











# Lab 10 – Counters and Clock Divider

Name:	ID:	Section:
-------	-----	----------

## Objectives

This lab consists of four different sections. The description of each section is given below:

Section	
a) <u>Introduction</u>  In this lab, procedural statements are introduced to design sequential circuits.	20
b) <u>Counter</u>  In this section, you will construct h_counter and v_counter for VGA controller	50
c) <u>Clock divider</u>  In this task, we will divide the system clock (100 MHz) to 25MHz	30
d) <u>Exercise1</u>   In this part, you will combine all the defined Verilog modules together to get the system.	40
<u>Exercise2</u>   In this part, you will use counters to simultaneously display different digits on LED segment	40



## a. Introduction

In previous labs, you learnt how to develop Verilog HDL modules of basic logic gates using a continuous assignment (`assign`) statement to define the behavior of a circuit. The `assign` statement runs continuously without passing through any procedure.

### i. Procedural Statements

In order to design a circuit which should execute for specific scenarios, there are other behavioral statements, categorized in Verilog as the *structured procedural statements*, called `initial` block and `always` block. These statements are the two most basic statements in *behavioral modeling*.

All other behavioral statements can appear only inside these structured procedural blocks. Each `always` and `initial` statement represents a separate activity flow in Verilog. Each activity flow starts at simulation time 0. The statements `always` and `initial` cannot be nested.

#### a. initial Statement

All statements inside an initial statement constitute an initial block. An initial block starts at time 0, executes exactly once during a simulation, and then does not execute again. If there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution independently of other blocks. The initial blocks are typically used for initialization, monitoring waveforms and other processes that must be executed only once during the entire simulation run. Multiple behavioral statements must be grouped, typically using the keywords `begin` and `end`.

```
5 initial
6   m = 1'b0; //single statement; does not need to be grouped
7 initial
8   begin
9       #5 a = 1'b1; //multiple statements; need to be grouped
10      #25 b = 1'b0;
11   end
12 endmodule
```

Figure 10. 1: Example module for initial block

In the Figure 10. 1, note that the two initial statements start to execute in *parallel* at time 0. If a delay `#<delay>` is seen before a statement, the statement is executed `<delay>` time units after the current simulation time. After the execution of code in Figure 10. 1, 'm' is assigned the value of 1'b0 at zero simulation time unit, 'a' is assigned value of 1'b1 at 5 simulation time units and 'b' is assigned value of 1'b0 at 30 (i.e. 25+5) simulation time units.



The initial block is not synthesizable in design module i.e. cannot be converted into a hardware schematic with digital elements. Its primary purpose is for settling sequence of values in simulations (i.e. test benches) and variable initializations.



### b. always Statement

The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion. This statement is used to model a block of activity that is repeated continuously in a digital circuit. An example is a clock generator module that toggles the clock signal every half cycle. In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on. Figure 10. 2 illustrates one method to model a clock generator in Verilog.

```

1 // Code your design here
2 module clock_gen (clock);
3 output clock;
4 reg clock;
5 //Initialize clock at time zero
6 initial
7   clock = 1'b0;
8 //Toggle clock every half-cycle (time period = 20)
9 always
10   #10 clock = ~clock;
11 initial
12   #1000 $finish;
13 endmodule

```

Figure 10. 2: Example module for always block

In Figure 10. 2, the always statement starts at time 0 and executes the statement `clock = ~clock` every 10 time units. Notice that the initialization of clock has to be done inside a separate initial statement. If we put the initialization of clock inside the always block, clock will be initialized every time the always is entered. Also, the simulation must be halted inside an initial statement. If there is no `$stop` or `$finish` statement to halt the simulation, the clock generator will run forever.

### ii. *Procedural Assignment*

Procedural assignments update values of reg, integer, real, or time variables. The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value. There are two types of procedural assignment statements: Blocking and Non-blocking.

#### a. Blocking Assignments

Blocking assignment statements are executed in the order, they are specified in a sequential block. The '=' operator is used to specify blocking assignments.

```

6 x = 0; y = 1;           //Scalar assignments
7 count = 0;             //Assignment to integer variables
8 count = count + 1;      //Assignment to an integer (increment)

```

Figure 10. 3: Examples of blocking assignment

In Figure 10. 3, the statement `y = 1` is executed only after `x = 0` is executed and `count = 0` is executed after `y = 1` is executed. The statement `count = count + 1` is executed last. The behavior



in a particular block is sequential in a begin-end block if blocking statements are used, because the statements can execute only in sequence.

