Indian Institute of Technology Dharwad

ModelSim Guidelines and Execution Steps

1. Introduction

Verilog HDL is one of the two most common Hardware Description Languages (HDL) used by integrated circuit (IC) designers. The other one is VHDL. HDL's allows the design to be simulated earlier in the design cycle in order to correct errors or experiment with different architectures. Designs described in HDL are technology-independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.

Verilog can be used to describe designs at four levels of abstraction:
(i) Algorithmic level (much like c code with if, case and loop statements).
(ii) Register transfer level (RTL uses registers connected by Boolean equations).
(iii) Gate level (interconnected AND, NOR etc.).
(iv) Switch level (the switches are MOS transistors inside gates).

The language also defines constructs that can be used to control the input and output of simulation. Verilog is also used as an input for synthesis programs which will generate a gate-level description (a netlist) for the circuit. Some Verilog constructs are not synthesizable. Also, the way the code is written will greatly affect the size and speed of the synthesized circuit.

2. Lexical Tokens

Verilog source text files consists of the following lexical tokens:

2.1. White Space

White spaces separate words and can contain spaces, tabs, new-lines and form feeds. Thus, a statement can extend over multiple lines without special continuation characters.

2.2. Comments

Comments can be specified in two ways:
- Begin the comment with double slashes (//). All text between these characters and the end of the line will be ignored by the Verilog compiler.
- Enclose comments between the characters /* and */. Using this method allows you to continue comments on more than one line. This is good for "commenting out" many lines code, or for very brief in-line comments.

Example

```
a = c + d;     // this is a simple comment
assign y = temp_reg; /* however, this comment continues on more than one line */
```

2.3. Numbers

Number storage is defined as a number of bits, but values can be specified in binary, octal, decimal or hexadecimal

Example

3'b001, a 3-bit binary number,

5’d30, (=5’b11110),
16’h5ED4, (=16’d24276)

2.4. Identifiers

Identifiers are user-defined words for variables, function names, module names, block names and instance names. Identifiers begin with a letter or underscore (Not with a number or $) and can include any number of letters and underscores. Identifiers in Verilog are case-sensitive.

2.5. Operators

Operators are one, two and sometimes three characters used to perform operations on variables. *Example* >, +, ~, &, !=.

2.6. Verilog Keywords

These are words that have special meaning in Verilog. Some examples are assign, case, while, wire, reg, and, or, nand, and module. They should not be used as identifiers.

3. Gate-Level Modelling

Primitive logic gates are part of the Verilog language.

3.1. Basic Gates

These implement the basic logic gates. They have one output and one or more inputs. In the gate instantiation syntax shown below, GATE stands for one of the keywords and, nand, or, nor, xor, xnor.

3.2. buf, not Gates

These implement buffers and inverters, respectively. They have one input and one or more outputs.

3.3. Three-State Gates: bufif1, bufif0, notif1, notif0

These implement 3-state buffers and inverters. They propagate z (3-state or high-impedance) if their control signal is de-asserted. These can have three delay specifications: a rise time, a fall time, and a time to go into 3-state.

4. Data Types

4.1. Value Set

Verilog consists of only four basic values. Almost all Verilog data types store all these values:
0 (logic zero, or false condition)
1 (logic one, or true condition)
x (unknown logic value)
z (high impedance state)

4.2. Wire

A wire represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block. A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module. Other specific types of wires include:

wand (wired-AND): The value of a wand depends on logical AND of all the drivers connected to it.
wor (wired-OR): the value of a wor depends on logical OR of all the drivers connected to it.

tri (three-state): all drivers connected to a tri must be z, except one (which determines the value of the tri).

4.3. Reg

A reg (register) is a data object that holds its value from one procedural assignment to the next. They are used only in functions and procedural blocks. See “Wire” on p. 4 above. A reg is a Verilog variable type and does not necessarily imply a physical register. In multi-bit registers, data is stored as unsigned numbers and no sign extension is done for what the user might have thought were two’s complement numbers.

