*function* or *task* declarations]
[*assign* continuous assignments]
[*initial* block]
[*always* blocks]
[gate instantiations]
[module instantiations]
endmodule

Creating a Project

1. To create a project in ModelSim, select File > New > Project.... A Create Project window shown in Figure 1.4 will appear.

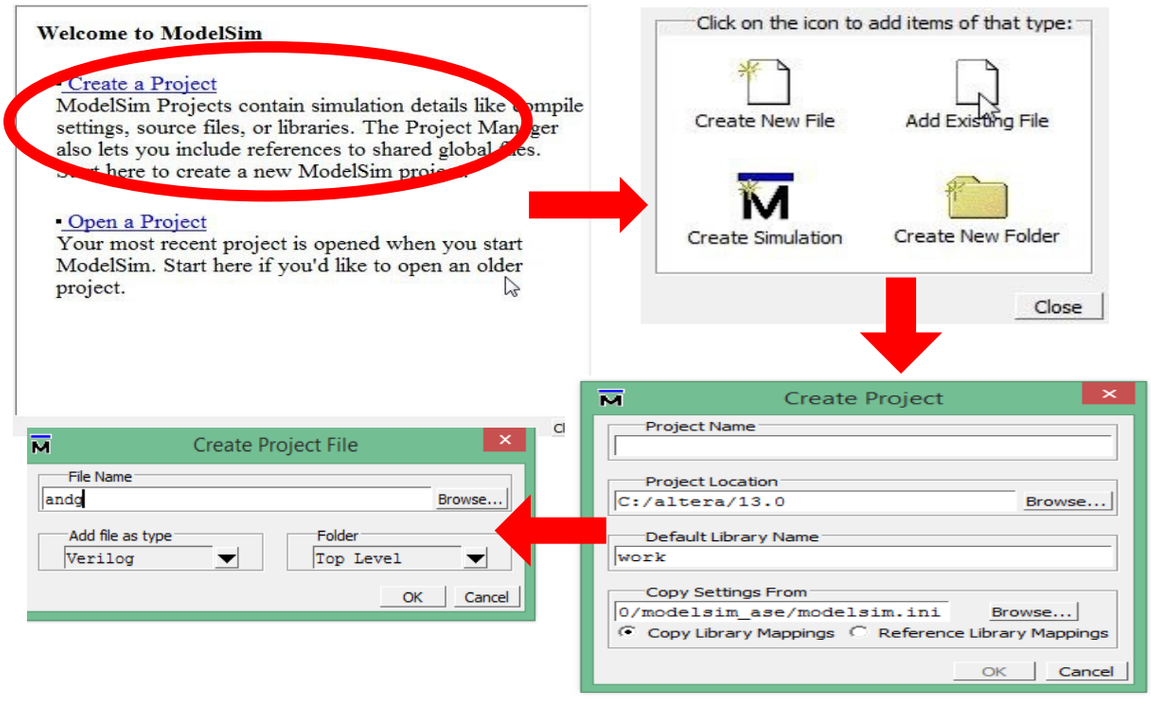


Figure1.4 ModelSim windows in creating a project

The create project window consists of several fields: project name, project location, default library name, and copy settings field. Project name is a user selected name and the location is the directory where the

source files are located. For this activity, we will name the project as ***ANDG***, and save it to its default project location and project library which is ***C:/altera/13.0*** and ***work*** respectively.

2. Select Add Existing button to create the project file. It is preferred to create a file name **andg** which is easier to remember. You should be seeing a similar window after you had created your file