### b. Non-blocking Assignments

Non-blocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block. A '<=' operator is used to specify non-blocking assignments.

```

5 reg x, y;
6 reg [15:0] reg_a, reg_b;
7 integer count, z;
8
9 initial
10 begin
11     x = 0; y = 1; z = 0; // blocking
12     count = 1; // blocking
13     count <= count + 1; // non-blocking
14     z <= count + y; // non-blocking
15 end

```

Figure 10. 4: Example for non-blocking assignment

In Figure 10. 4, the statements  $x = 0$  through  $count = 1$  are executed sequentially starting at time 0. Then the non-blocking assignment is processed at the same simulation time. After the execution of above code  $count$  will have value of 2 and  $z$  will also have value of 2.

### iii. *Delay based Timing Control*

Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed. Delays are specified by the symbol #. It is specified as a non-zero delay to the left of a procedural assignment.

```

4 reg x, y;
5 initial
6 begin
7     x = 0; // no delay control
8     #10 y = 1; // Delay execution of y = 1 by 10 units
9 end

```

Figure 10. 5: Example for delay based timing control

In Figure 10. 5, the execution of a procedural assignment is delayed by the number specified by the delay control. Delay is always relative to time when the statement is encountered. Thus,  $y = 1$  is executed after 10 time units once it is encountered in the activity flow.

Simulation time units can be defined by following compiler directive. Below command define one nanosecond as simulation time unit.

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



#### iv. Event based Timing Control

An event is the change in the value on a register or a net. Events can be utilized to trigger execution of a statement or a block of statements. The '@' symbol is used to specify an event control. Statements can be executed on changes in signal value or at a positive or negative transition of the signal value.

When the signal transits from level 0 to 1, that transition is known as positive or rising edge of the signal as shown in Figure 10. 6(a). Similarly, when the signal transits from level 1 to 0, it is known as negative or falling edge of the clock as shown in Figure 10. 6(b).

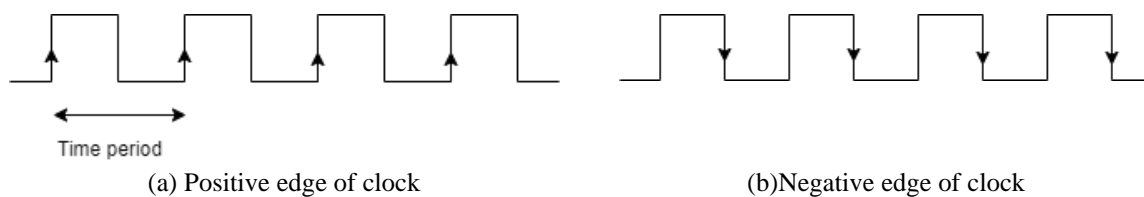


Figure 10. 6: Clock signal

In Verilog, the keyword 'posedge' is used for a positive transition and 'negedge' for negative transition. Figure 10. 7 shows some event based control examples.

```

25 @(clock) q = d;           //q = d is executed whenever signal clock changes value
26 @(posedge clock) q = d;    //q = d is executed whenever signal clock does a positive transition
27 @(negedge clock) q = d;    //q = d is executed whenever signal clock does a negative transition
28 always @(posedge clock)    // always block is executed at positive edge of clock
29   begin
30     ...
31   end

```

Figure 10. 7: Event based timing control

#### v. Clock Signal

A clock signal, shown in the figure below, is a particular type of signal that oscillates between a high and a low state and is used to coordinate actions of digital circuits. The period between the two consecutive positive edges or two consecutive negative edges is called a Time Period and is denoted with T (refer Figure 10. 6). Therefore, the frequency at which the digital circuit operates is defined by  $f = 1/T$ . The Basys3 board operates at 100MHz clock frequency. In order to have lower clock frequency, clock divider module is designed using counters.

In the next two sections, you'll implement the counter and clock divider circuit and then connect them together to work as a single module. This lab forms the foundation for VGA controller, the counters and clock divider design modules will be used in next lab.

#### b. Counter

A counter is a sequential circuit that increments its count value upon each positive edge of the clock pulse. Counters are used in digital electronics for counting purpose, they can count specific event happening in the circuit.