4.4. Input, Output, Inout

These keywords declare input, output and bidirectional ports of a module or task. Input and inout ports are of type wire. An output port can be configured to be of type wire, reg, wand, wor or tri. The default is wire.

5. Operators

5.1. Arithmetic Operators

These perform arithmetic operations. The + and - can be used as either unary (-z) or binary (x-y) operators.

Operators

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)
- % (modulus)

5.2. Relational Operators

Relational operators compare two operands and return a single bit 1or 0. These operators synthesize into comparators.

Operators

- < (less than)
- <= (less than or equal to)
- > (greater than)
- >= (greater than or equal to)
- == (equal to)
- != (not equal to)

5.3. Bit-wise and Reduction Operators

Bit-wise operators do a bit-by-bit comparison between two operands.

Operators

- ~ (bitwise NOT)
- & (bitwise AND)
- | (bitwise OR)
- ^ (bitwise XOR)

~^ or ^~(bitwise XNOR)

5.4. Logical Operators

Logical operators return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands. They can work on expressions, integers or groups of bits, and treat all values that are nonzero as "1". Logical operators are typically used in conditional (if ... else) statements since they work with expressions.

5.6. Shift Operators

Shift operators shift the first operand by the number of bits specified by the second operand. Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).

Operators
<< (shift left)
>> (shift right)

5.7. Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector.

Operator {}
Example:

```
wire [1:0] a, b;     wire [2:0] x;       wire [3;0] y, Z;
assign x = {1'b0, a}; // x[2]=0,  x[1]=a[1],  x[0]=a[0]
assign y = {a, b};   /* y[3]=a[1],  y[2]=a[0],  y[1]=b[1],
```

5.8. Replication Operator

The replication operator makes multiple copies of an item.

Syntax {n{item}}
Example:

```
wire [1:0] a, b;     wire [4:0] x;
assign x = {2{1'b0}, a}; // Equivalent to  x = {0,0,a }
assign y = {2{a}, 3{b}};   //Equivalent to  y = {a,a,b,b}
```

5.9. Conditional Operator

Conditional operator evaluates one of the two expressions based on a condition. It will synthesize to a multiplexer (MUX).

Operators
(cond) ? (result if cond true): (result if cond false)

5.10. Operator Precedence

Table 1 shows the precedence of operators from highest to lowest. Operators on the same level evaluate from left to right. It is strongly recommended to use parentheses to define order of precedence and improve the readability of your code.

Operator	Name
[ ]	bit-select or part-select
( )	parenthesis
!, ~	logical and bit-wise NOT
&, |, ~&, ~|, ^, ~^, ^~	reduction AND, OR, NAND, NOR, XOR, XNOR; If X=3'B101 and Y=3'B110, then X&Y=3'B100, X^Y=3'B011;
+, -	unary (sign) plus, minus; +17, -7
{ }	concatenation; {3'B101, 3'B110} = 6'B101110;
{{ }}	replication; {3{3'B110}} = 9'B110110110
*, /, %	multiply, divide, modulus; */ and % not be supported for synthesis*
+, -	binary add, subtract.
<<, >>	shift left, shift right; X<<2 is multiply by 4
<, <=, >, >=	comparisons. Reg and wire variables are taken as positive numbers.
= =, !=	logical equality, logical inequality
= = =, != =	case equality, case inequality; not synthesizable
&	bit-wise AND; AND together all the bits in a word
^, ~^, ^~	bit-wise XOR, bit-wise XNOR
|	bit-wise OR; AND together all the bits in a word
&&,	logical AND. Treat all variables as False (zero) or True (nonzero). logical OR. (7||0) is (T||F) = 1, (2||-3) is (T||T) =1, (3&&0) is (T&&F) = 0.
||	
? :	conditional. x=(cond)? T : F;

Table 1

6. Operands

6.1. Literals

Literals are constant-valued operands that can be used in Verilog expressions. The two common Verilog literals are:

(a) String: A string literal is a one-dimensional array of characters enclosed in double quotes (" ").