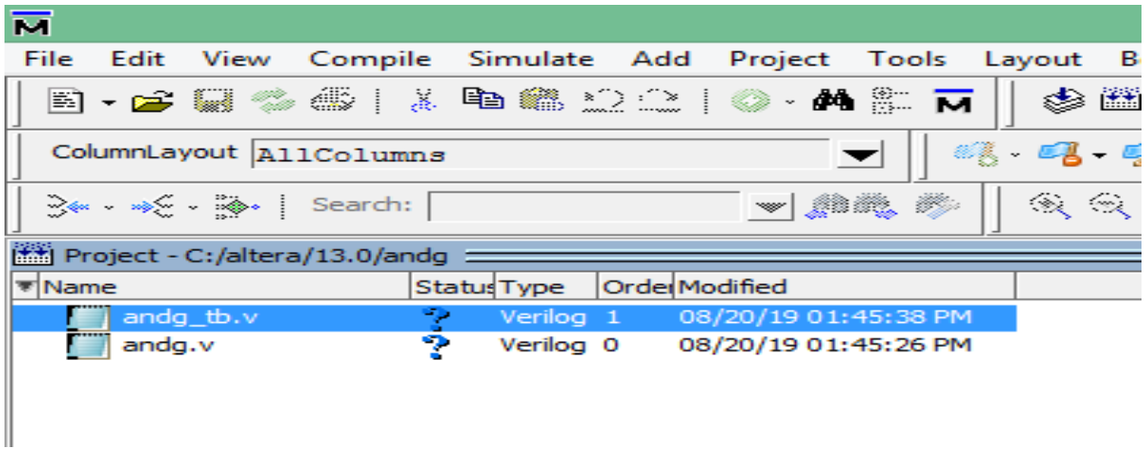


Figure1.5 ModelSim windows in Verilog code editor inside the Project file andg

3. **Right click** to reveal the **editor** and **compile** function. Enter the sample Verilog code below. This Verilog code is in continuous model which directly sends the resulting function to the output.

```
module andg(x,a,b);

        input a,b;
        output x;

        assign x=a&b;

endmodule
```

Table 1.1 Code Explanations of Syntax

Line No.	Explanation
1	This line shows the syntax of the Verilog format of a module. Beginning with keyword ***module,*** modulename ***andg*** and ports ***x,a*** and ***b.***
3	Declaration of ***input*** ports ***a*** and ***b.***
4	Declaration of ***output*** ports ***a*** and ***b.***
5	Assign keyword is used to tell ports a and b to execute the function of gate AND and send its result signa in port x
7	The keyword endmodule

4. After encoding the sample Verilog code, we will compile the andg file. To compile, right click again the file you wish to compile or edit. Select Compile →Compile Selected as shown in Figure 1.6.

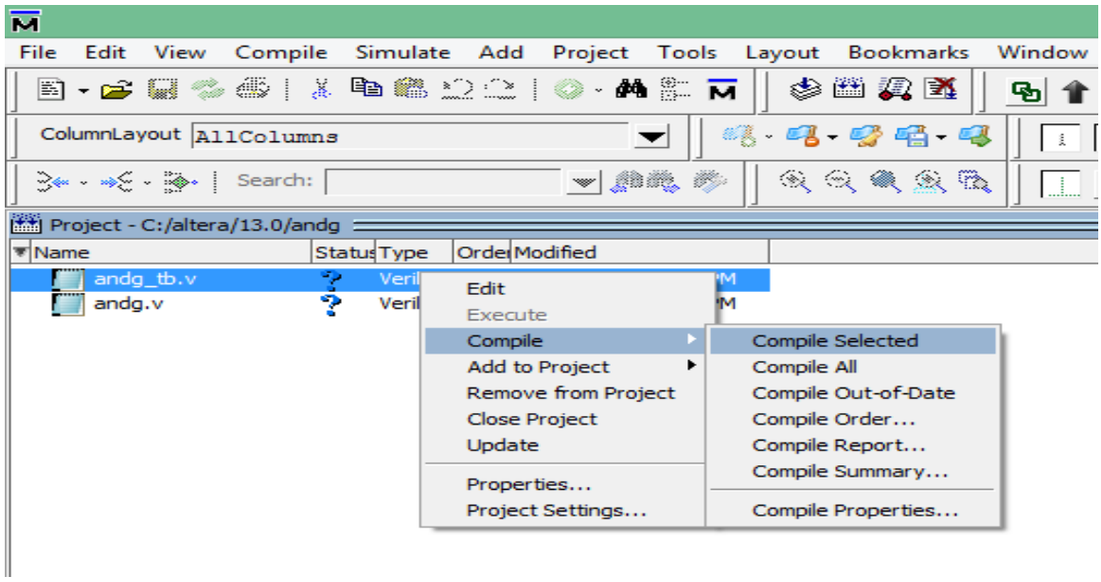


Figure 1.6 ModelSim Compilation step of selected file

Compiling the code to check for errors and correct it outright. This process is also known as **debugging**

5. After successful compilation, you should see a check symbol beside your selected file. This signifies that your Verilog code is syntactically correct. This file should be ready for simulation and showing output waveform.