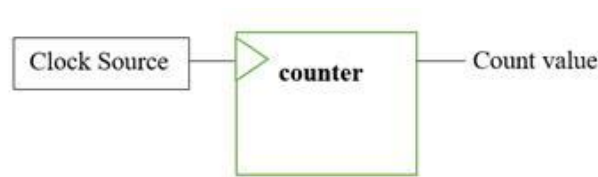


Figure 10. 8: Counter block

### Implementations of 4-bit Counter

A 4-bit counter counts up to  $2^4 - 1$  count value. Its Verilog implementation is shown in Figure 10. 9(a). For each positive edge of clock pulse, the count register increments until its value is 15. Once the count value exceeds 15 the count value is reset to 0.

```

1 // Code your design here
2 module counter_4bit(clk,count);
3   input clk;
4   output [3:0] count;
5   reg [3:0] count;
6   initial count = 0;
7   always @ (posedge clk)
8   begin
9     if (count <= 15)
10    begin
11      count <= count + 1;
12    end
13    else
14    begin
15      count <= 0;
16    end
17  end
18 endmodule

```

(a)

```

1 // Code your testbench here
2 // or browse Examples
3 module tb();
4   reg clk;
5   wire [3:0] count;
6   counter_4bit c1(.clk(clk),.count(count));
7
8   initial
9     clk = 1'b0;
10
11  always
12    #5 clk = ~clk;
13
14  initial
15    begin
16      $dumpfile("dump.vcd");
17      $dumpvars(1,tb);
18      $monitor("Time = ", $time, " Count = %d", count);
19      #200 $finish;
20    end
21
22 endmodule

```

(b)

Figure 10. 9: 4-bit counter module (a) design (b) testbench

### Task a:

Develop a 10-bit counter module, named `h_counter`, which should have following features:

- Input: 1-bit `clk` signal
- Outputs: 10-bit `h_count` signal and 1-bit `trig_v` signal.
- Increment the counter output (`h_count`) at every positive edge of clock signal (`clk`) and counts up to 799.
- The counter resets to 0 if the value exceeds 799.
- `trig_v` is set to 1 when counter resets.
- `trig_v` would be zero if count is greater than 1 after reset.

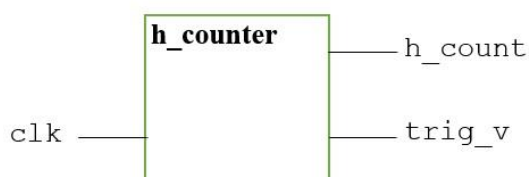


Figure 10. 10: h\_counter module

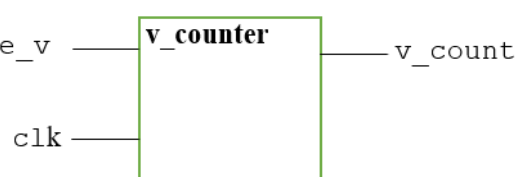


Figure 10. 11: v\_counter module



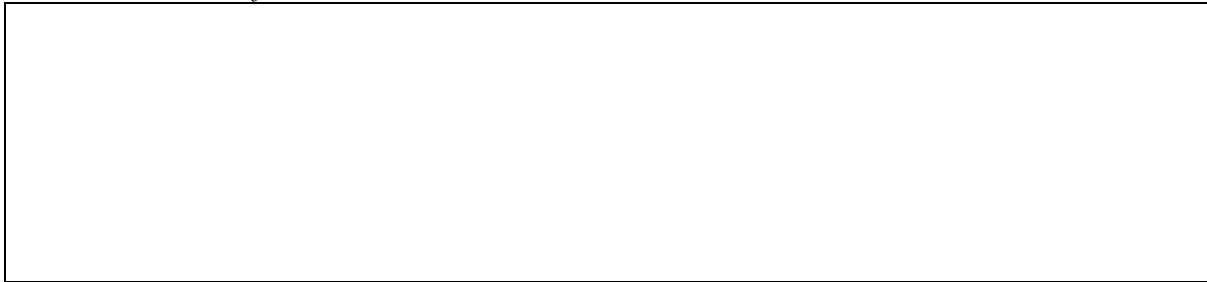
*Provide appropriately commented code for your h\_count design module*

Modify the testbench provided in Figure 10. 9(b) to verify the functionality of your designed h\_counter module.