(b) Numeric: constant numbers specified in binary, octal, decimal or hexadecimal.

6.2. Wires, Regs, and Parameters

Wires, regs and parameters can also be used as operands in Verilog expressions.

6.3. Bit-Selects "x[3]" and Part-Selects "x[5:3]"

Bit-selects and part-selects are a selection of a single bit and a group of bits, respectively, from a wire, reg or parameter vector using square brackets "[ ]"..

Syntax

variable_name[index]

variable_name[msb:lsb]

6.4. Function Calls

The return value of a function can be used directly in an expression without first assigning it to a register or wire variable. Simply place the function call as one of the operands. Make sure you know the bit width of the return value of the function call.

Syntax

assign a = b & c & chk_bc (c, b);// chk_bc is a function function_name (argument_list)

input c,b; chk_bc = b^c;

endfunction

7. Modules

7.1. Module Declaration

A module is the principal design entity in Verilog. The first line of a module declaration specifies the name and port list (arguments). The next few lines specifies the i/o type and width of each port. The default port width is 1 bit.

Then the port variables must be declared wire, wand,. . ., reg. The default is wire. Typically inputs are wire since their data is latched outside the module. Outputs are type reg if their signals were stored inside an always or initial block.

Example:

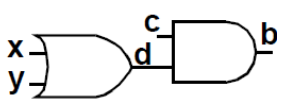
```
module add_sub(add, in1, in2, oot);
  input add;            // defaults to wire
  input [7:0] in1, in2; wire in1, in2;
  output [7:0] oot;   reg oot;
            ... statements ...
endmodule
```

7.2. Continuous Assignment

The continuous assignment is used to assign a value onto a wire in a module. It is the normal assignment outside of always or initial blocks. Continuous assignment is done with an explicit assign statement or by assigning a value to a wire during its declaration. Note that continuous assignment statements are concurrent and are continuously executed during simulation. The order of assign statements does not matter. Any change in any of the right-hand-side inputs will immediately change a left-hand-side output.

Example:

```
wire [1:0] a = 2'b01;  // assigned on declaration
assign b = c & d;       // using assign statement
assign d = x | y;
/* The order of the assign statements
does not matter. */
```



Execution Procedure

Step 1: Open Modelsim

Step 2: Click on File →New →Project

Step 3: Enter Project Name and click OK

Step 4: “Add items to Project” window will be automatically opened. Click on Create New File

Step 5 : Create Project File window will be automatically opened. Enter File Name, select Add file as type as verilog and click OK

Note : If Add items to Project window is not opened automatically, click on project tab, right click in project window area, click on Add to Project File→New File... and then add the details as mentioned in step 5.

Step 6 : Select .v file in Project window, right click on it and select Edit. Editor subwindow will be opened in Modelsim