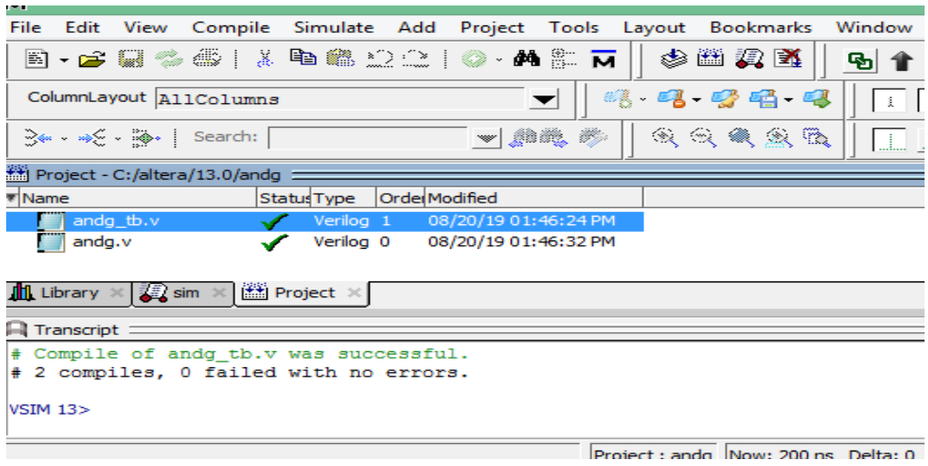


Figure 1.7 ModelSim Compiled file having check symbol

6. We are going to simulate the file. Click on **Simulate→ Start simulation**

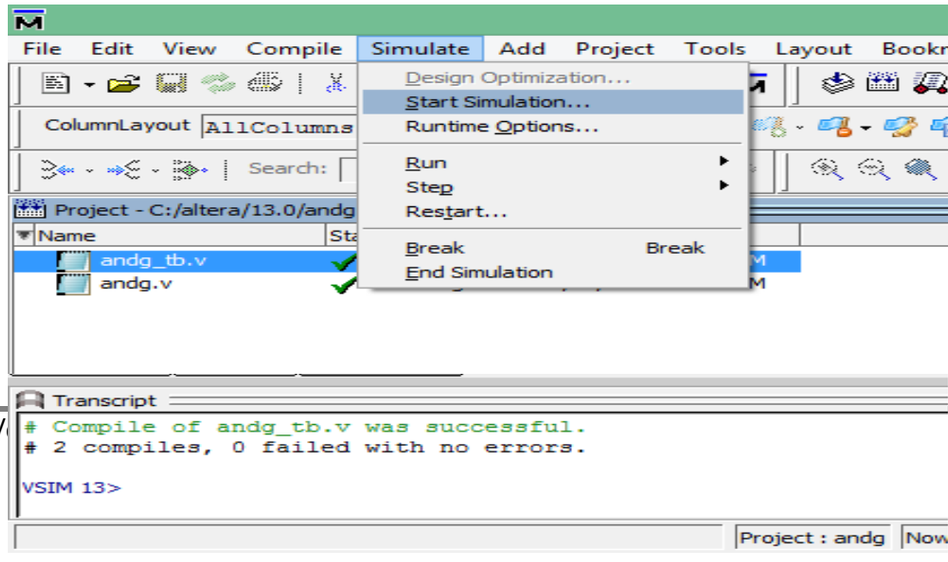


Figure 1.8 ModelSim Compilation step of selected file

7. After Clicking Start Simulation, you will see a similar window that contains the libraries of ModelSim. The projects we create in ModelSim is saved to **work** library by default. We select **work** library which contains the file we need to simulate. Click the plus sign on work to reveal the files then select the file created **andg** then click **OK.**