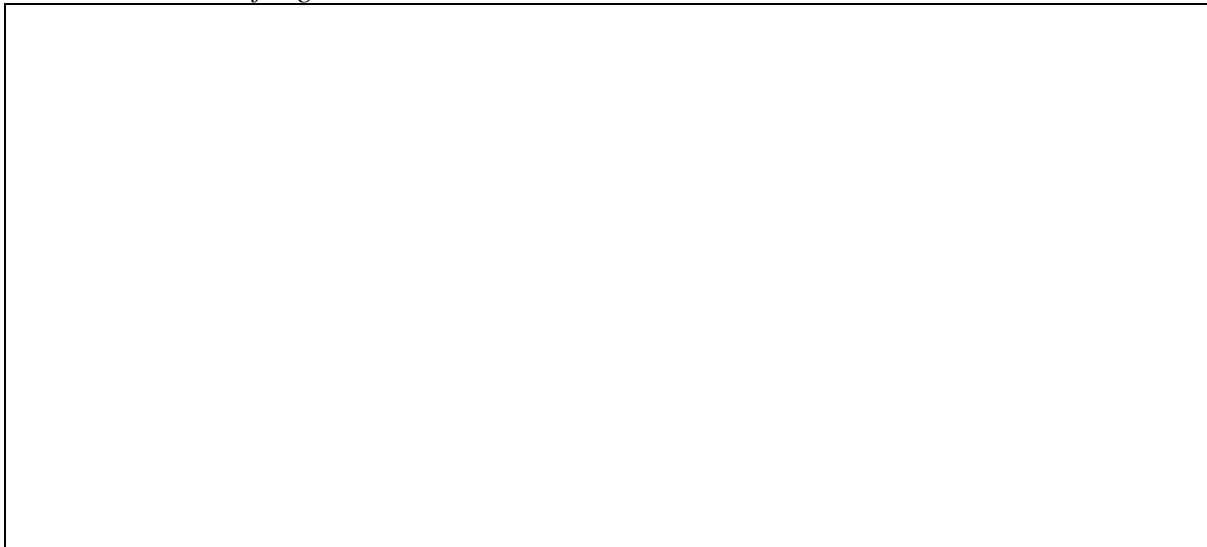
*Attach your testbench here.*



*Attach screenshot of EP wave here*



*Attach screenshot of log-window here*





*Concept check:*

Assuming that a time unit, in above mentioned testbench, is defined in nano seconds; what is the frequency at which the designed `h_counter` module would operate?

---

*Task b:*

Develop a 10-bit counter module, named `v_counter`, which should have following features:

- a. Inputs: 1-bit `clk` and `enable_v` signal
- b. Outputs: 10-bit `v_count` signal
- c. Increment the counter output (`v_count`) at every positive edge of clock signal (`clk`) and counts up to 524, only if `enable_v` signal is 1.
- d. If `enable_v` signal is low, counter does not increment, but only maintains its current state.

*Provide appropriately commented code for your `v_count` design module*

Modify the testbench provided in Figure 10. 9(b) to verify the functionality of your designed `v_counter` module.

*Attach your testbench here.*



*Attach screenshot of EP wave here*

*Attach screenshot of log-window here*



### c. Clock Divider

Basys3 board uses 100MHz clock ( $Time\ period = 1/(100 \times 10^6) = 10ns$ ) and the counter output is supposed to be updated at every positive edge of clock. While for driving VGA output, we require clock frequency of 25MHz. Therefore, we need to slow down the process by dividing the clock. In order to observe the output, we will run the counter at 25MHz (40ns).

#### Task c:

Write down the clock divider module shown in Figure 10. 12. This module has one input `clk`, and one output `clk_d`. In this experiment, the `clk` input will take the default clock (of 100 MHz).

```

1 // Code your design here
2 module clk_div (clk, clk_d);
3     parameter div_value = 1;
4     input clk;
5     output clk_d;
6     reg clk_d;
7     reg count;
8     initial
9     begin
10        clk_d = 0;
11        count = 0;
12    end
13    always @(posedge clk)
14    begin
15        if (count == div_value)
16            count <= 0;           // reset count
17        else
18            count <= count + 1;   // count up
19    end
20
21    always @(posedge clk)
22    begin
23        if (count == div_value)
24            clk_d <= ~ clk_d;     // toggle
25    end
26
27 endmodule

```

Figure 10. 12

Verilog allows user to define a constant value with a help of a keyword `parameter`, as shown in line 3. A register is defined with the name `count`, which counts to a value of `div_val = 1`. The value of one-bit output `clk_d` is toggled every time when the counter becomes equal to `div_val` as shown in Figure 10. 13.