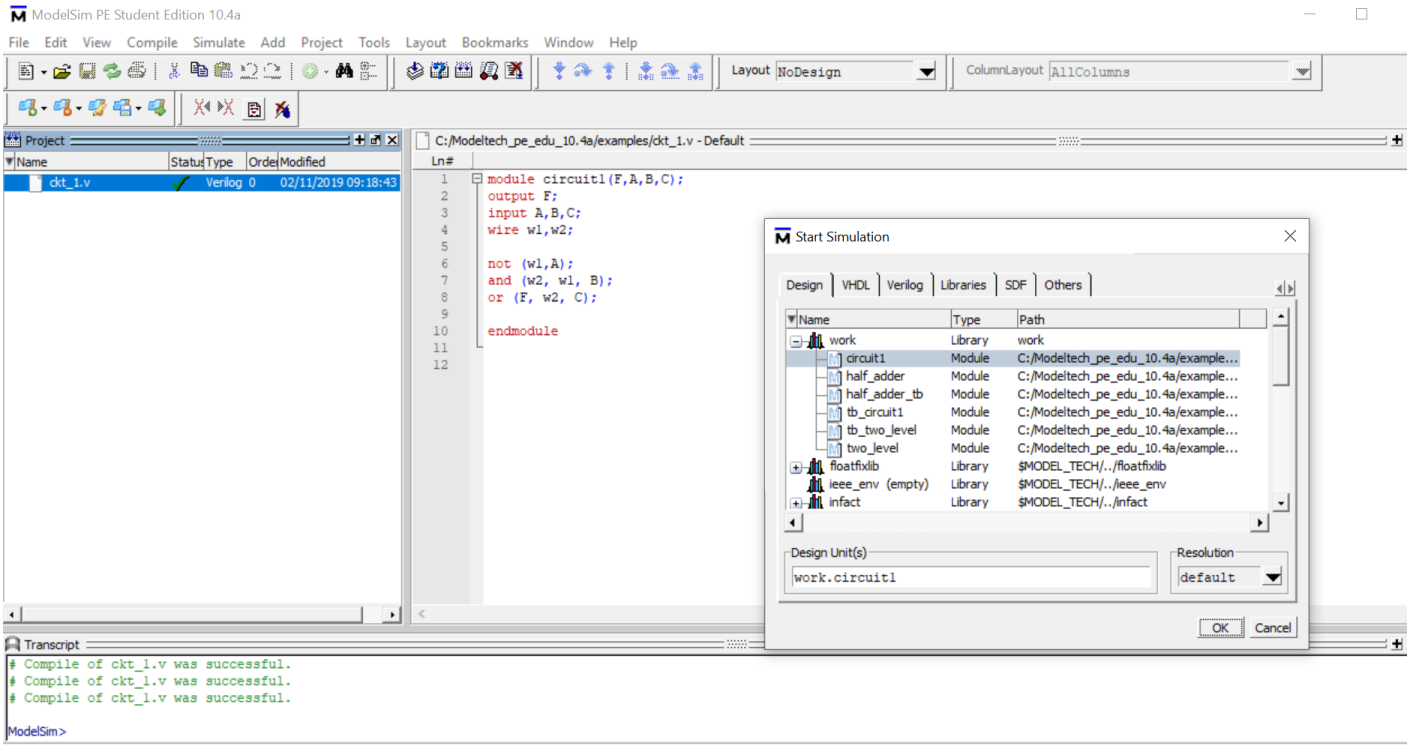
Step 7 : Enter the verilog code in Editor window and save it.
Note : The verilog file name can be different from Module name in verilog code. In fact they are different in this demonstration (Full adder G.v is file name and Full Adder G is module name. Remember that verilog is case sensitive).

Step 8 : Compilation- Right click on .v file in Project window, select Compile →Compile Selected/Compile All. If code is error-free, the code will be compiled successfully and such message can be seen in Transcript window. If any error is there in code, edit the verilog code, save it and compile it again until it is error-free.

SIMULATION WITHOUT TESTBENCH

Step 9a : Click on Simulate Tab→ Start Simulation. A window listing all the libraries would pop up.