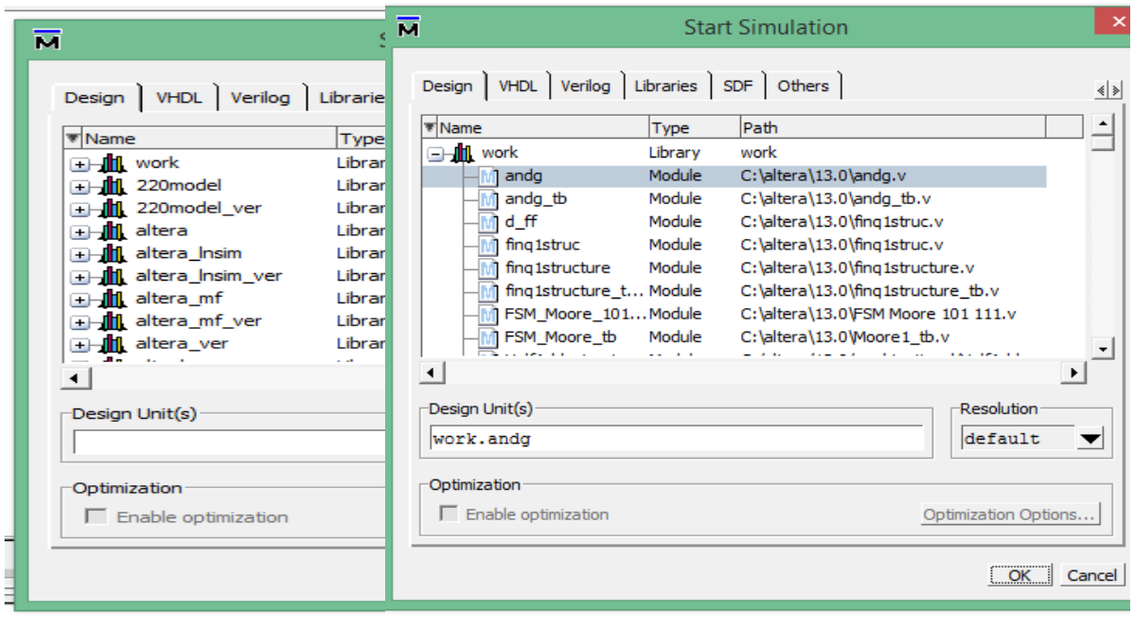


Figure 1.9 ModelSim Compilation work library

8. After selecting the file to Simulate. You can see the window of your ModelSim simulator the synthesized instances or ports from your Verilog code file. In order to test the instances and see the output waveform, click **Simulate→Run →Run 100** or simply press **F9** key.

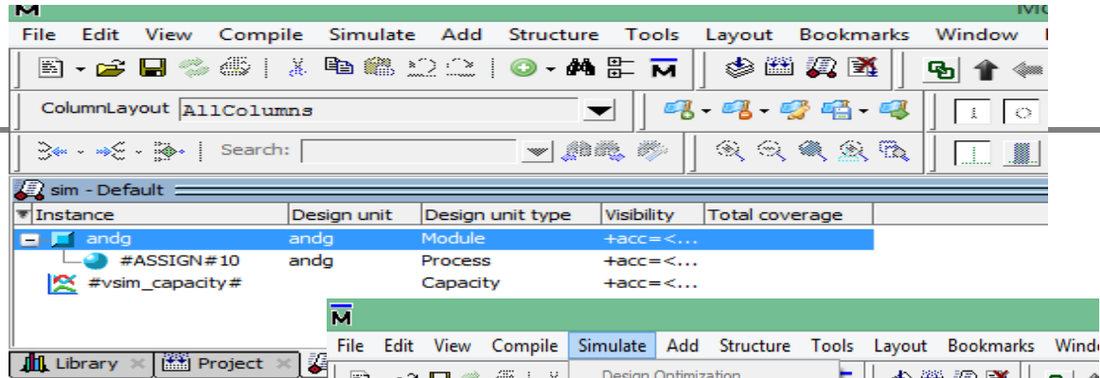


Figure 1.10 ModelSim Run Simulstion of Module instances window

After this step you can see the window for waveform simulation. By default, the timing of the waveform is set to 100 us. You can modify the timing by clicking the timing button but for this design purpose we will let it be on default mode. The practice to create waveform as a simulation output is to create a test bench which we will discuss on module 6. For this design, we will force the signal stimuli signal to work on whatever timing repetition we wish.

9. To force a signal to the stimuli, right click on the port and select **Force.** We will force the selected stimuli **a** and **b** to low and high manually**.** We will select it according to the truth table of **AND** gate**. Press F9.**