`div_val` is calculated using formula

$$div_{value} = \frac{input\ clock\ frequency}{2 \times desired\ clock\ frequency} - 1$$

Here input clock frequency is 100MHz and desired frequency is 25MHz.

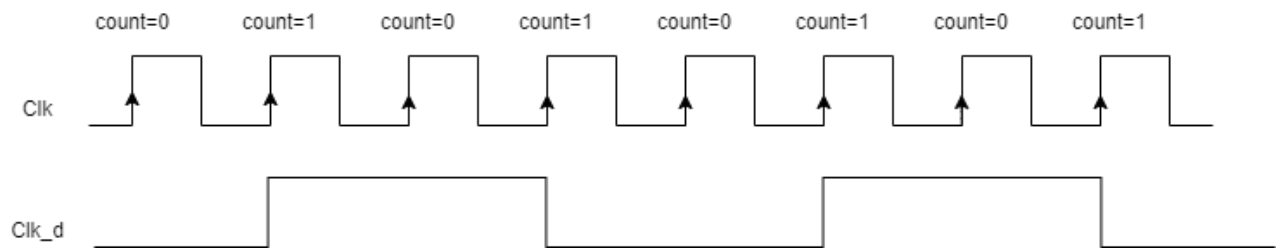


Figure 10. 13



### Exercise 1

Write a Verilog module `TopLevelModule` that combines all the blocks from this lab according to the block diagram provided in Figure 10. 14. This module inputs `clk` signal and outputs `h_count` and `v_count` value.

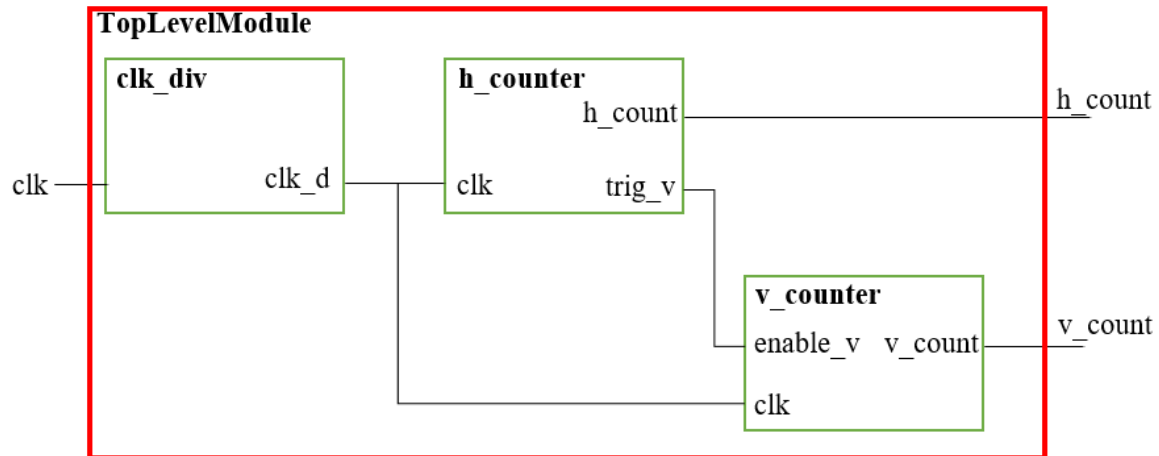


Figure 10. 14

*Provide appropriately commented code for your top-level design module*

Write a testbench to verify its functionality.

*Attach the testbench here.*



*Attach screenshot of EP wave here*

*Attach screenshot of log-window here*



## Exercise 2

In your top module for Lab 09, make modifications to display all digits on all 4 of the displays simultaneously.