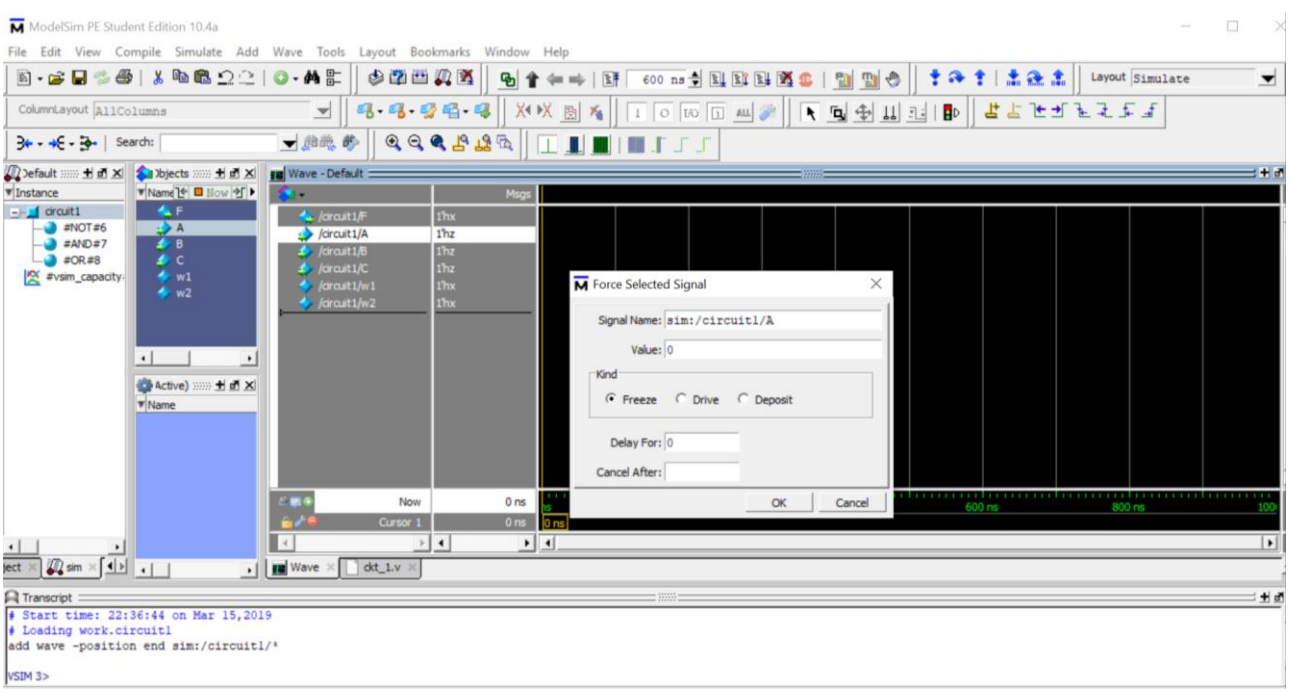
Step 9b : In the Library window, expand Work folder hierarchy. You will find module name (written in code, e.g. circuit1 in this demonstration) under it. Select module and click on OK.



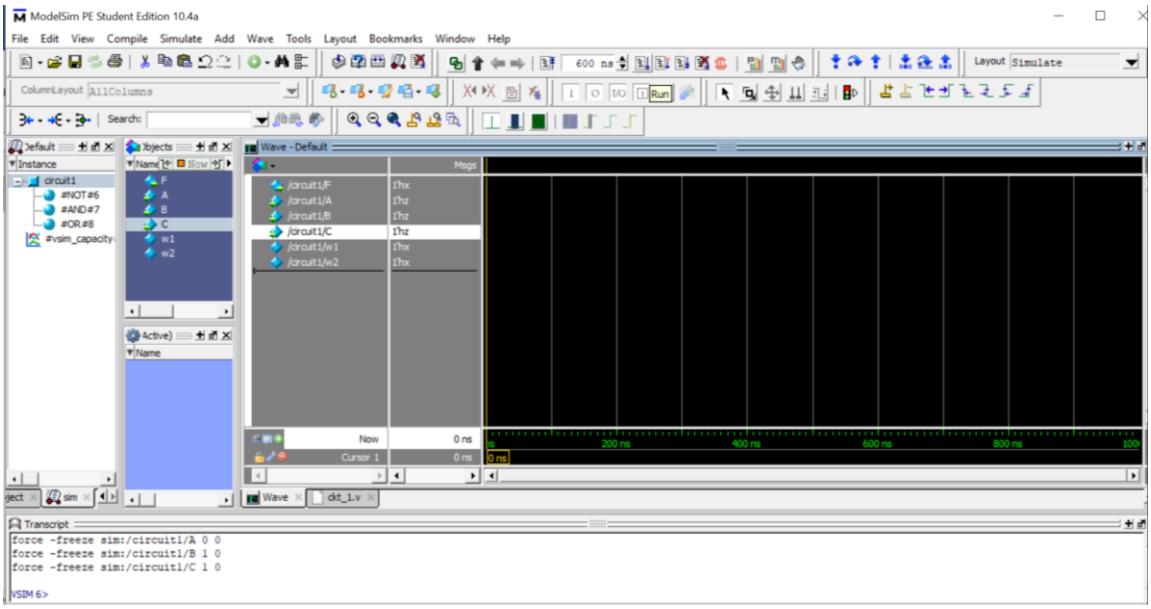
Step 9c : Three windows will be available on GUI (1) Library/Project/sim window on left (2) Objects window in the middle and (3) Wave/Editor window on the right. If any of these windows not available (e.g. Objects window and Wave window), they can be added by clicking on View option on top menu in GUI and then selecting required option. Make sim, Objects and Wave windows active by selecting them.