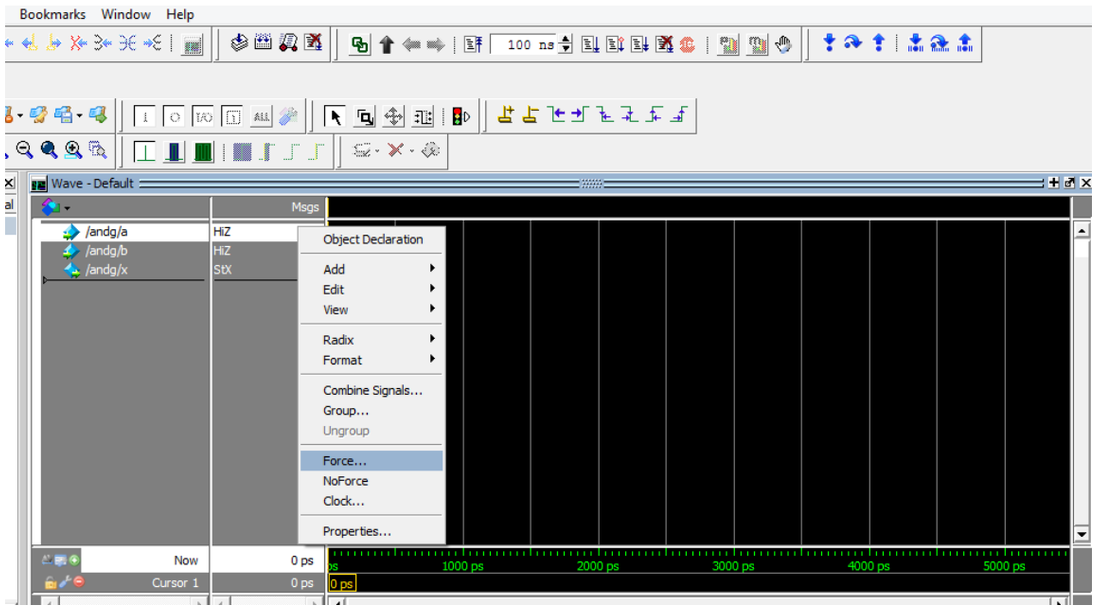


Figure 1.11 ModelSim Forcing Signal on Stimuli of the synthesized instance

Figure 1.12 shows the output waveform timing diagram of our Verilog code andg. It is a simulation of AND gate logic. The logic for AND gate is the all combinations of input stimuli will result to 0 except if the two input stimuli is both 1. With the result of the waveform yielding the correct logic means our Verilog code is correct.

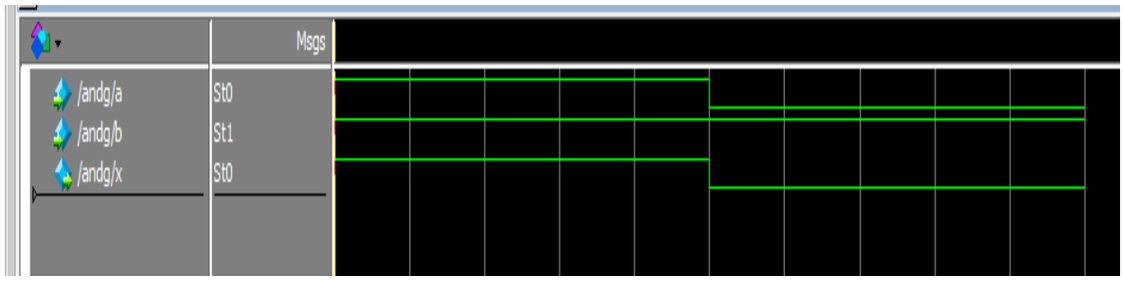


Figure 1.12 Waveform created by forcing the input stimuli by ModelSim

IV. Elaborate Application

Gate level Modelling

The circuit presented below is a simple schematic circuit diagram. Evaluating this into and expression we could have **D= AB+C'** and **C'=E.**

Table 1.2 truth table of the given Boolean Equation

D= A•B+[C]

A	B	C	D
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

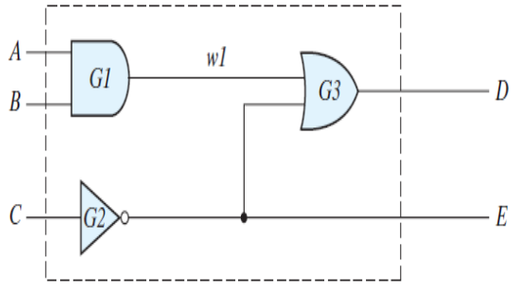


Figure 1.13 Sample Simple Circuit

Gate level modelling suggest module implementation are in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

Logic gates in Verilog HDL standard

Logic Gates	Verilog Code	Description
AND	and	Functions as logical AND

OR	or	Functions as logical OR
NOT	not	Functions as logical NOT
XOR	xor	Functions as logical XOR
NAND	nand	Functions as logical NAND