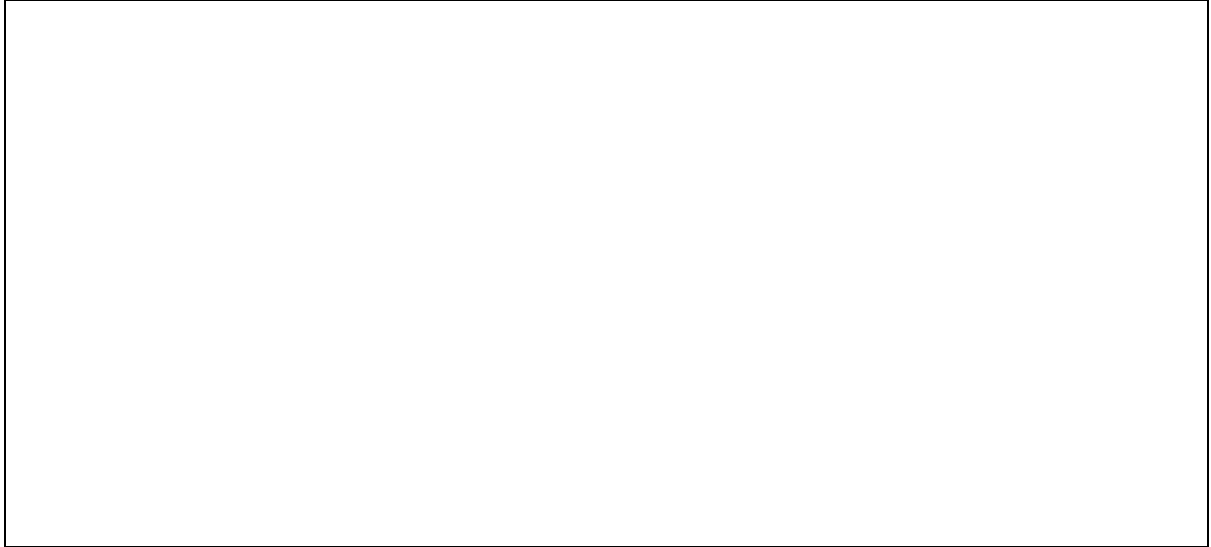
Add a 2bit counter in your top module that counts from 0 to 3 inclusive. Pass that value as select input for your mux and demux. The counter should increment at every positive edge of clk, same as how h\_counter behaves.

If done correctly, you should see all 4 segments showing you the 4 different digits.

You have already seen for yourself that at a given time only 1 segment can be active and you can see in your code that only 1 is active at a give time as well. But they are changing so quickly that your eyes are unable to pick that up and you perceive all of them to be active together. If you are curious about this mind game, check out this video <https://www.youtube.com/watch?v=3BJU2drrtCM> to see how CRT TVs actually work.

*Provide appropriately commented code for your top-level design module*

*Attach picture of FPGA here*





## Assessment Rubrics

### Marks Distribution:

		<b>LR2 Code</b>	<b>LR5 Results</b>	<b>LR7 Viva</b>
<b>In-lab</b>	<b>Task a</b>	10 points	5 points	10 points
	<b>Task b</b>	10 points	10 points	
	<b>Task c</b>	-	5 points	
<b>Exercise 1</b>		20 points	10 points	
<b>Exercise 2</b>		10 points	10 points	
<b>Total</b>				

### Marks Obtained:

		<b>LR2 Code</b>	<b>LR5 Results</b>	<b>LR7 Viva</b>
<b>In-lab</b>	<b>Task a</b>			
	<b>Task b</b>			
	<b>Task c</b>	-		
<b>Exercise 1</b>				
<b>Exercise 2</b>				
<b>Obtained</b>				

## Lab Evaluation Rubrics

#	Assessment Elements	Level 1: Unsatisfactory	Level 2: Developing	Level 3: Good	Level 4: Exemplary
LR2	<b>Program/Code / Simulation Model/ Network Model</b>	Program/code/simulation model/network model does not implement the required functionality and has several errors. The student is not able to utilize even the basic tools of the software.	Program/code/simulation model/network model has some errors and does not produce completely accurate results. Student has limited command on the basic tools of the software.	Program/code/simulation model/network model gives correct output but not efficiently implemented or implemented by computationally complex routine.	Program/code/simulation /network model is efficiently implemented and gives correct output. Student has full command on the basic tools of the software.
LR5	<b>Results &amp; Plots</b>	Figures/ graphs / tables are not developed or are poorly constructed with erroneous results. Titles, captions, units are not mentioned. Data is presented in an obscure manner.	Figures, graphs and tables are drawn but contain errors. Titles, captions, units are not accurate. Data presentation is not too clear.	All figures, graphs, tables are correctly drawn but contain minor errors or some of the details are missing.	Figures / graphs / tables are correctly drawn and appropriate titles/captions and proper units are mentioned. Data presentation is systematic.
LR7	<b>Viva</b>	Response shows a complete lack of understanding of the assigned / completed task	Response shows shallow understanding of the assigned task	Response shows substantial understanding of the assigned task	Response shows complete understanding of the completed task. The student is able to explain all the related concepts.