Step 9d : Select the signals to be plotted in Objects window, drag and drop them in Wave window.

Step 9e : Now select one input signal at a time from Wave window, right click on it and select Force option from right click menu. Force Selected Signal window will pop-up. Enter signal value in Value option in this form (either 0 or 1 in this demonstration) and click OK. Force the values on all input signals in similar manner.



Step 9e : Click on Run button (adjacent to 600 ns incrementer/decrementer option for simulation time). If clicked once simulation will be run for 600 ns, if clicked multiple times, simulation will run for multiples of 600 ns. You can adjust the timescale as per your convenience (if the implemented circuit does not have strict time restrictions because of delays etc).



Visibility of waveforms in Wave window can be adjusted with Zoom In, Zoom Out and Zoom Full buttons.

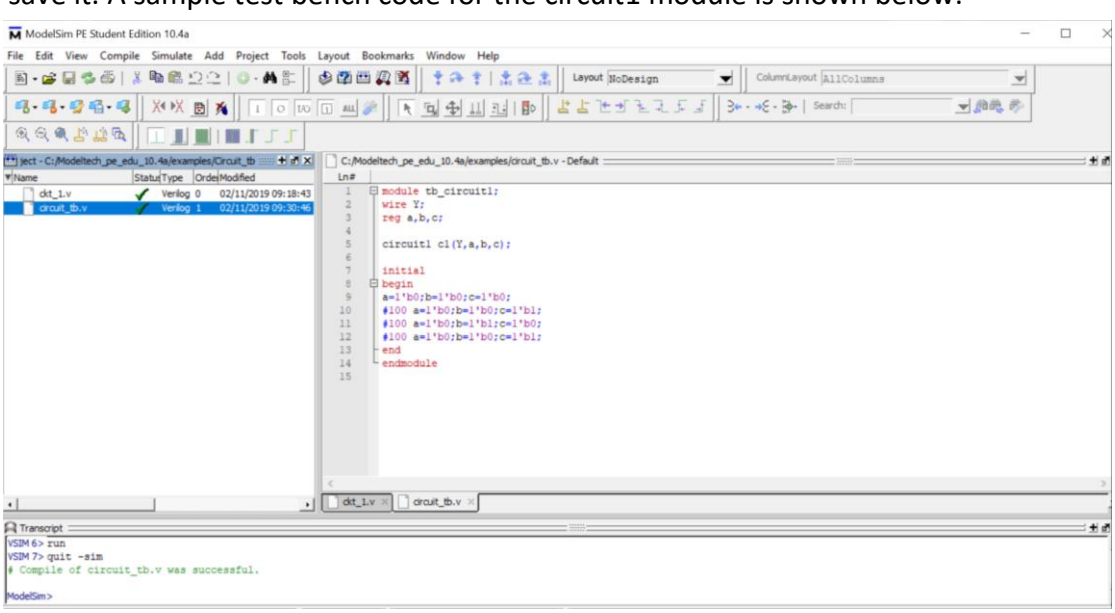
To stop simulation, select Simulate menu on top of GUI and click End Simulation option in it.