Here is an example of a gate level modelling Verilog code module for **Figure1.13.** Declaring ports and its module name **Simple_Circuit**. We will temporarily name the gates in the image as **G1**, **G2** and **G3** respectively. We will name the instance output wire to be **w1**. A **wire** represents a simple wire interconnection and driven by an only declared output.

```
1. module Simple_Circuit (A,B,C,D,E);
2. input      A,B,C;
3. output     D,E;
4. wire       w1;
5.
6. and        G1 (w1,A,B);
7. not        G2 (E,C);
8. or         G3 (D,w1,E);
9. endmodule
```

Figure 1.14 Verilog Code for Simple_Circuit

The circuit Verilog code is written in a Gate level modelling manner where actual gates were implemented. G1 labeled as our AND gate. in this code logical gates are called upon as it is the circuit. For instance our G1 is our AND gate implementation. G1 is our initially assigned gate name. The order of the ports format is **(output, input1… inputn) .**

Table 1.3 Code Explanations of Syntax

Line No.	Explanation
1	This line shows the syntax of the Verilog format of a module. Beginning with keyword ***module,*** modulename ***Simple_Circuit*** and ports ***A,B,C,D*** and ***E.***
2	Declaration of ***input*** ports ***A,B*** and ***C.***
3	Declaration of ***output*** ports ***D*** and ***E.***
4	Declaration of ***G1*** AND gate, ***w1*** as wire output driven by input ***A*** and ***B***.
5	Declaration of ***G2*** NOT gate, ***E*** as output driven by input ***C***.
6	Declaration of ***G3*** OR gate, ***D*** as output driven by ***w1***which is the ***G3*** input and ***E***.
7	The keyword endmodule

Verilog Level of Abstraction

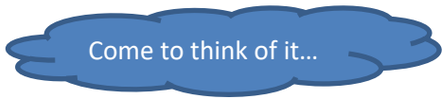
The module behaves identically with the external environment irrespective of the level of abstraction at which the module is described. The internals of the module are hidden from the environment. Thus, the level of abstraction to describe a module can be changed without any change in the environment. These levels will be studied in detail in separate modules later. The levels are defined below.

Behavioral or algorithmic level- This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

Dataflow level- At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

Gate level- The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

Switch level- This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.



Verilog provides the concept of a module. A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.

Verilog allows the designer to mix and match all four levels of abstractions in a design. In the digital design community, the term register transfer level (RTL) is frequently used for a Verilog description that uses a combination of behavioral and dataflow constructs and is acceptable to logic synthesis tools.

If a design contains four modules, Verilog allows each of the modules to be written at a different level of abstraction. As the design matures, most modules are replaced with gate-level implementations.

Normally, the higher the level of abstraction, the more flexible and technology independent the design. As one goes lower toward switch-level design, the design becomes technology-dependent and inflexible. A small modification can cause a significant number of changes in the design. Consider the analogy with C programming and assembly language programming. It is easier to program in a higher-level language such as C. The program can be easily ported to any machine. However, if you design at the assembly level, the program is specific for that machine and cannot be easily ported to another machine.

References:

Brown, S. D., & Vranesic, Z. G. (2014). Fundamentals of digital logic with Verilog design. New York: McGraw-Hill Higher Education.
Harris, S. L., & Harris, D. M. (2018). Digital design and computer architecture. Amsterdam: Elsevier / Morgan Kaufmann.
Paltnikar, Samir (2003). Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition / Prentice Hall PTR
Padmanabhan,T.R. & B. Bala Tripura Sundari (2004). Design through Verilog HDL / IEEE Press
Quick Quartus: Verilog. (n.d.). Retrieved August 18, 2020, from http://www.swarthmore.edu/NatSci/echeeve1/Ref/embedRes/QQS_V/QuickQuartusVerilog.html
Verilog Simulation. (n.d.). Retrieved August 18, 2020, from https://www.chipverify.com/verilog/verilog-simulation

Name:_____Score:_____
Course/Sec:_____Schedule:_____Date Submitted:_____

V. Evaluate my Learning

A. Create a Verilog module for the following with waveform of each gates (and,or,not,xnor,xor,nand,nor) following these Boolean equation.

- And Gate: Y = (A.B)
- Or Gate: Y = (A + B)
- Nand Gate: Y = (A.B)'
- Nor Gate: Y = (A+B)'
- Xor Gate: Y = A.B' + A'.B
- Xnor Gate: Y = A.B + A'.B'

Name:_____Score:_____
Course/Sec:_____Schedule:_____Date Submitted:_____

B. Draw and explain the vector waveform of the constructed Verilog code for problem A.