SIMULATION WITH TESTBENCH

Step 10a : In the Project window area, right click and select Add to Project File→New File.... In Create Project File form, make sure to add File Name and file type fields appropriately.

The .v file with file name entered in Step 10a will be created and listed in Project window

Step 10b : In Project window, select newly added file, and open for editing by right clicking it. Enter code in editor and save it. A sample test bench code for the circuit1 module is shown below:

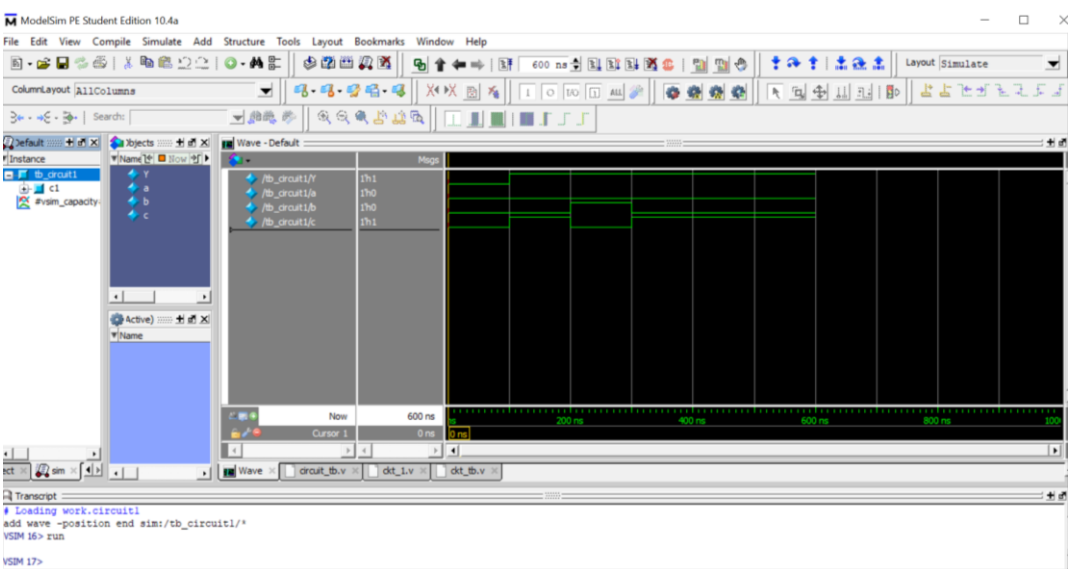


Step 10c : Compilation- Right click on .v (testbench) file in Project window, select Compile →Compile All. If code is error-free, the code will be compiled successfully and such message can be seen in Transcript window. If any error is there in code, edit the verilog code, save it and compile it again until it is error-free. Automatically other file gets compiled as it is component in testbench.

Step 10d : Click on Simulate Tab→ Start Simulation. In the Library window, expand Work folder hierarchy. You will find two modules (written in code, e.g. ckt_1 and circuit_tb. Select testbench module i.e. circuit_tb here and click OK.

Step 10e : Three windows will be available on GUI (1) Library/Project/sim window on left (2) Objects window in the middle and (3) Wave/Editor window on the right. If any of these windows not available (e.g. Objects window and Wave window), they can be added by clicking on View option on top menu in GUI and then selecting required option. Make sim, Objects and Wave windows active by selecting them. Also select the signals to be plotted in Objects window, drag and drop them in Wave window.

Step 10f : Click on Run All button to simulate it for time specified in the testbench. Inputs are taken from the initial block in testbench file.



This file describes the basics of Verilog partially. Please explore the option of User Defined Primitives in Verilog for the assignment.